NOTE: The Intel 64 and IA-32 Architectures Software Developer’s Manual consists of four volumes: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.
Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting http://www.intel.com/design/literature.htm.

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 1997-2018, Intel Corporation. All Rights Reserved.
3.2 INSTRUCTIONS (A-L) ...................................................... 3-18

3.1.1.19 Operation Section .............................................................. 3-10
3.1.1.20 Intel® C/C++ Compiler Intrinsics Equivalents Section ................... 3-13
3.1.1.21 Flags Affected Section ...................................................... 3-15
3.1.1.22 FPU Flags Affected Section .............................................. 3-15
3.1.1.23 Protected Mode Exceptions Section .................................... 3-15
3.1.1.24 Real-Address Mode Exceptions Section ................................ 3-16
3.1.1.25 Virtual-8086 Mode Exceptions Section ................................. 3-16
3.1.1.26 Floating-Point Exceptions Section ..................................... 3-17
3.1.1.27 SIMD Floating-Point Exceptions Section ............................... 3-17
3.1.1.28 Compatibility Mode Exceptions Section ............................... 3-17
3.1.1.29 64-Bit Mode Exceptions Section ....................................... 3-17

3.1.2 INSTRUCTIONS (A-L) ...................................................... 3-18

AAA—ASCII Adjust After Addition ................................................. 3-19
AAD—ASCII Adjust AX Before Division ............................................ 3-21
AAM—ASCII Adjust AX After Multiply ............................................ 3-23
AAS—ASCII Adjust AL After Subtraction .......................................... 3-25
ADC—Add with Carry ................................................................. 3-27
ADDCX — Unsigned Integer Addition of Two Operands with Carry Flag .......... 3-30
ADD—Add ................................................................................. 3-32
ADDP—Add Packed Double-Precision Floating-Point Values ...................... 3-34
ADDPX—Add Packed Single-Precision Floating-Point Values ........................ 3-37
ADDS—Add Scalar Double-Precision Floating-Point Values ........................ 3-40
ADDSS—Add Scalar Single-Precision Floating-Point Values ....................... 3-42
ADDSUBP—Packed Double-FP Add/Subtract ..................................... 3-44
ADDSUBP—Packed Single-FP Add/Subtract ..................................... 3-46
adox — Unsigned Integer Addition of Two Operands with Overflow Flag .......... 3-49
AESDEC—Perform One Round of an AES Decryption Flow .......................... 3-51
AESDECLAST—Perform Last Round of an AES Decryption Flow ................. 3-52
AESENC—Perform One Round of an AES Encryption Flow ........................ 3-53
AESENCLAST—Perform Last Round of an AES Encryption Flow .................. 3-54
AESIMC—Perform the AES InvMixColumn Transformation .......................... 3-59
AESKEYGENASSIST—AES Round Key Generation Assist ............................. 3-60
AND—Logical AND ................................................................. 3-62
ANDN—Logical AND NOT .......................................................... 3-64
ANDDP—Bitwise Logical AND of Packed Double Precision Floating-Point Values .......................................................... 3-64
ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values .......................................................... 3-65
ANDNP—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values .......................................................... 3-66
ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values .......................................................... 3-67
ARPL—Adjust RPL Field of Segment Selector ..................................... 3-72
BLENDP—Blend Packed Single Precision Floating-Point Values .................. 3-79
BLENDP—Blend Packed Single Precision Floating-Point Values .................. 3-81
BLENDP—Variable Blend Packed Double Precision Floating-Point Values .... 3-82
BLENDP—Variable Blend Packed Single PrecisionFloating-Point Values ........ 3-84
BLSI—Extract Lowest Set Isolated Bit ............................................. 3-89
BLSMK—Get Mask Up to Lowest Set Bit .......................................... 3-90
BSR—Bit Field Extract .............................................................. 3-91
BNDCW—Check Upper Bound ..................................................... 3-92
BNDCW/BNDNC—Check Upper Bound ............................................. 3-94
BDLX—Load Extended Bounds Using Address Translation ......................... 3-96
BDLMK—Make Bounds .............................................................. 3-99
BDMOV—Move Bounds .............................................................. 3-101
BDSTL—Store Extended Bounds Using Address Translation ...................... 3-104
BOUND—Check Array Index Against Bounds .................................... 3-107
BSF—Bit Scan Forward .............................................................. 3-109
BSR—Bit Scan Reverse .............................................................. 3-111
BSWAP—Byte Swap ................................................................. 3-113
BT—Bit Test ............................................................................. 3-114
BTC—Bit Test and Complement ..................................................... 3-116
<table>
<thead>
<tr>
<th>Operation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTR—Bit Test and Reset</td>
<td>3-118</td>
</tr>
<tr>
<td>BTS—Bit Test and Set</td>
<td>3-120</td>
</tr>
<tr>
<td>BZH—Zero High Bits Starting with Specified Bit Position</td>
<td>3-122</td>
</tr>
<tr>
<td>CALL—Call Procedure</td>
<td>3-123</td>
</tr>
<tr>
<td>CBW/CWD/CQDE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword</td>
<td>3-136</td>
</tr>
<tr>
<td>CLAC—Clear AC Flag in EFLAGS Register</td>
<td>3-137</td>
</tr>
<tr>
<td>CLC—Clear Carry Flag</td>
<td>3-138</td>
</tr>
<tr>
<td>CLD—Clear Direction Flag</td>
<td>3-139</td>
</tr>
<tr>
<td>CLFLUSH—Flush Cache Line</td>
<td>3-140</td>
</tr>
<tr>
<td>CLFLUSHOPT—Flush Cache Line Optimized</td>
<td>3-142</td>
</tr>
<tr>
<td>CLI—Clear Interrupt Flag</td>
<td>3-144</td>
</tr>
<tr>
<td>CLTS—Clear Task-Switched Flag in CR0</td>
<td>3-146</td>
</tr>
<tr>
<td>CLWB—Cache Line Write Back</td>
<td>3-147</td>
</tr>
<tr>
<td>CM—Complement Carry Flag</td>
<td>3-149</td>
</tr>
<tr>
<td>CMOVcc—Conditional Move</td>
<td>3-150</td>
</tr>
<tr>
<td>CMP—Compare Two Operands</td>
<td>3-154</td>
</tr>
<tr>
<td>CMPDP—Compare Packed Double-Precision Floating-Point Values</td>
<td>3-156</td>
</tr>
<tr>
<td>CMPSS—Compare Packed Single-Precision Floating-Point Values</td>
<td>3-163</td>
</tr>
<tr>
<td>CMPS/CMPX/MPSW/CMPX/CMPSD/CMPXQ—Compare String Operands</td>
<td>3-170</td>
</tr>
<tr>
<td>CMPSD—Compare Scalar Double-Precision Floating-Point Value</td>
<td>3-174</td>
</tr>
<tr>
<td>CMPSS—Compare Scalar Single-Precision Floating-Point Value</td>
<td>3-178</td>
</tr>
<tr>
<td>CMPXCHG—Compare and Exchange</td>
<td>3-182</td>
</tr>
<tr>
<td>CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes</td>
<td>3-184</td>
</tr>
<tr>
<td>CMSSD—Compare Scalar Ordered Double-Precision Floating-Point Values Set EFLAGS</td>
<td>3-187</td>
</tr>
<tr>
<td>CMISS—Compare Scalar Ordered Single-Precision Floating-Point Values Set EFLAGS</td>
<td>3-189</td>
</tr>
<tr>
<td>CPUID—CPU Identification</td>
<td>3-191</td>
</tr>
<tr>
<td>CRC32—Accumulate CRC32 Value</td>
<td>3-228</td>
</tr>
<tr>
<td>CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</td>
<td>3-231</td>
</tr>
<tr>
<td>CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</td>
<td>3-235</td>
</tr>
<tr>
<td>CVTDQ2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td>3-238</td>
</tr>
<tr>
<td>CVTDQ2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers</td>
<td>3-242</td>
</tr>
<tr>
<td>CVTDQ2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values</td>
<td>3-243</td>
</tr>
<tr>
<td>CVTPD2PI—Convert Packed Dword Integers to Packed Single-Precision FP Values</td>
<td>3-247</td>
</tr>
<tr>
<td>CVTPD2PS—Convert Packed Dword Integers to Packed Double-Precision Floating-Point Values</td>
<td>3-248</td>
</tr>
<tr>
<td>CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values</td>
<td>3-249</td>
</tr>
<tr>
<td>CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values</td>
<td>3-252</td>
</tr>
<tr>
<td>CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers</td>
<td>3-255</td>
</tr>
<tr>
<td>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</td>
<td>3-256</td>
</tr>
<tr>
<td>CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value</td>
<td>3-258</td>
</tr>
<tr>
<td>CVTSD2SI—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</td>
<td>3-260</td>
</tr>
<tr>
<td>CVTSD2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</td>
<td>3-262</td>
</tr>
<tr>
<td>CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value</td>
<td>3-264</td>
</tr>
<tr>
<td>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</td>
<td>3-266</td>
</tr>
<tr>
<td>CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td>3-268</td>
</tr>
<tr>
<td>CVTTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers</td>
<td>3-272</td>
</tr>
<tr>
<td>CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values</td>
<td>3-273</td>
</tr>
<tr>
<td>CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values</td>
<td>3-276</td>
</tr>
<tr>
<td>CVTTS2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer</td>
<td>3-277</td>
</tr>
<tr>
<td>CVTTS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer</td>
<td>3-279</td>
</tr>
<tr>
<td>CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword</td>
<td>3-281</td>
</tr>
<tr>
<td>DAA—Decimal Adjust AL after Addition</td>
<td>3-282</td>
</tr>
<tr>
<td>DAS—Decimal Adjust AL after Subtraction</td>
<td>3-284</td>
</tr>
<tr>
<td>DEC—Decrement by 1</td>
<td>3-286</td>
</tr>
<tr>
<td>DIV—Unsigned Divide</td>
<td>3-288</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVPD</td>
<td>Divide Packed Double-Precision Floating-Point Values</td>
<td>3-291</td>
</tr>
<tr>
<td>DIVPS</td>
<td>Divide Packed Single-Precision Floating-Point Values</td>
<td>3-294</td>
</tr>
<tr>
<td>DIVSD</td>
<td>Divide Scalar Double-Precision Floating-Point Value</td>
<td>3-297</td>
</tr>
<tr>
<td>DIVSS</td>
<td>Divide Scalar Single-Precision Floating-Point Values</td>
<td>3-299</td>
</tr>
<tr>
<td>DPPD</td>
<td>Dot Product of Packed Double Precision Floating-Point Values</td>
<td>3-301</td>
</tr>
<tr>
<td>DPPS</td>
<td>Dot Product of Packed Single Precision Floating-Point Values</td>
<td>3-303</td>
</tr>
<tr>
<td>EMMS</td>
<td>Empty MMX Technology State</td>
<td>3-306</td>
</tr>
<tr>
<td>ENTER</td>
<td>Make Stack Frame for Procedure Parameters</td>
<td>3-307</td>
</tr>
<tr>
<td>EXTRACTS</td>
<td>Extract Packed Floating-Point Values</td>
<td>3-310</td>
</tr>
<tr>
<td>F2XM1</td>
<td>Compute 2x-1</td>
<td>3-312</td>
</tr>
<tr>
<td>FABS</td>
<td>Absolute Value</td>
<td>3-314</td>
</tr>
<tr>
<td>FADD</td>
<td>ADD/FADDP/FIADD—Add</td>
<td>3-315</td>
</tr>
<tr>
<td>FBLD</td>
<td>Load Binary Coded Decimal</td>
<td>3-318</td>
</tr>
<tr>
<td>FBSTP</td>
<td>Store BCD Integer and Pop</td>
<td>3-320</td>
</tr>
<tr>
<td>FCHS</td>
<td>Change Sign</td>
<td>3-322</td>
</tr>
<tr>
<td>FCLEX</td>
<td>Clear Exceptions</td>
<td>3-324</td>
</tr>
<tr>
<td>FCMOVcc</td>
<td>Floating-Point Conditional Move</td>
<td>3-326</td>
</tr>
<tr>
<td>FCOM</td>
<td>FCOMP/FCOMPP—Compare Floating Point Values</td>
<td>3-328</td>
</tr>
<tr>
<td>FCOMIP</td>
<td>FUCOM/FUCOMP—Compare Floating Point Values and Set EFLAGS</td>
<td>3-331</td>
</tr>
<tr>
<td>FCOS</td>
<td>Cosine</td>
<td>3-334</td>
</tr>
<tr>
<td>FDECSTP</td>
<td>Decrement Stack-Top Pointer</td>
<td>3-336</td>
</tr>
<tr>
<td>FDIV</td>
<td>Divide/FDIVP/FIDIV—Divide</td>
<td>3-337</td>
</tr>
<tr>
<td>FDIVR</td>
<td>Reverse Divide/FDIVRP/FDIVR—Reverse Divide</td>
<td>3-340</td>
</tr>
<tr>
<td>FFREE</td>
<td>Free Floating-Point Register</td>
<td>3-343</td>
</tr>
<tr>
<td>FICOMP</td>
<td>Compare Integer</td>
<td>3-344</td>
</tr>
<tr>
<td>FICOMP</td>
<td>Compare Integer</td>
<td>3-344</td>
</tr>
<tr>
<td>FILD</td>
<td>Load Integer</td>
<td>3-346</td>
</tr>
<tr>
<td>FINSTP</td>
<td>Increment Stack-Top Pointer</td>
<td>3-348</td>
</tr>
<tr>
<td>FINIT</td>
<td>Initialize Floating-Point Unit</td>
<td>3-349</td>
</tr>
<tr>
<td>FIST</td>
<td>Store Integer</td>
<td>3-351</td>
</tr>
<tr>
<td>FISTTP</td>
<td>Store Integer with Truncation</td>
<td>3-354</td>
</tr>
<tr>
<td>FLD</td>
<td>Load Floating Point Value</td>
<td>3-356</td>
</tr>
<tr>
<td>FLDL1/FDLD2T/FLDL2E/FLDL2I/FLDLG2/FLDZN2/FLDZ</td>
<td>Load Constant</td>
<td>3-358</td>
</tr>
<tr>
<td>FLDCw</td>
<td>Load x87 FPU Control Word</td>
<td>3-360</td>
</tr>
<tr>
<td>FLDENV</td>
<td>Load x87 FPU Environment</td>
<td>3-362</td>
</tr>
<tr>
<td>FMUL/FMULP/FIMUL—Multiply</td>
<td>3-364</td>
<td></td>
</tr>
<tr>
<td>FNOP</td>
<td>No Operation</td>
<td>3-367</td>
</tr>
<tr>
<td>FPATAN</td>
<td>Partial Arctangent</td>
<td>3-368</td>
</tr>
<tr>
<td>FPREM</td>
<td>Partial Remainder</td>
<td>3-370</td>
</tr>
<tr>
<td>FPREM1</td>
<td>Partial Remainder</td>
<td>3-372</td>
</tr>
<tr>
<td>FPTAN</td>
<td>Partial Tangent</td>
<td>3-374</td>
</tr>
<tr>
<td>FREDINT</td>
<td>Round to Integer</td>
<td>3-376</td>
</tr>
<tr>
<td>FRSTOR</td>
<td>Restore x87 FPU State</td>
<td>3-377</td>
</tr>
<tr>
<td>FSADD/FSNSAVE</td>
<td>Store x87 FPU State</td>
<td>3-379</td>
</tr>
<tr>
<td>FSCALE</td>
<td>Scale</td>
<td>3-382</td>
</tr>
<tr>
<td>FSIN</td>
<td>Sine</td>
<td>3-384</td>
</tr>
<tr>
<td>FSINCOS</td>
<td>Sine and Cosine</td>
<td>3-386</td>
</tr>
<tr>
<td>FSQRT</td>
<td>Square Root</td>
<td>3-388</td>
</tr>
<tr>
<td>FST/FSTP</td>
<td>Store Floating Point Value</td>
<td>3-390</td>
</tr>
<tr>
<td>FSTCW/FNSTCW</td>
<td>Store x87 FPU Control Word</td>
<td>3-392</td>
</tr>
<tr>
<td>FSTENV/FNSTENV</td>
<td>Store x87 FPU Environment</td>
<td>3-394</td>
</tr>
<tr>
<td>FSTSW/FNSTSW</td>
<td>Store x87 FPU Status Word</td>
<td>3-396</td>
</tr>
<tr>
<td>FSUB/FSUBP/FISUB—Subtract</td>
<td>3-398</td>
<td></td>
</tr>
<tr>
<td>FSUBL/FISUBP/FISUBR—Reverse Subtract</td>
<td>3-401</td>
<td></td>
</tr>
<tr>
<td>FTST/TEST</td>
<td></td>
<td>3-404</td>
</tr>
<tr>
<td>FUCOM/FUCOMP/FUCOMP—Unordered Compare Floating Point Values</td>
<td>3-406</td>
<td></td>
</tr>
<tr>
<td>FXAM/EXamine Floating-Point</td>
<td>3-409</td>
<td></td>
</tr>
<tr>
<td>FXCH/Exchange Register Contents</td>
<td>3-411</td>
<td></td>
</tr>
<tr>
<td>FXRSTOR</td>
<td>Restore x87 FPU, MMX, XMM, and MXCSR State</td>
<td>3-413</td>
</tr>
<tr>
<td>FXSAVE/Save x87 FPU, MMX Technology, and SSE State</td>
<td>3-416</td>
<td></td>
</tr>
</tbody>
</table>
4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM .......................... 4-1
4.1.1 General Description ............................................ 4-1
4.1.2 Source Data Format ............................................... 4-2
4.1.3 Aggregation Operation .......................................... 4-2
4.1.4 Polarity ............................................................ 4-3
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULSS—Multiply Scalar Single-Precision Floating-Point Values</td>
<td>4-154</td>
</tr>
<tr>
<td>MULX — Unsigned Multiply without Affecting Flags.</td>
<td>4-156</td>
</tr>
<tr>
<td>MWAIT—Monitor Wait.</td>
<td>4-158</td>
</tr>
<tr>
<td>NE—Two’s Complement Negation</td>
<td>4-161</td>
</tr>
<tr>
<td>NOP—No Operation</td>
<td>4-163</td>
</tr>
<tr>
<td>NOT—One’s Complement Negation</td>
<td>4-164</td>
</tr>
<tr>
<td>OR—Logical Inclusive OR</td>
<td>4-166</td>
</tr>
<tr>
<td>ORP—Bitwise Logical OR of Packed Double Precision Floating-Point Values</td>
<td>4-168</td>
</tr>
<tr>
<td>ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values</td>
<td>4-171</td>
</tr>
<tr>
<td>OUT—Output to Port.</td>
<td>4-174</td>
</tr>
<tr>
<td>OUTS/OUTSB/OUTSW/OUTSD—Output String to Port</td>
<td>4-176</td>
</tr>
<tr>
<td>PABS/PABSQ/PABSQV/PABSQX—Packed Absolute Value</td>
<td>4-180</td>
</tr>
<tr>
<td>PACK/PPACK/P PACK—Pack with Signed Saturation</td>
<td>4-186</td>
</tr>
<tr>
<td>PACKUS/PPACKUS/P PACKUS—Pack with Unsigned Saturation</td>
<td>4-194</td>
</tr>
<tr>
<td>PACKUSW/P PACKUSW/P PACKUSW—Pack with Unsigned Saturation</td>
<td>4-199</td>
</tr>
<tr>
<td>PADD/PADD2/P ADD/PADDQ—Add Packed Integers</td>
<td>4-204</td>
</tr>
<tr>
<td>PADD/PADD2/P ADD/PADDQ—Add Packed Signed Integers with Signed Saturation</td>
<td>4-211</td>
</tr>
<tr>
<td>PADDUS/PADDUSW/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation</td>
<td>4-215</td>
</tr>
<tr>
<td>PAIGNR—Packed Align Right</td>
<td>4-219</td>
</tr>
<tr>
<td>PAND—Logical AND</td>
<td>4-223</td>
</tr>
<tr>
<td>PANDN—Logical AND NOT.</td>
<td>4-226</td>
</tr>
<tr>
<td>PAUSE—Spin Loop Hint</td>
<td>4-229</td>
</tr>
<tr>
<td>PV/PAVG—Average Packed Integers</td>
<td>4-230</td>
</tr>
<tr>
<td>PBLEND/P BLEND—Variable Blend Packed Bytes</td>
<td>4-234</td>
</tr>
<tr>
<td>PBLENDW—Blend Packed Words</td>
<td>4-238</td>
</tr>
<tr>
<td>PCLMUL/PCLMULQ—Carry-Less Multiplication Quadword</td>
<td>4-241</td>
</tr>
<tr>
<td>PCMPEQ/P CMPEQW/P CMPEQW—Compare Packed Data for Equal</td>
<td>4-244</td>
</tr>
<tr>
<td>PCMPEQ/P CMPEQW/P CMPEQW—Compare Packed Data for Equal</td>
<td>4-250</td>
</tr>
<tr>
<td>PCMPESTRI—Packed Compare Explicit Length Strings, Return Index</td>
<td>4-253</td>
</tr>
<tr>
<td>PCMPESTRM—Packed Compare Explicit Length Strings, Return Mask</td>
<td>4-255</td>
</tr>
<tr>
<td>PCMPESTRI—Packed Compare Explicit Length Strings, Return Index</td>
<td>4-257</td>
</tr>
<tr>
<td>PCMPESTRI—Packed Compare Explicit Length Strings, Return Mask</td>
<td>4-263</td>
</tr>
<tr>
<td>PCMPESTRI—Packed Compare Implicit Length Strings, Return Index</td>
<td>4-266</td>
</tr>
<tr>
<td>PCMPESTRI—Packed Compare Implicit Length Strings, Return Mask</td>
<td>4-268</td>
</tr>
<tr>
<td>PDEP—Parallel Bits Deposit</td>
<td>4-270</td>
</tr>
<tr>
<td>PEXT—Parallel Bits Extract</td>
<td>4-272</td>
</tr>
<tr>
<td>PEXTB/P EXT/P EXTQ—Extract Byte/Dword/Qword</td>
<td>4-274</td>
</tr>
<tr>
<td>PEXTW—Extract Word</td>
<td>4-277</td>
</tr>
<tr>
<td>PHADD—Packed Horizontal Add</td>
<td>4-280</td>
</tr>
<tr>
<td>PHADDSW—Packed Horizontal Add and Saturate</td>
<td>4-284</td>
</tr>
<tr>
<td>PHSUB/SUBD—Packed Horizontal Subtract</td>
<td>4-286</td>
</tr>
<tr>
<td>PHSUB/SUBD—Packed Horizontal Subtract and Saturate</td>
<td>4-288</td>
</tr>
<tr>
<td>PIN/INS/INSR/PINSQ—Insert Byte/Dword/Qword</td>
<td>4-293</td>
</tr>
<tr>
<td>PINB/PINSQ/PINSRQ—Insert Word</td>
<td>4-296</td>
</tr>
<tr>
<td>PMADDUB/PMADDUB—Multiply and Add Packed Signed and Unsigned Bytes</td>
<td>4-298</td>
</tr>
<tr>
<td>PMADDUD—Multiply and Add Packed Integers</td>
<td>4-301</td>
</tr>
<tr>
<td>PMAXS/P MAXS/P MAXS/P MAXS—Maximum of Packed Signed Integers</td>
<td>4-304</td>
</tr>
<tr>
<td>PMAXUS/P MAXUS/P MAXUS/P MAXUS—Maximum of Packed Unsigned Integers</td>
<td>4-311</td>
</tr>
<tr>
<td>PMAXU/P MAXU/P MAXU/P MAXU—Maximum of Packed Unsigned Integers</td>
<td>4-316</td>
</tr>
<tr>
<td>PMINS/P MIN/P INS/P INS—Minimum of Packed Signed Integers</td>
<td>4-320</td>
</tr>
<tr>
<td>PMINS/P MIN/P INS/P INS—Minimum of Packed Signed Integers</td>
<td>4-325</td>
</tr>
<tr>
<td>PMIN/P MIN/P MIN/P MIN—Minimum of Packed Unsigned Integers</td>
<td>4-329</td>
</tr>
<tr>
<td>PMINS/P MIN/P INS/P INS—Minimum of Packed Unsigned Integers</td>
<td>4-334</td>
</tr>
<tr>
<td>PMOVMSK—Move Byte Mask</td>
<td>4-338</td>
</tr>
<tr>
<td>PMOVSS—Packed Move with Sign Extend</td>
<td>4-340</td>
</tr>
<tr>
<td>PMOVZX—Packed Move with Zero Extend</td>
<td>4-349</td>
</tr>
<tr>
<td>PMULDP—Multiply Packed Doubleword Integers</td>
<td>4-358</td>
</tr>
<tr>
<td>PMULRHS—Packed Multiply High with Round and Scale</td>
<td>4-361</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PMULHUw</td>
<td>Multiply Packed Unsigned Integers and Store High Result</td>
</tr>
<tr>
<td>PMULHw</td>
<td>Multiply Packed Signed Integers and Store High Result</td>
</tr>
<tr>
<td>PMULLD/PMULLQ</td>
<td>Multiply Packed Integers and Store Low Result</td>
</tr>
<tr>
<td>PMULLw</td>
<td>Multiply Packed Signed Integers and Store Low Result</td>
</tr>
<tr>
<td>PMULLUDQ</td>
<td>Multiply Packed Unsigned Doubleword Integers</td>
</tr>
<tr>
<td>POP</td>
<td>Pop a Value from the Stack</td>
</tr>
<tr>
<td>POPA/POPAD</td>
<td>Pop All General-Purpose Registers</td>
</tr>
<tr>
<td>POPCNT</td>
<td>Return the Count of Number of Bits Set to 1</td>
</tr>
<tr>
<td>POPFD/POPFD/POPFQ</td>
<td>Pop Stack into EFLAGS Register</td>
</tr>
<tr>
<td>POR</td>
<td>Bitwise Logical OR</td>
</tr>
<tr>
<td>PREFETCHh</td>
<td>Prefetch Data Into Caches</td>
</tr>
<tr>
<td>PREFETCHw</td>
<td>Prefetch Data into Caches in Anticipation of a Write</td>
</tr>
<tr>
<td>PSADBw</td>
<td>Compute Sum of Absolute Differences</td>
</tr>
<tr>
<td>PSHUFb</td>
<td>Pack Shuffle Bytes</td>
</tr>
<tr>
<td>PSHUDf</td>
<td>Shuffle Packed Doublewords</td>
</tr>
<tr>
<td>PSHUFIw</td>
<td>Shuffle Packed High Words</td>
</tr>
<tr>
<td>PSHUFLw</td>
<td>Shuffle Packed Low Words</td>
</tr>
<tr>
<td>PSHUFw</td>
<td>Shuffle Packed Words</td>
</tr>
<tr>
<td>PSIGNB/PSIGNW/PSIGND</td>
<td>Packed SIGN</td>
</tr>
<tr>
<td>PSLLQ</td>
<td>Shift Double Quadword Left Logical</td>
</tr>
<tr>
<td>PSLLLW/PSLLD/PSLLQ</td>
<td>Shift Packed Data Left Logical</td>
</tr>
<tr>
<td>PSRAW/PSRAD/PSRAQ</td>
<td>Shift Packed Data Right Arithmetic</td>
</tr>
<tr>
<td>PSRLQ</td>
<td>Shift Double Quadword Right Logical</td>
</tr>
<tr>
<td>PSRLW/PSRLD/PSRLQ</td>
<td>Shift Packed Data Right Logical</td>
</tr>
<tr>
<td>PSUBB/PSUBW/PSUBD</td>
<td>Subtract Packed Integers</td>
</tr>
<tr>
<td>PSUBQ</td>
<td>Subtract Packed Quadword Integers</td>
</tr>
<tr>
<td>PSUBSB/PSUBSW</td>
<td>Subtract Packed Signed Integers with Signed Saturation</td>
</tr>
<tr>
<td>PSUSBUS/PSUSBUSW</td>
<td>Subtract Packed Unsigned Integers with Unsigned Saturation</td>
</tr>
<tr>
<td>PTEST</td>
<td>Logical Compare</td>
</tr>
<tr>
<td>PWRITE</td>
<td>Write Data to a Processor Trace Packet</td>
</tr>
<tr>
<td>PUNPCKHbw/PUNPCKHw/PUNPCKHdq/PUNPCKHqd</td>
<td>Unpack High Data</td>
</tr>
<tr>
<td>PUNPCKlw/PUNPCKlwd/PUNPCKldq/PUNPCKldq</td>
<td>Unpack Low Data</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push Word, Doubleword or Quadword Onto the Stack</td>
</tr>
<tr>
<td>PUSHA/PUSHAD</td>
<td>Push All General-Purpose Registers</td>
</tr>
<tr>
<td>PUSHF/PUSHFD/PUSHFQ</td>
<td>Push EFLAGS Register onto the Stack</td>
</tr>
<tr>
<td>PXOR</td>
<td>Logical Exclusive OR</td>
</tr>
<tr>
<td>RCL/RCR/ROL/ROR</td>
<td>Rotate</td>
</tr>
<tr>
<td>RCPPS</td>
<td>Compute Reciprocals of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>RCPSS</td>
<td>Compute Reciprocal of Scalar Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>RDFSBASE/RDGSBASE</td>
<td>Read FS/GS Segment Base</td>
</tr>
<tr>
<td>RDSMR</td>
<td>Read from Model Specific Register</td>
</tr>
<tr>
<td>RDPID</td>
<td>Read Processor ID</td>
</tr>
<tr>
<td>RDPKRU</td>
<td>Read Protection Key Rights for User Pages</td>
</tr>
<tr>
<td>RDPMC</td>
<td>Read Performance-Monitoring Counters</td>
</tr>
<tr>
<td>RDRAND</td>
<td>Read Random Number</td>
</tr>
<tr>
<td>RDSEED</td>
<td>Read Random SEED</td>
</tr>
<tr>
<td>RDTSC</td>
<td>Read Time-Stamp Counter</td>
</tr>
<tr>
<td>RDTSCP</td>
<td>Read Time-Stamp Counter and Processor ID</td>
</tr>
<tr>
<td>REP/REPE/REPZ/REPNE/REPNZ</td>
<td>Repeat String Operation Prefix</td>
</tr>
<tr>
<td>RET</td>
<td>Return from Procedure</td>
</tr>
<tr>
<td>RORX</td>
<td>Rotate Right Logical Without Affecting Flags</td>
</tr>
<tr>
<td>ROUNDPD</td>
<td>Round Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>ROUNDPS</td>
<td>Round Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>ROUNDSD</td>
<td>Round Scalar Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>ROUNDSS</td>
<td>Round Scalar Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>RSM</td>
<td>Resume from System Management Mode</td>
</tr>
<tr>
<td>RSQRTPS</td>
<td>Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>RSQRTSS</td>
<td>Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value</td>
</tr>
<tr>
<td>SAHF</td>
<td>Store AH into Flags</td>
</tr>
</tbody>
</table>
PART II
FUNCTIONALITY

CHAPTER 5
INSTRUCTION SET REFERENCE, V-Z

5.1 TERNARY BIT VECTOR LOGIC TABLE

5.2 INSTRUCTIONS (V-Z)

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

VBLENDMDP/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

VBBROADCAST—Load with Broadcast Floating-Point Data
VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

VCVTTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

VCVTQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

VCVTQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values

VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Signed Doubleword Integer

VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer

VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers

VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

VCVTSS2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unigned Integer

VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

VCVTUDQPS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

VCVTUQ2PD—Convert Packed Unigned Quadword Integers to Packed Double-Precision Floating-Point Values

VCVTUQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

VCVTSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value

VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values

VFIXUPIMMPD—Fix Up Special Packed Double64 Values

VFIXUPIMMPS—Fix Up Special Packed Float32 Values

VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

VFMAADD128PD/VFMADD128PS/VFMADD2128PS—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

VFMAADD128PS/VFMADD2128PS/VFMADD2312PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

VFMAADD128PD/VFMADD2128PD/VFMADD2312PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

VFMAADD2128PD/VFMADD2312PD/VFMADD2312PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

VFMAADD2128PD/VFMADD2312PD/VFMADD2312PS/VFMADD2312PS—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

VFMAADD2128PD/VFMADD2312PD/VFMADD2312PS/VFMADD2312PS/VFMADD2312PS/VFMADD2312PS—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values
CHAPTER 6
SAFER MODE EXTENSIONS REFERENCE

6.1 OVERVIEW ................................................................. 6-1
6.2 SMX FUNCTIONALITY .................................................. 6-1
6.2.1 Detecting and Enabling SMX ................................. 6-1
6.2.2 SMX Instruction Summary ..................................... 6-2
6.2.2.1 GETSEC[CAPABILITIES] ............................... 6-2
6.2.2.2 GETSEC[ENTERACC] .................................. 6-2
6.2.2.3 GETSEC(EXITAC) ....................................... 6-3
6.2.2.4 GETSEC[SENTER] ....................................... 6-3
6.2.2.5 GETSEC[SEXIT] ........................................ 6-3
6.2.2.6 GETSEC[PARAMETERS] ............................... 6-4
6.2.2.7 GETSEC[SMCTRL] ..................................... 6-4
6.2.2.8 GETSEC[WAKEUP] ...................................... 6-4
6.2.3 Measured Environment and SMX ............................ 6-4
6.3 GETSEC LEAF FUNCTIONS .......................................... 6-5
APPENDIX A

OPCODE MAP

A.1 USING OPCODE TABLES ............................................................................................................ A-1
A.2 KEY TO ABBREVIATIONS ........................................................................................................ A-1
A.2.1 Codes for Addressing Method .......................................................................................... A-1
A.2.2 Codes for Operand Type ................................................................................................... A-2
A.2.3 Register Codes .................................................................................................................. A-3
A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes ............................ A-3
  A.2.4.1 One-Byte Opcode Instructions ................................................................................. A-3
  A.2.4.2 Two-Byte Opcode Instructions .............................................................................. A-4
  A.2.4.3 Three-Byte Opcode Instructions .......................................................................... A-5
  A.2.4.4 VEX Prefix Instructions ......................................................................................... A-5
A.2.5 Superscripts Utilized in Opcode Tables .............................................................................. A-6
A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS ................................................................... A-6
Table 5-20. VRSQRT14SS Special Cases.  
Table 5-21. VSCALEFPD/SD/PS/SS Special Cases.  
Table 5-22. Additional VSACLEFPD/SD Special Cases.  
Table 6-1. Layout of IA32_FEATURE_CONTROL.  
Table 6-2. GETSEC Leaf Functions.  
Table 6-3. Getsec Capability Result Encoding (EBX = 0).  
Table 6-4. Register State Initialization after GETSEC[ENTERACCS].  
Table 6-5. IA32_MISC_ENABLE MSR Initialization by ENTERACCS and SENTER.
| Table B-21. | Formats and Encodings of P6 Family Instructions | B-42 |
| Table B-22. | Formats and Encodings of SSE Floating-Point Instructions | B-43 |
| Table B-23. | Formats and Encodings of SSE Integer Instructions | B-48 |
| Table B-25. | Encoding of Granularity of Data Field (gg) | B-49 |
| Table B-24. | Format and Encoding of SSE Cacheability & Memory Ordering Instructions | B-49 |
| Table B-27. | Formats and Encodings of SSE2 Integer Instructions | B-55 |
| Table B-28. | Format and Encoding of SSE2 Cacheability Instructions | B-59 |
| Table B-29. | Formats and Encodings of SSE3 Floating-Point Instructions | B-60 |
| Table B-30. | Formats and Encodings for SSE3 Event Management Instructions | B-60 |
| Table B-31. | Formats and Encodings for SSE3 Integer and Move Instructions | B-60 |
| Table B-32. | Formats and Encodings for SSSE3 Instructions | B-61 |
| Table B-33. | Formats and Encodings of AESNI and PCLMULQDQ Instructions | B-64 |
| Table B-34. | Special Case Instructions Promoted Using REX.W | B-65 |
| Table B-35. | Encodings of SSE4.1 instructions | B-67 |
| Table B-36. | Encodings of SSE4.2 instructions | B-73 |
| Table B-37. | Encodings of AVX instructions | B-74 |
| Table B-38. | General Floating-Point Instruction Formats | B-114 |
| Table B-39. | Floating-Point Instruction Formats and Encodings | B-114 |
| Table B-40. | Encodings for VMX Instructions | B-118 |
| Table B-41. | Encodings for SMX Instructions | B-119 |
| Table C-1. | Simple Intrinsics | C-2 |
| Table C-2. | Composite Intrinsics | C-14 |
The *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569) are part of a set that describes the architecture and programming environment of all Intel 64 and IA-32 architecture processors. Other volumes in this set are:


The *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

### 1.1 Intel® 64 and IA-32 Processors Covered in This Manual

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series
ABOUT THIS MANUAL

- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- Intel® Xeon® processor E5-2600/1600 product families
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- 4th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2400/1400 v3 product families
- Intel® Xeon® processor E5-2600/1600 v3 product families
- Intel® Xeon® processor E3-1500m v5 product family
- Intel® Xeon® processor E5 v5 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- Intel® Xeon® Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family
- Intel® Xeon® Phi™ Processor 7215, 7285, 7295 Series

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.


The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Intel® microarchitecture code name Ivy Bridge-E and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Intel® microarchitecture code name Haswell-E and support Intel 64 architecture.

The Intel® Xeon® processor Z8000 series is based on the Intel microarchitecture code name Airmont.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Intel microarchitecture code name Silvermont.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Kaby Lake and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Intel® microarchitecture code name Knights Landing and supports Intel 64 architecture.
The 8th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Coffee Lake and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Intel® microarchitecture code name Knights Mill and supports Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

1.2 OVERVIEW OF VOLUME 2A, 2B, 2C AND 2D: INSTRUCTION SET REFERENCE

A description of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D content follows:

Chapter 1 — About This Manual. Gives an overview of all seven volumes of the Intel® 64 and IA-32 Architectures Software Developer's Manual. It also describes the notational conventions in these manuals and lists related Intel® manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — Instruction Format. Describes the machine-level instruction format used for all IA-32 instructions and gives the allowable encodings of prefixes, the operand-identifier byte (ModR/M byte), the addressing-mode specifier byte (SIB byte), and the displacement and immediate bytes.

Chapter 3 — Instruction Set Reference, A-L. Describes Intel 64 and IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3/MMX/SSE4 extensions, and system instructions are included.

Chapter 4 — Instruction Set Reference, M-U. Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It starts Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

Chapter 5 — Instruction Set Reference, V-Z. Continues the description of Intel 64 and IA-32 instructions started in chapters 3 and 4. It provides the balance of the alphabetized list of instructions and starts Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C.

Chapter 6 — Safer Mode Extensions Reference. Describes the safer mode extensions (SMX). SMX is intended for a system executive to support launching a measured environment in a platform where the identity of the software controlling the platform hardware can be measured for the purpose of making trust decisions. This chapter starts Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

Appendix A — Opcode Map. Gives an opcode map for the IA-32 instruction set.

Appendix B — Instruction Formats and Encodings. Gives the binary encoding of each form of each IA-32 instruction.

Appendix C — Intel® C/C++ Compiler Intrinsics and Functional Equivalents. Lists the Intel® C/C++ compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX and SSE/SSE2/SSE3 instructions.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.
### 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

**NOTE**

Avoid any software dependence upon the state of reserved bits in IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

### 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.
For example:

LOADREG: MOV EAX, SUBTOTAL

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The “B” designation is only used in situations where confusion as to the type of number might arise.

### 1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes in memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

```
CS:EIP
```

### 1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```

### 1.3.7 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.
1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

See also:
• The data sheet for a particular Intel 64 or IA-32 processor
• The specification update for a particular Intel 64 or IA-32 processor
• Intel® C++ Compiler documentation and online help: http://software.intel.com/en-us/articles/intel-compilers/
• Intel® Fortran Compiler documentation and online help: http://software.intel.com/en-us/articles/intel-compilers/
ABOUT THIS MANUAL

- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor: https://software.intel.com/sites/default/files/22/30/25602
- Performance Monitoring Unit Sharing Guide http://software.intel.com/file/30388

Literature related to selected features in future Intel processors are available at:

More relevant links are:
- Intel® Developer Zone: https://software.intel.com/en-us
- Processor support general link: http://www.intel.com/support/processors/
This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- Group 1
  - Lock and repeat prefixes:
    - LOCK prefix is encoded using F0H.
    - REPNE/REPNZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPNE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. F3H is also used as a mandatory prefix for POPCNT, LZCNT and ADOX instructions.

---

**Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format**

1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, “REX Prefixes” for additional information.
2. For VEX encoding information, see Section 2.3, “Intel® Advanced Vector Extensions (Intel® AVX)”.
3. Some rare instructions can take an 8B immediate or 8B displacement.
INSTRUCTION FORMAT

— BND prefix is encoded using F2H if the following conditions are true:
  • CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.
  • BNDCFGU.EN and/or IA32_BNDCFGS.EN is set.
  • When the F2 prefix precedes a near CALL, a near RET, a near JMP, a short Jcc, or a near Jcc instruction (see Chapter 17, “Intel® MPX,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

• Group 2
  — Segment override prefixes:
    • 2EH—CS segment override (use with any branch instruction is reserved).
    • 36H—SS segment override prefix (use with any branch instruction is reserved).
    • 3EH—DS segment override prefix (use with any branch instruction is reserved).
    • 26H—ES segment override prefix (use with any branch instruction is reserved).
    • 64H—FS segment override prefix (use with any branch instruction is reserved).
    • 65H—GS segment override prefix (use with any branch instruction is reserved).
  — Branch hints\(^1\):
    • 2EH—Branch not taken (used only with Jcc instructions).
    • 3EH—Branch taken (used only with Jcc instructions).

• Group 3
  • Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).

• Group 4
  • 67H—Address-size override prefix.

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See "LOCK—Assert LOCK# Signal Prefix" in Chapter 3, “Instruction Set Reference, A-L,” for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string and I/O instructions (MOV$S, CMOV, SCAS, LODS, STOS, INS, and OUTS). Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Some instructions may use F2H,F3H as a mandatory prefix to express distinct functionality.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (Jcc). Other use of branch hint prefixes and/or other undefined opcodes with Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality.

Other use of the 66H prefix is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

---

\(^1\) Some earlier microarchitectures used these as branch hints, but recent generations have not and they are reserved for future hint usage.
2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:
- An escape opcode byte 0FH as the primary opcode and a second opcode byte.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet).

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:
- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet).

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:
- The mod field combines with the r/m field to form 32 possible values: eight registers and 24 addressing modes.
- The reg/opcode field specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.
- The r/m field can specify a register as an operand or it can be combined with the mod field to encode an addressing mode. Sometimes, certain combinations of the mod field and the r/m field are used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:
- The scale field specifies the scale factor.
- The index field specifies the register number of the index register.
- The base field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.
2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled “REG =”). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute.

If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled “/digit (Opcode)”). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label “Value of ModR/M Byte (in Hexadecimal)”) contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4 and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1 and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

---

**Figure 2-2. Table Interpretation of ModR/M Byte (C8H)**

<table>
<thead>
<tr>
<th>Mod</th>
<th>RM</th>
<th>REG</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>001</td>
<td>C8H</td>
</tr>
</tbody>
</table>

---
Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BX+SI]</td>
<td>00 000</td>
<td>00 00 08 10 1B 20 2B 30 3B</td>
</tr>
<tr>
<td>[BX+DI]</td>
<td>001 01 09 11 19 21 29 31 39</td>
<td></td>
</tr>
<tr>
<td>[BP+SI]</td>
<td>010 02 0A 12 1A 22 2A 32 3A</td>
<td></td>
</tr>
<tr>
<td>[BP+DI]</td>
<td>011 03 0B 13 1B 23 2B 33 3B</td>
<td></td>
</tr>
<tr>
<td>[SI]</td>
<td>100 04 0C 14 1C 24 2C 34 3C</td>
<td></td>
</tr>
<tr>
<td>[DI]</td>
<td>101 05 0D 15 1D 25 2D 35 3D</td>
<td></td>
</tr>
<tr>
<td>disp16</td>
<td>110 06 0E 16 1E 26 2E 36 3E</td>
<td></td>
</tr>
<tr>
<td>[BX]</td>
<td>111 07 0F 17 1F 27 2F 37 3F</td>
<td></td>
</tr>
<tr>
<td>[BX+SI]1+disp8</td>
<td>01 000</td>
<td>00 40 4B 50 5B 60 6B 70 7B</td>
</tr>
<tr>
<td>[BX+DI]1+disp8</td>
<td>001 01 09 11 19 21 29 31 39</td>
<td></td>
</tr>
<tr>
<td>[BP+SI]1+disp8</td>
<td>010 02 0A 12 1A 22 2A 32 3A</td>
<td></td>
</tr>
<tr>
<td>[BP+DI]1+disp8</td>
<td>011 03 0B 13 1B 23 2B 33 3B</td>
<td></td>
</tr>
<tr>
<td>[SI]1+disp8</td>
<td>100 04 0C 14 1C 24 2C 34 3C</td>
<td></td>
</tr>
<tr>
<td>[DI]1+disp8</td>
<td>101 05 0D 15 1D 25 2D 35 3D</td>
<td></td>
</tr>
<tr>
<td>[BP]+disp8</td>
<td>110 06 0E 16 1E 26 2E 36 3E</td>
<td></td>
</tr>
<tr>
<td>[BX]+disp8</td>
<td>111 07 0F 17 1F 27 2F 37 3F</td>
<td></td>
</tr>
<tr>
<td>[BX]+disp16</td>
<td>10 000</td>
<td>00 80 8B 90 9B 40 4B 50 5B</td>
</tr>
<tr>
<td>[BX]+disp16</td>
<td>001 01 09 11 19 21 29 31 39</td>
<td></td>
</tr>
<tr>
<td>[BP]+disp16</td>
<td>010 02 0A 12 1A 22 2A 32 3A</td>
<td></td>
</tr>
<tr>
<td>[BP]+disp16</td>
<td>011 03 0B 13 1B 23 2B 33 3B</td>
<td></td>
</tr>
<tr>
<td>[SI]+disp16</td>
<td>100 04 0C 14 1C 24 2C 34 3C</td>
<td></td>
</tr>
<tr>
<td>[DI]+disp16</td>
<td>101 05 0D 15 1D 25 2D 35 3D</td>
<td></td>
</tr>
<tr>
<td>[BP]1+disp16</td>
<td>110 06 0E 16 1E 26 2E 36 3E</td>
<td></td>
</tr>
<tr>
<td>[BX]1+disp16</td>
<td>111 07 0F 17 1F 27 2F 37 3F</td>
<td></td>
</tr>
<tr>
<td>EAX/AX/AL/MM0/XMM0</td>
<td>11 000</td>
<td>00 C0 C8 D0 D8 E0 E8 F0 F8</td>
</tr>
<tr>
<td>ECX/CX/CL/MM1/XMM1</td>
<td>001 01 09 11 19 21 29 31 39</td>
<td></td>
</tr>
<tr>
<td>EDX/DX/DL/MM2/XMM2</td>
<td>010 02 0A 12 1A 22 2A 32 3A</td>
<td></td>
</tr>
<tr>
<td>EBX/BX/BL/MM3/XMM3</td>
<td>011 03 0B 13 1B 23 2B 33 3B</td>
<td></td>
</tr>
<tr>
<td>ESP/SP/AM0/MM4/XMM4</td>
<td>100 04 0C 14 1C 24 2C 34 3C</td>
<td></td>
</tr>
<tr>
<td>EBP/BP/CH/MM5/XMM5</td>
<td>101 05 0D 15 1D 25 2D 35 3D</td>
<td></td>
</tr>
<tr>
<td>ESI/SI/DH/MM6/XMM6</td>
<td>110 06 0E 16 1E 26 2E 36 3E</td>
<td></td>
</tr>
<tr>
<td>EDI/DI/BH/MM7/XMM7</td>
<td>111 07 0F 17 1F 27 2F 37 3F</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.
### NOTES:
1. The [-][-] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte’s base field. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4, and 5) and the scaling factor (determined by SIB byte bits 6 and 7).
Table 2-3. 32-Bit Addressing Forms with the SIB Byte

<table>
<thead>
<tr>
<th>Scaled Index</th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>00</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>ECX</td>
<td>001</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>EDX</td>
<td>010</td>
<td>02</td>
<td>010</td>
</tr>
<tr>
<td>EBX</td>
<td>011</td>
<td>03</td>
<td>011</td>
</tr>
<tr>
<td>ESP</td>
<td>010</td>
<td>04</td>
<td>010</td>
</tr>
<tr>
<td>[*]</td>
<td>011</td>
<td>05</td>
<td>011</td>
</tr>
<tr>
<td>ESI</td>
<td>100</td>
<td>06</td>
<td>100</td>
</tr>
<tr>
<td>EDI</td>
<td>101</td>
<td>07</td>
<td>101</td>
</tr>
<tr>
<td>EAX*2</td>
<td>00</td>
<td>08</td>
<td>000</td>
</tr>
<tr>
<td>ECX*2</td>
<td>001</td>
<td>09</td>
<td>001</td>
</tr>
<tr>
<td>EDX*2</td>
<td>010</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>EBX*2</td>
<td>011</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>none</td>
<td>100</td>
<td>12</td>
<td>100</td>
</tr>
<tr>
<td>EBP</td>
<td>101</td>
<td>13</td>
<td>101</td>
</tr>
<tr>
<td>ESP*2</td>
<td>110</td>
<td>14</td>
<td>110</td>
</tr>
<tr>
<td>ESI*2</td>
<td>111</td>
<td>15</td>
<td>111</td>
</tr>
<tr>
<td>EDI*2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX*4</td>
<td>10</td>
<td>16</td>
<td>100</td>
</tr>
<tr>
<td>ECX*4</td>
<td>101</td>
<td>17</td>
<td>101</td>
</tr>
<tr>
<td>EDX*4</td>
<td>110</td>
<td>18</td>
<td>110</td>
</tr>
<tr>
<td>EBX*4</td>
<td>111</td>
<td>19</td>
<td>111</td>
</tr>
<tr>
<td>none</td>
<td>100</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>EBP*4</td>
<td>101</td>
<td>21</td>
<td>101</td>
</tr>
<tr>
<td>ESP*4</td>
<td>110</td>
<td>22</td>
<td>110</td>
</tr>
<tr>
<td>ESI*4</td>
<td>111</td>
<td>23</td>
<td>111</td>
</tr>
<tr>
<td>EDI*4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX*8</td>
<td>11</td>
<td>24</td>
<td>110</td>
</tr>
<tr>
<td>ECX*8</td>
<td>111</td>
<td>25</td>
<td>111</td>
</tr>
<tr>
<td>EDX*8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBX*8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>100</td>
<td>26</td>
<td>100</td>
</tr>
<tr>
<td>EBP*8</td>
<td>101</td>
<td>27</td>
<td>101</td>
</tr>
<tr>
<td>ESP*8</td>
<td>110</td>
<td>28</td>
<td>110</td>
</tr>
<tr>
<td>ESI*8</td>
<td>111</td>
<td>29</td>
<td>111</td>
</tr>
<tr>
<td>EDI*8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:
   - MOD bits               | Effective Address
       00                  | [scaled index] + disp32
       01                  | [scaled index] + disp8 + [EBP]
       10                  | [scaled index] + disp32 + [EBP]

2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:
- **Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- **64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.
2.2.1 REX Prefixes

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.
- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored. Only one REX prefix is allowed per instruction. If used, the REX prefix byte must immediately precede the opcode byte or the escape opcode byte (0FH). When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can be immediately preceding the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

<table>
<thead>
<tr>
<th>Legacy Prefixes</th>
<th>REX Prefix</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grp 1, Grp 2, Grp 3, Grp 4 (optional)</td>
<td>(optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

Figure 2-3. Prefix Ordering in 64-bit Mode

2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 through Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.
- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

Table 2-4. REX Prefix Fields [BITS: 0100WRXB]

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Position</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>7:4</td>
<td>0100</td>
</tr>
<tr>
<td>W</td>
<td>3</td>
<td>0 = Operand size determined by CS.D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = 64 Bit Operand Size</td>
</tr>
<tr>
<td>R</td>
<td>2</td>
<td>Extension of the ModR/M reg field</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>Extension of the SIB index field</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Extension of the ModR/M r/m field, SIB base field, or Opcode reg field</td>
</tr>
</tbody>
</table>

Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used
In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte’s reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations. Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.
2.2.1.3  Displacement
Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

2.2.1.4  Direct Memory-Offset MOVs
In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>MOV AL, moffset</td>
</tr>
<tr>
<td>A1</td>
<td>MOV EAX, moffset</td>
</tr>
<tr>
<td>A2</td>
<td>MOV moffset, AL</td>
</tr>
<tr>
<td>A3</td>
<td>MOV moffset, EAX</td>
</tr>
</tbody>
</table>

2.2.1.5  Immediates
In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.
Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.
For example:

48 B8 887765544332211 MOV RAX,112234455667788H
### 2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

#### Table 2-7. RIP-Relative Addressing

<table>
<thead>
<tr>
<th>ModR/M and SIB Sub-field Encodings</th>
<th>Compatibility Mode Operation</th>
<th>64-bit Mode Operation</th>
<th>Additional Implications in 64-bit mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModR/M Byte</td>
<td>disp32</td>
<td>RIP + disp32</td>
<td>Must use SIB form with normal (zero-based) displacement addressing</td>
</tr>
<tr>
<td>r/m = 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r/m = 101 (one)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIB Byte</td>
<td>base = 101 (none)</td>
<td>if mod = 00, disp32</td>
<td>Same as legacy</td>
</tr>
<tr>
<td>index = 100 (none)</td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>scale = 0, 1, 2, 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00R still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

### 2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

### 2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8-DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).
2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel AVX instructions are encoded using an encoding scheme that combines prefix bytes, opcode extension field, operand encoding fields, and vector length encoding capability into a new prefix, referred to as VEX. In the VEX encoding scheme, the VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the two-byte or three-byte length of the VEX prefix, the VEX encoding format provides a more compact representation/packing of the components of encoding an instruction in Intel 64 architecture. The VEX encoding scheme also allows more headroom for future growth of Intel 64 architecture.

2.3.1 Instruction Format

Instruction encoding using VEX prefix provides several advantages:

• Instruction syntax support for three operands and up-to four operands when necessary. For example, the third source register used by VBLENDVPD is encoded using bits 7:4 of the immediate byte.
• Encoding support for vector length of 128 bits (using XMM registers) and 256 bits (using YMM registers).
• Encoding support for instruction syntax of non-destructive source operands.
• Elimination of escape opcode byte (0FH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
• Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
• Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
• Extensibility for future instruction extensions without significant instruction length increase.

Figure 2-8 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, "single-purpose" byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

![Figure 2-8. Instruction Encoding Format with VEX Prefix](image)

2.3.2 VEX and the LOCK prefix

Any VEX-encoded instruction with a LOCK prefix preceding VEX will #UD.

2.3.3 VEX and the 66H, F2H, and F3H prefixes

Any VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

2.3.4 VEX and the REX prefix

Any VEX-encoded instruction with a REX prefix proceeding VEX will #UD.
2.3.5 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- **Non-destructive source register encoding (applicable to three and four operand syntax):** This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1’s complement form (inverted form), i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.

- **Vector length encoding:** This 1-bit field represented by the notation VEX.L. L= 0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.

- **REX prefix functionality:** Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1’s complement form, i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
  
  — Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1’s complement encoding. This is represented as VEX.R.
  
  — Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1’s complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
  
  — Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.

- **Compaction of SIMD prefix:** Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.

- **Compaction of two-byte and three-byte opcode:** More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.
The following subsections describe the various fields in two or three-byte VEX prefix.

### 2.3.5.1 VEX Byte 0, bits[7:0]

VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

### 2.3.5.2 VEX Byte 1, bit [7] - 'R'

VEX Byte 1, bit [7] contains a bit analogous to a bit inverted REX.R. In protected and compatibility modes the bit must be set to ‘1’ otherwise the instruction is LES or LDS.
INSTRUCTION FORMAT

This bit is present in both 2- and 3-byte VEX prefixes.
The usage of WTXB bits for legacy instructions is explained in detail section 2.2.1.2 of Intel 64 and IA-32 Architectures Software developer's manual, Volume 2A.
This bit is stored in bit inverted format.

2.3.5.3 3-byte VEX byte 1, bit[6] - 'X'
Bit[6] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.X. It is an extension of the SIB Index field in 64-bit modes. In 32-bit modes, this bit must be set to '1' otherwise the instruction is LES or LDS. This bit is available only in the 3-byte VEX prefix.
This bit is stored in bit inverted format.

2.3.5.4 3-byte VEX byte 1, bit[5] - 'B'
Bit[5] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.B. In 64-bit modes, it is an extension of the ModR/M r/m field, or the SIB base field. In 32-bit modes, this bit is ignored. This bit is available only in the 3-byte VEX prefix.
This bit is stored in bit inverted format.

2.3.5.5 3-byte VEX byte 2, bit[7] - 'W'
Bit[7] of the 3-byte VEX byte 2 is represented by the notation VEX.W. It can provide following functions, depending on the specific opcode.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes for these instructions, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don’t care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3] - 'vvvv' the Source or Dest Register Specifier
In 32-bit mode the VEX first byte C4 and C5 alias onto the LES and LDS instructions. To maintain compatibility with existing programs the VEX 2nd byte, bits [7:6] must be 11b. To achieve this, the VEX payload bits are selected to place only inverted, 64-bit valid fields (extended register selectors) in these upper bits.
The 2-byte VEX Byte 1, bits [6:3] and the 3-byte VEX, Byte 2, bits [6:3] encode a field (shorthand VEX.vvvv) that for instructions with 2 or more source registers and an XMM or YMM or memory destination encodes the first source register specifier stored in inverted (1's complement) form.
VEX.vvvv is not used by the instructions with one source (except certain shifts, see below) or on instructions with no XMM or YMM or memory destination. If an instruction does not use VEX.vvvv then it should be set to 1111b otherwise instruction will #UD.
In 64-bit mode all 4 bits may be used. See Table 2-8 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).
Table 2-8. VEX.vvvv to register name mapping

<table>
<thead>
<tr>
<th>VEX.vvvv</th>
<th>Dest Register</th>
<th>Valid in Legacy/Compatibility 32-bit modes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111B</td>
<td>XMM0/YMM0</td>
<td>Valid</td>
</tr>
<tr>
<td>1110B</td>
<td>XMM1/YMM1</td>
<td>Valid</td>
</tr>
<tr>
<td>1101B</td>
<td>XMM2/YMM2</td>
<td>Valid</td>
</tr>
<tr>
<td>1100B</td>
<td>XMM3/YMM3</td>
<td>Valid</td>
</tr>
<tr>
<td>1011B</td>
<td>XMM4/YMM4</td>
<td>Valid</td>
</tr>
<tr>
<td>1010B</td>
<td>XMM5/YMM5</td>
<td>Valid</td>
</tr>
<tr>
<td>1001B</td>
<td>XMM6/YMM6</td>
<td>Valid</td>
</tr>
<tr>
<td>1000B</td>
<td>XMM7/YMM7</td>
<td>Valid</td>
</tr>
<tr>
<td>0111B</td>
<td>XMM8/YMM8</td>
<td>Invalid</td>
</tr>
<tr>
<td>0110B</td>
<td>XMM9/YMM9</td>
<td>Invalid</td>
</tr>
<tr>
<td>0101B</td>
<td>XMM10/YMM10</td>
<td>Invalid</td>
</tr>
<tr>
<td>0100B</td>
<td>XMM11/YMM11</td>
<td>Invalid</td>
</tr>
<tr>
<td>0011B</td>
<td>XMM12/YMM12</td>
<td>Invalid</td>
</tr>
<tr>
<td>0010B</td>
<td>XMM13/YMM13</td>
<td>Invalid</td>
</tr>
<tr>
<td>0001B</td>
<td>XMM14/YMM14</td>
<td>Invalid</td>
</tr>
<tr>
<td>0000B</td>
<td>XMM15/YMM15</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

The VEX.vvvv field is encoded in bit inverted format for accessing a register operand.

2.3.6 Instruction Operand Encoding and VEX.vvvv, ModR/M

VEX-encoded instructions support three-operand and four-operand instruction syntax. Some VEX-encoded instructions have syntax with less than three operands, e.g. VEX-encoded pack shift instructions support one source operand and one destination operand.

The roles of VEX.vvvv, reg field of ModR/M byte (ModR/M.reg), r/m field of ModR/M byte (ModR/M.r/m) with respect to encoding destination and source operands vary with different type of instruction syntax.

The role of VEX.vvvv can be summarized to three situations:

- VEX.vvvv encodes the first source register operand, specified in inverted (1’s complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1’s complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 2-9. The notation in the “Opcode” column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 2-9. Instructions with a VEX.vvvv destination

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDD.128.66.0F 73 /7 ib</td>
<td>VPSLLDQ xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 73 /3 ib</td>
<td>VPSRLDQ xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 71 /2 ib</td>
<td>VPSRLW xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 72 /2 ib</td>
<td>VPSRLD xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 73 /2 ib</td>
<td>VPSRLQ xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 71 /4 ib</td>
<td>VPSRAW xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 72 /4 ib</td>
<td>VPSRAD xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 71 /6 ib</td>
<td>VPSLLW xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 72 /6 ib</td>
<td>VPSLLD xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F 73 /6 ib</td>
<td>VPSLLQ xmm1, xmm2, imm8</td>
</tr>
</tbody>
</table>
The role of ModR/M.r/m field can be summarized to two situations:

- ModR/M.r/m encodes the instruction operand that references a memory address.
- For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:

- ModR/M.reg encodes either the destination register operand or a source register operand.
- For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves the following situation:

- Imm8[7:4] encodes the third source register operand.

2.3.6.1 3-byte VEX byte 1, bits[4:0] - “m-mmmm”

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

<table>
<thead>
<tr>
<th>VEX.m-mmmm</th>
<th>Implied Leading Opcode Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000B</td>
<td>Reserved</td>
</tr>
<tr>
<td>00001B</td>
<td>0F</td>
</tr>
<tr>
<td>00010B</td>
<td>0F 38</td>
</tr>
<tr>
<td>00011B</td>
<td>0F 3A</td>
</tr>
<tr>
<td>00100-1111B</td>
<td>Reserved</td>
</tr>
<tr>
<td>(2-byte VEX)</td>
<td>0F</td>
</tr>
</tbody>
</table>

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- “L”

The vector length field, VEX.L, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If “VEX.L = 1”, it indicates 256-bit vector operation. “VEX.L = 0” indicates scalar and 128-bit vector operations.

The instruction VZEROUPPER is a special case that is encoded with VEX.L = 0, although its operation zero’s bits 255:128 of all YMM registers accessible in the current operating mode.

See the following table.

<table>
<thead>
<tr>
<th>VEX.L</th>
<th>Vector Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128-bit (or 32/64-bit scalar)</td>
</tr>
<tr>
<td>1</td>
<td>256-bit</td>
</tr>
</tbody>
</table>

2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- “pp”

Up to one implied prefix is encoded by bits[1:0] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes.

See the following table.
2.3.7 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

2.3.8 The MODRM, SIB, and Displacement Bytes

The encodings are unchanged but the interpretation of reg_field or rm_field differs (see above).

2.3.9 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

2.3.10 AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper bits (above bit 128) of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper bits.

2.3.10.1 Vector Length Transition and Programming Considerations

An instruction encoded with a VEX.128 prefix that loads a YMM register operand operates as follows:

- Data is loaded into bits 127:0 of the register
- Bits above bit 127 in the register are cleared.

Thus, such an instruction clears bits 255:128 of a destination YMM register on processors with a maximum vector-register width of 256 bits. In the event that future processors extend the vector registers to greater widths, an instruction encoded with a VEX.128 or VEX.256 prefix will also clear any bits beyond bit 255. (This is in contrast with legacy SSE instructions, which have no VEX prefix; these modify only bits 127:0 of any destination register operand.)

Programmers should bear in mind that instructions encoded with VEX.128 and VEX.256 prefixes will clear any future extensions to the vector registers. A calling function that uses such extensions should save their state before calling legacy functions. This is not possible for involuntary calls (e.g., into an interrupt-service routine). It is recommended that software handling involuntary calls accommodate this by not executing instructions encoded with VEX.128 and VEX.256 prefixes. In the event that it is not possible or desirable to restrict these instructions, then software must take special care to avoid actions that would, on future processors, zero the upper bits of vector registers.

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true

<table>
<thead>
<tr>
<th>pp</th>
<th>Implies this prefix after other prefixes but before VEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>00B</td>
<td>None</td>
</tr>
<tr>
<td>01B</td>
<td>66</td>
</tr>
<tr>
<td>10B</td>
<td>F3</td>
</tr>
<tr>
<td>11B</td>
<td>F2</td>
</tr>
</tbody>
</table>

Table 2-12. VEX.pp interpretation
if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

### 2.3.11 AVX Instruction Length

The AVX instructions described in this document (including VEX and ignoring other prefixes) do not exceed 11 bytes in length, but may increase in the future. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes.

### 2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® AVX2), an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of Intel AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-3 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte’s base field. The register names also include R8L-R15L applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-3). In 32-bit mode, R8L-R15L does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The leftmost column includes vector registers VR8-VR15 (i.e. XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.

<table>
<thead>
<tr>
<th>Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>(In decimal) Base =</td>
</tr>
<tr>
<td>(In binary) Base =</td>
</tr>
<tr>
<td>Scaled Index</td>
</tr>
<tr>
<td>VR0/VR8</td>
</tr>
<tr>
<td>VR1/VR9</td>
</tr>
<tr>
<td>VR2/VR10</td>
</tr>
<tr>
<td>VR3/VR11</td>
</tr>
<tr>
<td>VR4/VR12</td>
</tr>
<tr>
<td>VR5/VR13</td>
</tr>
<tr>
<td>VR6/VR14</td>
</tr>
<tr>
<td>VR7/VR15</td>
</tr>
</tbody>
</table>

*1 00 000 00 01 02 03 04 05 06 07
*2 01 000 40 41 42 43 44 45 46 47

2.3.12.1 64-bit Mode VSIB Memory Addressing

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose register as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 2-13 base register should be interpreted as the full 64-bit of each register.

### Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte (Contd.)

<table>
<thead>
<tr>
<th>VR0/VR8</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR1/VR9</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>VR2/VR10</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>VR3/VR11</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>VR4/VR12</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>VR5/VR13</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>VR6/VR14</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>VR7/VR15</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

**NOTES:**
1. If ModR/M.mod = 00b, the base address is zero, then effective address is computed as [scaled vector index] + disp32. Otherwise the base address is computed as [EBP/R13] + disp, the displacement is either 8 bit or 32 bit depending on the value of ModR/M.mod:

<table>
<thead>
<tr>
<th>MOD</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>[Scaled Vector Register] + Disp32</td>
</tr>
<tr>
<td>01b</td>
<td>[Scaled Vector Register] + Disp8 + [EBP/R13]</td>
</tr>
<tr>
<td>10b</td>
<td>[Scaled Vector Register] + Disp32 + [EBP/R13]</td>
</tr>
</tbody>
</table>

2.4 AVX AND SSE INSTRUCTION EXCEPTION SPECIFICATION

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-14 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.4.1 through 2.5.1. For example, ADDPS contains the entry:

"See Exceptions Type 2"

In this entry, "Type2" can be looked up in Table 2-14.

The instruction’s corresponding CPUID feature flag can be identified in the fourth column of the Instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

**NOTE**

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
### Table 2-14. Exception class description

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction set</th>
<th>Mem arg</th>
<th>Floating-Point Exceptions (#XM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>AVX, Legacy SSE</td>
<td>16/32 byte explicitly aligned</td>
<td>None</td>
</tr>
<tr>
<td>Type 2</td>
<td>AVX, Legacy SSE</td>
<td>16/32 byte not explicitly aligned</td>
<td>Yes</td>
</tr>
<tr>
<td>Type 3</td>
<td>AVX, Legacy SSE</td>
<td>&lt; 16 byte</td>
<td>Yes</td>
</tr>
<tr>
<td>Type 4</td>
<td>AVX, Legacy SSE</td>
<td>16/32 byte not explicitly aligned</td>
<td>No</td>
</tr>
<tr>
<td>Type 5</td>
<td>AVX, Legacy SSE</td>
<td>&lt; 16 byte</td>
<td>No</td>
</tr>
<tr>
<td>Type 6</td>
<td>AVX (no Legacy SSE)</td>
<td>Varies</td>
<td>(At present, none do)</td>
</tr>
<tr>
<td>Type 7</td>
<td>AVX, Legacy SSE</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Type 8</td>
<td>AVX</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Type 11</td>
<td>F16C</td>
<td>8 or 16 byte, Not explicitly aligned, no AC#</td>
<td>Yes</td>
</tr>
<tr>
<td>Type 12</td>
<td>AVX2</td>
<td>Not explicitly aligned, no AC#</td>
<td>No</td>
</tr>
</tbody>
</table>

See Table 2-15 for lists of instructions in each exception class.
INSTRUCTION FORMAT

Table 2-15. Instructions in each Exception Class
Exception Class

Instruction

Type 1

(V)MOVAPD, (V)MOVAPS, (V)MOVDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS

Type 2

(V)ADDPD, (V)ADDPS, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ,
(V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*,
VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS,
VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUB132PS, VFMADDSUB213PS,
VFMADDSUB231PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS,
VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS,
VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS,
VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS,
VFNMSUB213PS, VFNMSUB231PS, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS,
(V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS

Type 3

(V)ADDSD, (V)ADDSS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS,
(V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTTSD2SI, (V)CVTTSS2SI, (V)DIVSD, (V)DIVSS,
VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADD132SS, VFMADD213SS, VFMADD231SS,
VFMSUB132SD, VFMSUB213SD, VFMSUB231SD, VFMSUB132SS, VFMSUB213SS, VFMSUB231SS,
VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS,
VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS,
(V)MAXSD, (V)MAXSS, (V)MINSD, (V)MINSS, (V)MULSD, (V)MULSS, (V)ROUNDSD, (V)ROUNDSS, (V)SQRTSD,
(V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMISD, (V)UCOMISS

Type 4

(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD,
(V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, VBLENDVPD, VBLENDVPS, (V)LDDQU***,
(V)MASKMOVDQU, (V)PTEST, VTESTPS, VTESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*,
(V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW,
(V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW,
(V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB,
(V)PBLENDW, (V)PCMP(E/I)STRI/M***, (V)PCMPEQB, (V)PCMPEQW, (V)PCMPEQD, (V)PCMPEQQ, (V)PCMPGTB,
(V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADDD, (V)PHADDSW,
(V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, (V)PMADDWD, (V)PMADDUBSW, (V)PMAXSB,
(V)PMAXSW, (V)PMAXSD, (V)PMAXUB, (V)PMAXUW, (V)PMAXUD, (V)PMINSB, (V)PMINSW, (V)PMINSD,
(V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRSW, (V)PMULHW, (V)PMULLW, (V)PMULLD,
(V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUFB, (V)PSHUFD, (V)PSHUFHW, (V)PSHUFLW, (V)PSIGNB,
(V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ,
(V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PUNPCKHBW, (V)PUNPCKHWD,
(V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ,
(V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD,
(V)UNPCKLPS, (V)XORPD, (V)XORPS, VPBLENDD, VPERMD, VPERMPS, VPERMPD, VPERMQ, VPSLLVD, VPSLLVQ,
VPSRAVD, VPSRLVD, VPSRLVQ, VPERMILPD, VPERMILPS, VPERM2F128

Type 5

(V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPS,
(V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB,
(V)PINSRD, (V)PINSRW, (V)PINSRQ, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX, VLDMXCSR*, VSTMXCSR

Type 6

VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS**,
VMASKMOVPD**, VPMASKMOVD, VPMASKMOVQ, VBROADCASTI128, VPBROADCASTB, VPBROADCASTD,
VPBROADCASTW, VPBROADCASTQ, VEXTRACTI128, VINSERTI128, VPERM2I128

Type 7

(V)MOVLHPS, (V)MOVHLPS, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMSKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW,
(V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ

Type 8

VZEROALL, VZEROUPPER

Type 11

VCVTPH2PS, VCVTPS2PH

Type 12

VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD,
VPGATHERQQ

(*) - Additional exception restrictions are present - see the Instruction description for details
(**) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e. no
alignment checks are performed.

Vol. 2A 2-23


INSTRUCTION FORMAT

(*** ) - PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-17 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

### Table 2-16. #UD Exception and VEX.W=1 Encoding

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>#UD If VEX.W = 1 in all modes</th>
<th>#UD If VEX.W = 1 in non-64-bit modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>VBLENDVPD, VBLENDVPS, VPBLENDB, VTESTPD, VTESTPS, VPBLENDD, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMILPD, VPERMILPS, VPERM2F128</td>
<td></td>
</tr>
<tr>
<td>Type 2</td>
<td>VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSETTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSETTI128</td>
<td></td>
</tr>
<tr>
<td>Type 3</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 4</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 5</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 6</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 7</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 8</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 11</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 12</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Exception Class</td>
<td>#UD If VEX.L = 0</td>
<td>#UD If (VEX.L = 1 &amp;&amp; AVX2 not present &amp;&amp; AVX present)</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>Type 1</td>
<td>VMOVNTDQA</td>
<td></td>
</tr>
<tr>
<td>Type 2</td>
<td>VDPPD</td>
<td></td>
</tr>
<tr>
<td>Type 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 5</td>
<td>VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPS, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTREB, VPEXTR, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVPSX/ZX, VLDMQ, VSTMCSR</td>
<td>Same as column 3</td>
</tr>
<tr>
<td>Type 6</td>
<td>VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128</td>
<td></td>
</tr>
<tr>
<td>Type 7</td>
<td>VMOVLHPS, VMOVLHPS, VMOVMSKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ</td>
<td>VMOVLHPS, VMOVLHPS</td>
</tr>
<tr>
<td>Type 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.4.1 Exceptions Type 1 (Aligned memory reference)

**Table 2-18. Type 1 Class Exception Conditions**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invalid Opcode, #UD</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td><strong>Device Not Available, #NM</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td><strong>Stack, SS(0)</strong></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td><strong>General Protection, #GP(0)</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>VEX.256: Memory operand is not 32-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VEX.128: Memory operand is not 16-byte aligned.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>Legacy SSE: Memory operand is not 16-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td><strong>Page Fault #PF(fault-code)</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>
### 2.4.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned)

#### Table 2-19. Type 2 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Legacy SSE: Memory operand is not 16-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
## 2.4.3 Exceptions Type 3 (<16 Byte memory argument)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For a memory address referencing the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 Bytes or less is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
### 2.4.4 Exceptions Type 4 (>=16 Byte mem arg no alignment, no floating-point exceptions)

#### Table 2-21. Type 4 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X</td>
<td>X</td>
<td></td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.E[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X X</td>
<td>X</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X X</td>
<td>X</td>
<td></td>
<td>Legacy SSE: Memory operand is not 16-byte aligned.¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X X</td>
<td>X</td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>

#### NOTES:
1. PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.
## 2.4.5 Exceptions Type 5 (<16 Byte mem arg and no FP exceptions)

### Table 2-22. Type 5 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
2.4.6 Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues)

Note: At present, the AVX instructions in this category do not generate floating-point exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
### 2.4.7 Exceptions Type 7 (No FP exceptions, no memory arg)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ ‘11b’.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
</tbody>
</table>

### 2.4.8 Exceptions Type 8 (AVX and no memory argument)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Always in Real or Virtual-8086 mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ ‘11b’.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If proceeded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CPUID.01H.ECX.AVX[bit 28]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
</tbody>
</table>
### Exception Type 11 (VEX-only, mem arg no AC, floating-point exceptions)

Table 2-26. Type 11 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3] = 1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS seg-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF (fault-code)</td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>SIMD Floating-Point Exception, #XM</td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCEPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
### Exception Type 12 (VEX-only, VSIB mem arg, no AC, no floating-point exceptions)

#### Table 2-27. Type 12 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>NA</td>
<td>X</td>
<td>If address size attribute is 16 bit.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If ModR/M.mod = ‘11b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If ModR/M.rm ≠ ‘100b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any vector register is used more than once between the destination register, mask register and the index register in VSIB addressing.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF (fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>

### 2.5 VEX Encoding Support for GPR Instructions

VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (OFH), two-byte escape via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access or memory addressing.
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only.
- VEX-encoded GPR instructions are encoded with VEX.L=0.
Any VEX-encoded GPR instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.
Any VEX-encoded GPR instruction with a REX prefix proceeding VEX will #UD.
VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

2.5.1 Exception Conditions for VEX-Encoded GPR Instructions
The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GRP instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-29</td>
<td>ANDN, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX</td>
</tr>
</tbody>
</table>

(*) - Additional exception restrictions are present - see the Instruction description for details.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-806</th>
<th>Protected and Compatibility 64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If BMI1/BMI2 CPUID feature flag is '0'.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If a VEX prefix is present.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>

2.6 INTEL® AVX-512 ENCODING
The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.
The significant feature differences between EVEX and VEX are summarized below.
INSTRUCTION FORMAT

- EVEX is a 4-Byte prefix (the first byte must be 62H); VEX is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- EVEX prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- EVEX prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- EVEX memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve the encoding density of the instruction byte stream.
- EVEX prefix can encode functionality that are specific to instruction classes (e.g., packed instruction with “load+op” semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support “suppress all exceptions” functionality).

2.6.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10. Note that the values contained within brackets are optional.

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).
The bit fields in P[23:0] are divided into the following functional groups (Table 2-30 provides a tabular summary):

- **Reserved bits:** P[3:2] must be 0, otherwise #UD.
- **Fixed-value bit:** P[10] must be 1, otherwise #UD.
- **Compressed legacy prefix/escape bytes:** P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- **Operand specifier modifier bits for vector register, general purpose register, memory addressing:** P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- **Operand specifier modifier bit for vector register:** P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with EVEX.R and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- **Non-destructive source/vector index operand specifier:** P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- **Op-mask register specifiers:** P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- **EVEX.W:** P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- **Vector destination merging/zeroing:** P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- **Broadcast/Static-rounding/SAE context bit:** P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:
  - Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
  - Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).
  - Enable SAE for floating-point instructions with arithmetic semantic that is not rounding.
  - For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.

### Table 2-30. EVEX Prefix Bit Field Functional Grouping

<table>
<thead>
<tr>
<th>Notation</th>
<th>Bit field Group</th>
<th>Position</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>Reserved</td>
<td>P[3:2]</td>
<td>Must be 0.</td>
</tr>
<tr>
<td>EVEX.mm</td>
<td>Compressed legacy escape</td>
<td>P[1:0]</td>
<td>Identical to lowest 2 bits of VEX.mmmmm.</td>
</tr>
<tr>
<td>EVEX.RXB</td>
<td>Next-8 register specifier modifier</td>
<td>P[7:5]</td>
<td>Combine with ModR/M.reg, ModR/M.rm (base, index/vidx).</td>
</tr>
<tr>
<td>EVEX.vvvv</td>
<td>NDS register specifier</td>
<td>P[14:11]</td>
<td>Same as VEX.vvvv.</td>
</tr>
<tr>
<td>EVEXV'</td>
<td>High-16 NDS/VIDX register specifier</td>
<td>P[19]</td>
<td>Combine with EVEX.vvvv or when VSIB present.</td>
</tr>
<tr>
<td>EVEX.aaa</td>
<td>Embedded opmask register specifier</td>
<td>P[18:16]</td>
<td></td>
</tr>
<tr>
<td>EVEX.W</td>
<td>Osize promotion/Opcode extension</td>
<td>P[15]</td>
<td></td>
</tr>
<tr>
<td>EVEX.z</td>
<td>Zeroing/Merging</td>
<td>P[23]</td>
<td></td>
</tr>
<tr>
<td>EVEX.b</td>
<td>Broadcast/RC/SAE Context</td>
<td>P[20]</td>
<td></td>
</tr>
<tr>
<td>EVEX.L'L</td>
<td>Vector length/RC</td>
<td>P[22:21]</td>
<td></td>
</tr>
</tbody>
</table>
• Vector length/rounding control specifier: P[22:21] can serve one of three options.
  — Vector length information for packed vector instructions.
  — Ignored for instructions operating on vector register content as a single data element.
  — Rounding control for floating-point instructions that have a rounding semantic and whose source and
    destination operands are all vector registers.

2.6.2 Register Specifier Encoding and EVEX
EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in
64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support
instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory
addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction
syntax and memory addressing in 64-bit mode are shown in Table 2-31. Opmask register encoding is described in
Section 2.6.3.

Table 2-31. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits

<table>
<thead>
<tr>
<th></th>
<th>41</th>
<th>3</th>
<th>[2:0]</th>
<th>Reg. Type</th>
<th>Common Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>EVEX.R'</td>
<td>REX.R</td>
<td>modrm.reg</td>
<td>GPR, Vector</td>
<td>Destination or Source</td>
</tr>
<tr>
<td>NDS/NDD</td>
<td>EVEX.V'</td>
<td>EVEX.vvvv</td>
<td></td>
<td>GPR, Vector</td>
<td>2ndSource or Destination</td>
</tr>
<tr>
<td>RM</td>
<td>EVEX.X</td>
<td>EVEX.B</td>
<td>modrm.r/m</td>
<td>GPR, Vector</td>
<td>1st Source or Destination</td>
</tr>
<tr>
<td>BASE</td>
<td>0</td>
<td>EVEX.B</td>
<td>modrm.r/m</td>
<td>GPR</td>
<td>memory addressing</td>
</tr>
<tr>
<td>INDEX</td>
<td>0</td>
<td>EVEX.X</td>
<td>sib.index</td>
<td>GPR</td>
<td>memory addressing</td>
</tr>
<tr>
<td>VIDX</td>
<td>EVEX.V'</td>
<td>EVEX.X</td>
<td>sib.index</td>
<td>Vector</td>
<td>VSIB memory addressing</td>
</tr>
</tbody>
</table>

NOTES:
1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are
shown in Table 2-32.

Table 2-32. EVEX Encoding Register Specifiers in 32-bit Mode

<table>
<thead>
<tr>
<th></th>
<th>[2:0]</th>
<th>Reg. Type</th>
<th>Common Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>modrm.reg</td>
<td>GPR, Vector</td>
<td>Destination or Source</td>
</tr>
<tr>
<td>NDS/NDD</td>
<td>EVEX.vvv</td>
<td>GPR, Vector</td>
<td>2nd Source or Destination</td>
</tr>
<tr>
<td>RM</td>
<td>modrm.r/m</td>
<td>GPR, Vector</td>
<td>1st Source or Destination</td>
</tr>
<tr>
<td>BASE</td>
<td>modrm.r/m</td>
<td>GPR</td>
<td>Memory Addressing</td>
</tr>
<tr>
<td>INDEX</td>
<td>sib.index</td>
<td>GPR</td>
<td>Memory Addressing</td>
</tr>
<tr>
<td>VIDX</td>
<td>sib.index</td>
<td>Vector</td>
<td>VSIB Memory Addressing</td>
</tr>
</tbody>
</table>

2.6.3 Opmask Register Encoding
There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:
• Opmask registers that are the source or destination operands of an instruction treating the content of opmask
  register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using
  standard modR/M byte’s reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support
  conditional update of the destination operand.
• An opmask register providing conditional processing and/or conditional update of the destination register of a
  vector instruction is encoded using EVEX.aaa field (see Section 2.6.4).
• An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte’s reg field and rm fields.

### Table 2-33. Opmask Register Specifier Encoding

<table>
<thead>
<tr>
<th>[2:0]</th>
<th>Register Access</th>
<th>Common Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>modrm.reg</td>
<td>k0-k7</td>
</tr>
<tr>
<td>NDS</td>
<td>VEX.vvvv</td>
<td>k0-k7</td>
</tr>
<tr>
<td>RM</td>
<td>modrm.r/m</td>
<td>k0-7</td>
</tr>
<tr>
<td>{k1}</td>
<td>EVEX.aaa</td>
<td>k0¹-k7</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.

#### 2.6.4 Masking Support in EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.
- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided into the following groups:

- Instructions which support “zeroing-masking”.
  - Also allow merging-masking.
- Instructions which require aaa = 000.
  - Do not allow any form of masking.
- Instructions which allow merging-masking but do not allow zeroing-masking.
  - Require EVEX.z to be set to 0.
  - This group is mostly composed of instructions that write to memory.
- Instructions which require aaa <> 000 do not allow EVEX.z to be set to 1.
  - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

#### 2.6.5 Compressed Displacement (disp8*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-34 and Table 2-35 below, where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-34 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of
INSTRUCTION FORMAT

numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.6.11).

EVEX-encoded instruction that are pure load/store, and “Load+op” instruction semantic that operate on data element size less then dword do not support broadcasting using EVEX.b. These are listed in Table 2-35. Table 2-35 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-35. Instruction classified in Table 2-35 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tuple type will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8*N rules still apply when using 16b addressing.

<table>
<thead>
<tr>
<th>Table 2-34. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>TupleType</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Full</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Half</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2-35. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>TupleType</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Full Mem</td>
</tr>
<tr>
<td>Tuple1 Scalar</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tuple1 Fixed</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tuple2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tuple4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tuple8</td>
</tr>
<tr>
<td>Half Mem</td>
</tr>
<tr>
<td>Quarter Mem</td>
</tr>
<tr>
<td>Eighth Mem</td>
</tr>
<tr>
<td>Mem128</td>
</tr>
<tr>
<td>MOVVDUP</td>
</tr>
</tbody>
</table>
2.6.6 EVEX Encoding of Broadcast/Rounding/SAE Support
EVEX.b can provide three types of encoding context, depending on the instruction classes:

• Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with “load+op” semantic.

• Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.

• “Suppress All exceptions” (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that do not have rounding semantic.

2.6.7 Embedded Broadcast Support in EVEX
EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single-precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

• Instructions with only one scalar result is written to the vector destination.

• Instructions with explicit broadcast functionality provided by its opcode.

• Instruction semantic is a pure load or a pure store operation.

2.6.8 Static Rounding Support in EVEX
Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L’L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

2.6.9 SAE Support in EVEX
The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

2.6.10 Vector Length Orthogonality
The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L’L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L’L bits. When EVEX is used to encode scalar instructions, L’L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L’L fields are summarized in Table 2-36.
2.6.11 #UD Equations for EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

2.6.11.1 State Dependent #UD

In general, attempts of execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-37 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-37 will cause #UD.

<table>
<thead>
<tr>
<th>Instruction Categories</th>
<th>Vector Register State Access</th>
<th>Required XCR0 Bit Vector [7:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy SIMD prefix encoded Instructions (e.g SSE)</td>
<td>XMM</td>
<td>xxxxxxx11b</td>
</tr>
<tr>
<td>VEX-encoded instructions operating on YMM</td>
<td>YMM</td>
<td>xxxxxxx11b</td>
</tr>
<tr>
<td>EVEX-encoded 128-bit instructions</td>
<td>ZMM</td>
<td>111xx11b</td>
</tr>
<tr>
<td>EVEX-encoded 256-bit instructions</td>
<td>ZMM</td>
<td>111xx11b</td>
</tr>
<tr>
<td>EVEX-encoded 512-bit instructions</td>
<td>ZMM</td>
<td>111xx11b</td>
</tr>
<tr>
<td>VEX-encoded instructions operating on opmask</td>
<td>k-reg</td>
<td>xx1xxxx11b</td>
</tr>
</tbody>
</table>

2.6.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-38.

<table>
<thead>
<tr>
<th>Position</th>
<th>Notation</th>
<th>64-bit #UD</th>
<th>Non-64-bit #UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[3 : 2]</td>
<td>--</td>
<td>if &gt; 0</td>
<td>if &gt; 0</td>
</tr>
<tr>
<td>P[10]</td>
<td>--</td>
<td>if 0</td>
<td>if 0</td>
</tr>
<tr>
<td>P[1 : 0]</td>
<td>EVEX.mm</td>
<td>if 00b</td>
<td>if 00b</td>
</tr>
<tr>
<td>P[7 : 6]</td>
<td>EVEX.RX</td>
<td>None (valid)</td>
<td>None (BOUND if EVEX.RX != 11b)</td>
</tr>
</tbody>
</table>
### 2.6.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-39 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

#### Table 2-39. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields

<table>
<thead>
<tr>
<th>Notation</th>
<th>Position</th>
<th>Operand Encoding</th>
<th>64-bit #UD</th>
<th>Non-64-bit #UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.R</td>
<td>P[7]</td>
<td>ModRM.reg encodes k-reg</td>
<td>if EVEX.R = 0</td>
<td>None (BOUND if EVEX.RX != 11b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg is opcode extension</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg encodes all other registers</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td>EVEX.X</td>
<td>P[6]</td>
<td>ModRM.r/m encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m encodes k-reg or GPR</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m without SIB/VSIB</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m with SIB/VSIB</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td>EVEX.B</td>
<td>P[5]</td>
<td>ModRM.r/m encodes k-reg</td>
<td>None (ignored)</td>
<td>None (ignored)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m encodes other registers</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m base present</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m base not present</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td>EVEXR'</td>
<td>P[4]</td>
<td>ModRM.reg encodes k-reg or GPR</td>
<td>if 0</td>
<td>None (ignored)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg is opcode extension</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td>EVEX.vvvv</td>
<td>P[14 : 11]</td>
<td>vvvv encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td>None (valid)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Otherwise if != 1111b</td>
<td>if != 1111b</td>
<td>if != 1111b</td>
</tr>
<tr>
<td>EVEXV'</td>
<td>P[19]</td>
<td>Encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td>None (ignored)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Otherwise if 0</td>
<td>0</td>
<td>None (ignored)</td>
</tr>
</tbody>
</table>

Table 2-40 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

#### Table 2-40. #UD Conditions of Opmask Related Encoding Field

<table>
<thead>
<tr>
<th>Notation</th>
<th>Position</th>
<th>Operand Encoding</th>
<th>64-bit #UD</th>
<th>Non-64-bit #UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.aaa</td>
<td>P[18 : 16]</td>
<td>Instructions do not use opmask for conditional processing</td>
<td>if a0a = 000b</td>
<td>if a0a = 000b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opmask used as conditional processing mask and updated at completion</td>
<td>if a0a = 00b</td>
<td>if a0a = 00b;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opmask used as conditional processing.</td>
<td>None (valid3)</td>
<td>None (valid1)</td>
</tr>
<tr>
<td>EVEX.z</td>
<td>P[23]</td>
<td>Vector instruction using opmask as source or destination</td>
<td>if EVEX.z != 0</td>
<td>if EVEX.z != 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store instructions or gather/scatter instructions.</td>
<td>if EVEX.z != 0</td>
<td>if EVEX.z != 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction supporting conditional processing mask with EVEX.aaa = 000b.</td>
<td>if EVEX.z != 0</td>
<td>if EVEX.z != 0</td>
</tr>
<tr>
<td>VEX.vvvv</td>
<td>Varies</td>
<td>K-regs are instruction operands not mask control.</td>
<td>if vvvv = 0xxxb</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTES:**
1. E.g., VAACSTXxx, VPMOVM2x, VPMOVx2M.
2. E.g., Gather/Scatter family.
3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of ‘all ones’ regardless of the actual value in K0.
4. E.g., VFPCLASSPD/PS, VCMPPB/D/Q/W family, VPMOV2x, VPMOVx2M.

Table 2-41 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Position</th>
<th>Operand Encoding</th>
<th>64-bit #UD</th>
<th>Non-64-bit #UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.L'Lb</td>
<td>P[22:20]</td>
<td>Reg-reg, FP instructions with rounding semantic.</td>
<td>None (valid$^1$)</td>
<td>None (valid$^1$)</td>
</tr>
<tr>
<td>Other reg-reg, FP instructions that can cause #XF.</td>
<td></td>
<td>None (valid$^2$)</td>
<td>None (valid$^2$)</td>
<td></td>
</tr>
<tr>
<td>Other reg-mem instructions in Table 2-34.</td>
<td></td>
<td>None (valid$^3$)</td>
<td>None (valid$^3$)</td>
<td></td>
</tr>
<tr>
<td>Other instruction classes$^4$ in Table 2-35.</td>
<td></td>
<td>If EVEX.b &gt; 0</td>
<td>If EVEX.b &gt; 0</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. L'L specifies rounding control, see Table 2-36, supports {er} syntax.
2. L'L specifies vector length, see Table 2-36, supports {sae} syntax.
3. L'L specifies vector length, see Table 2-36, supports embedded broadcast syntax.
4. L'L specifies either vector length or ignored.

2.6.12 Device Not Available
EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3]= 1.

2.6.13 Scalar Instructions
EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

2.7 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS
The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of "E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction set</th>
<th>Mem arg</th>
<th>(#XM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type E1</td>
<td>Vector Moves/Load/Stores</td>
<td>Explicitly aligned, w/ fault suppression</td>
<td>None</td>
</tr>
<tr>
<td>Type E1NF</td>
<td>Vector Non-temporal Stores</td>
<td>Explicitly aligned, no fault suppression</td>
<td>None</td>
</tr>
<tr>
<td>Type E2</td>
<td>FP Vector Load+op</td>
<td>Support fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E2NF</td>
<td>FP Vector Load+op</td>
<td>No fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E3</td>
<td>FP Scalar/Partial Vector, Load+Op</td>
<td>Support fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E3NF</td>
<td>FP Scalar/Partial Vector, Load+Op</td>
<td>No fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E4</td>
<td>Integer Vector Load+op</td>
<td>Support fault suppression</td>
<td>No</td>
</tr>
<tr>
<td>Type E4NF</td>
<td>Integer Vector Load+op</td>
<td>No fault suppression</td>
<td>No</td>
</tr>
<tr>
<td>Type E5</td>
<td>Legacy-like Promotion</td>
<td>Varies, Support fault suppression</td>
<td>No</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMAT

Table 2-42. EVEX-Encoded Instruction Exception Class Summary
Exception Class

Instruction set

Mem arg

(#XM)

Type E5NF

Legacy-like Promotion

Varies, No fault suppression

No

Type E6

Post AVX Promotion

Varies, w/ fault suppression

No

Type E6NF

Post AVX Promotion

Varies, no fault suppression

No

Type E7NM

Register-to-register op

None

None

Type E9NF

Miscellaneous 128-bit

Vector-length Specific, no fault suppression

None

Type E10

Non-XF Scalar

Vector Length ignored, w/ fault suppression

None

Type E10NF

Non-XF Scalar

Vector Length ignored, no fault suppression

None

Type E11

VCVTPH2PS

Half Vector Length, w/ fault suppression

Yes

Type E11NF

VCVTPS2PH

Half Vector Length, no fault suppression

Yes

Type E12

Gather and Scatter Family

VSIB addressing, w/ fault suppression

None

Type E12NP

Gather and Scatter Prefetch Family

VSIB addressing, w/o page fault

None

Table 2-43 lists EVEX-encoded instruction mnemonic by exception classes.

Table 2-43. EVEX Instructions in each Exception Class
Exception Class
Type E1
Type E1NF

Instruction
VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
VADDPD, VADDPS, VCMPPD, VCMPPS, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPS2DQ, VCVTTPD2DQ,
VCVTTPS2DQ, VDIVPD, VDIVPS, VFMADDxxxPD, VFMADDxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPS,
VFMSUBxxxPD, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPS, VMAXPD,
VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS

Type E2

VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPS2UDQS, VCVTQQ2PD, VCVTQQ2PS, VCVTTPD2DQ,
VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PS, VCVTUQQ2PD,
VCVTUQQ2PS, VFIXUPIMMPD, VFIXUPIMMPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VRANGEPD,
VRANGEPS, VREDUCEPD, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPS, VSCALEFPD, VSCALEFPS, VRCP28PD,
VRCP28PS, VRSQRT28PD, VRSQRT28PS
VADDSD, VADDSS, VCMPSD, VCMPSS, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VDIVSD, VDIVSS, VMAXSD, VMAXSS,
VMINSD, VMINSS, VMULSD, VMULSS, VSQRTSD, VSQRTSS, VSUBSD, VSUBSS

Type E3

Type E3NF

VCVTPS2QQ, VCVTPS2UQQ, VCVTTPS2QQ, VCVTTPS2UQQ, VFMADDxxxSD, VFMADDxxxSS, VFMSUBxxxSD,
VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSS, VFIXUPIMMSD,
VFIXUPIMMSS, VGETEXPSD, VGETEXPSS, VGETMANTSD, VGETMANTSS, VRANGESD, VRANGESS, VREDUCESD,
VREDUCESS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD,
VRSQRT28SS
VCOMISD, VCOMISS, VCVTSD2SI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTTSD2SI, VCVTTSS2SI, VUCOMISD,
VUCOMISS
VCVTSD2USI, VCVTTSD2USI, VCVTSS2USI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SS

Vol. 2A 2-45


INSTRUCTION FORMAT

Table 2-43. EVEX Instructions in each Exception Class (Contd.)
Exception Class

Type E4

Instruction
VANDPD, VANDPS, VANDNPD, VANDNPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDD, VPADDQ, VPANDD, VPANDQ,
VPANDND, VPANDNQ, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPMAXSD, VPMAXSQ, VPMAXUD,
VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD,
VPORQ, VPSUBD, VPSUBQ, VPXORD, VPXORQ, VXORPD, VXORPS, VPSLLVD, VPSLLVQ,
VBLENDMPD, VBLENDMPS, VPBLENDMD, VPBLENDMQ, VFPCLASSPD, VFPCLASSPS, VPCMPD, VPCMPQ, VPCMPUD,
VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPROLD, VPROLQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)1,
VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VRCP14PD, VRCP14PS,
VRSQRT14PD, VRSQRT14PS, VPSRAVW, VPSRAVD, VPSRAVW, VPSRAVQ, VPMADD52LUQ, VPMADD52HUQ
VMOVUPD, VMOVUPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VPCMPB, VPCMPW, VPCMPUB,
VPCMPUW, VEXPANDPD, VEXPANDPS, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ,
VCOMPRESSPD, VCOMPRESSPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB,
VPADDUSW, VPAVGB, VPAVGW, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPMAXSB, VPMAXSW,
VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW,
VPMULLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW, VPSLLW,
VPSRAW, VPSRLW, VPSLLVW, VPSRLVW

E4.nb2

VPACKSSDW, VPACKUSDW VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFPD,
VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS, VPERMD, VPERMPS, VPERMPD, VPERMQ,
Type E4NF

E4NF.nb

2

Type E5

VALIGND, VALIGNQ, VPCONFLICTD, VPCONFLICTQ, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMT2D,
VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VSHUFI32X4, VSHUFI64X2, VSHUFF32X4,
VSHUFF64X2, VPMULTISHIFTQB
VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP,
VPSADBW, VPSHUFB, VPSHUFHW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD,
VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)3, VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD,
VPERMW, VPERMI2W, VPERMT2W
VCVTDQ2PD, PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ,
PMOVZXBW, PMOVZXBD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ
VCVTUDQ2PD

Type E5NF

VMOVDDUP
VBROADCASTSS, VBROADCASTSD, VBROADCASTF32X4, VBROADCASTI32X4, VPBROADCASTB, VPBROADCASTD,
VPBROADCASTW, VPBROADCASTQ,

Type E6

Type E6NF
Type
E7NM.1284
Type E7NM.

2-46 Vol. 2A

VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4,
VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4,
VFPCLASSSD, VFPCLASSSS, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW,
VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW,
VPMOVUSDW, VPMOVWB, VPMOVSWB, VPMOVUSWB
VEXTRACTF32X4, VEXTRACTF64X2, VEXTRACTF32X8, VINSERTF32X4, VINSERTF64X2, VINSERTF64X4,
VINSERTF32X8, VINSERTI32X4, VINSERTI64X2, VINSERTI64X4, VINSERTI32X8, VEXTRACTI32X4,
VEXTRACTI64X2, VEXTRACTI32X8, VEXTRACTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D
VMOVLHPS, VMOVHLPS
(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW)5, VPMOVM2B, VPMOVM2D, VPMOVM2Q,
VPMOVM2W, VPMOVB2M, VPMOVD2M, VPMOVQ2M, VPMOVW2M


### Table 2-43. EVEX Instructions in each Exception Class (Contd.)

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type E9NF</td>
<td>VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ</td>
</tr>
<tr>
<td>Type E10</td>
<td>VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRSQRT14SD, VRSQRT14SS,</td>
</tr>
<tr>
<td>Type E10NF</td>
<td>(VCVTIS2SD, VCVTUSI2SD)⁶</td>
</tr>
<tr>
<td>Type E11</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
</tr>
<tr>
<td>Type E12</td>
<td>VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQ, VPSGATTERDD, VPSGATTERDQ, VPSGATTERQD, VPSGATTERQQ, VSCATTERDPS, VSCATTERDQ, VSCATTERQD, VSCATTERQPS</td>
</tr>
<tr>
<td>Type E12NP</td>
<td>VGATHERPF0DPD, VGATHERPF0DPS, VGATHERPF0QPD, VGATHERPF0QPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPF0DPD, VSCATTERPF0DPS, VSCATTERPF0QPD, VSCATTERPF0QPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS</td>
</tr>
</tbody>
</table>

### NOTES:
1. Operand encoding Full tupletype with immediate.
2. Embedded broadcast is not supported with the “.nb” suffix.
3. Operand encoding Mem128 tupletype.
4. #UD raised if EVEX.L’L’L’O0b (VL=128).
5. The source operand is a general purpose register.
6. WO encoding only.
### 2.7.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1.

#### Table 2-44. Type E1 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any one of the following conditions applies:</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>• If EVEX.b 1= 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>• If EVEX.L'L 1= 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede an EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ’0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td>EVEX.512: Memory operand is not 64-byte aligned.</td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>EVEX.256: Memory operand is not 32-byte aligned.</td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>EVEX.128: Memory operand is not 16-byte aligned.</td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
</tbody>
</table>
EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L'L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
</tbody>
</table>

Table 2-45. Type E1NF Class Exception Conditions

Vol. 2A 2-49
### Exceptions Type E2 of EVEX-Encoded Instructions

EVEX-encoded vector instructions with arithmetic semantic follow exception class E2.

#### Table 2-46. Type E2 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L'L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX.B=1, alignment checking is enabled, and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If an unmasked SIMD floating-point exception, (sae) or (er) not set, and CR4.OSXMMEXCPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
2.7.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

EVEX-encoded scalar instructions with arithmetic semantic that support memory fault suppression follow exception class E3.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If EVEK prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSX-MMEXCPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
EVEX-encoded scalar instructions with arithmetic semantic that do not support memory fault suppression follow exception class E3NF.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X X</td>
<td>EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X X X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X</td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X X X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X X X X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS seg-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>or less is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSX-</td>
</tr>
</tbody>
</table>
### 2.7.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E4.

#### Table 2-49. Type E4 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual/80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- If EVEX.b = 0 and in E4.nb subclass (see E4.nb entries in Table 2-43).</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- If EVEX.L’L != 10b (VL=512).</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If fault suppression not set, and any part of the operand lies outside the</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX.B=1, alignment checking is enabled, and an unaligned memory reference of</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8 bytes or less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
EVEX-encoded vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E4NF.

### Table 2-50. Type E4NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ If EVEX.b != 0 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-43).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ If EVEX.L’L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>
2.7.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

Table 2-51. Type E5 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L'.L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>

EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.
### Table 2-52. Type E5NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L’L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
### Exceptions Type E6 and E6NF

#### Table 2-53. Type E6 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L' != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

**Table 2-54. Type E6NF Class Exception Conditions**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L'L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
2.7.7 **Exceptions Type E7NM**

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM.

<table>
<thead>
<tr>
<th>Table 2-55. Type E7NM Class Exception Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exception</strong></td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>Invalid Opcode, #UD</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
## 2.7.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

### Table 2-56. Type E9 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L’L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L’L != 00b (VL=128).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
## 2.7.9 Exceptions Type E10

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding and do not cause no SIMD FP exception, support memory fault suppression follow exception class E10.

### Table 2-58. Type E10 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
EVEX-encoded scalar instructions that must be encoded with VEX.L’L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E10NF.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
2.7.10 Exception Type E11 (EVEX-only, mem arg no AC, floating-point exceptions)

EVEX-encoded instructions that can cause SIMD FP exception, memory operand support fault suppression but do not cause #AC follow exception class E11.

Table 2-60. Type E11 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility 64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L'L != 10b (VL=512).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF (fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
</tbody>
</table>
## Exception Type E12 and E12NP (VSIB mem arg, no AC, no floating-point exceptions)

### Table 2-61. Type E12 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility 64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If CR4.OSXSAVE(bit 18)=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L’L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If vvvv != 1111b.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (FOH).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>NA</td>
<td>If address size attribute is 16 bit.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If ModR/M.mod = ‘11b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If ModR/M.rm != ‘100b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If k0 is used (gather or scatter operation).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If index = destination register (gather operation).</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF (fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>
EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

### Table 2-62. Type E12NP Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility 64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X X</td>
<td>X X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.b != 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• If EVEX.L’L != 10b (VL=512).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X X</td>
<td>X X If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>NA</td>
<td>X X X X If address size attribute is 16 bit.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X X</td>
<td>X X X X If ModR/M.mod = ‘11b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X X</td>
<td>X X X X If ModR/M.rm != ‘100b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X X</td>
<td>X X X X If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X X</td>
<td>X X X X If k0 is used (gather or scatter operation).</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X X</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
</tbody>
</table>
### 2.8 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS

The exception behavior of VEX-encoded opmask instructions are listed below. Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

#### Table 2-63. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If relevant CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If a VEX prefix is present.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
</tbody>
</table>
Exception conditions of Opmask instructions that address memory are listed as Type K21.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual/80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X X</td>
<td>If relevant CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If a VEX prefix is present.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X X</td>
<td>If CR4.OSXSAVE[bit 18]=0.                                                                                                           • State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>

For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

### 3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections.

#### 3.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The heading below introduces the example. The table below provides an example summary table.

**CMC—Complement Carry Flag [this is an example]**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>CMC</td>
<td>ZO</td>
<td>V/V</td>
<td>NA</td>
<td>Complement carry flag.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix)

The “Opcode” column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **NP** — Indicates the use of 66/F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.

- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.

- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction’s opcode.

- **/r** — Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.

- **cb, cw, cd, cp, co, ct** — A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.

- **ib, iw, id, io** — A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords and quadwords are given with the low-order byte first.

- **+rb, +rw, +rd, +ro** — Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. “+ro” is applicable only in 64-bit mode. See Table 3-1 for the codes.

- **+i** — A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

### Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

<table>
<thead>
<tr>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>None</td>
<td>0</td>
<td>AX</td>
<td>None</td>
<td>0</td>
<td>EAX</td>
<td>None</td>
<td>0</td>
<td>RAX</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>CL</td>
<td>None</td>
<td>1</td>
<td>CX</td>
<td>None</td>
<td>1</td>
<td>ECX</td>
<td>None</td>
<td>1</td>
<td>RCX</td>
<td>None</td>
<td>1</td>
</tr>
<tr>
<td>DL</td>
<td>None</td>
<td>2</td>
<td>DX</td>
<td>None</td>
<td>2</td>
<td>EDX</td>
<td>None</td>
<td>2</td>
<td>RDX</td>
<td>None</td>
<td>2</td>
</tr>
<tr>
<td>BL</td>
<td>None</td>
<td>3</td>
<td>BX</td>
<td>None</td>
<td>3</td>
<td>EBX</td>
<td>None</td>
<td>3</td>
<td>RBX</td>
<td>None</td>
<td>3</td>
</tr>
<tr>
<td>AH</td>
<td>Not encodable (N.E.)</td>
<td>4</td>
<td>SP</td>
<td>None</td>
<td>4</td>
<td>ESP</td>
<td>None</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CH</td>
<td>N.E.</td>
<td>5</td>
<td>BP</td>
<td>None</td>
<td>5</td>
<td>EBP</td>
<td>None</td>
<td>5</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>DH</td>
<td>N.E.</td>
<td>6</td>
<td>SI</td>
<td>None</td>
<td>6</td>
<td>ESI</td>
<td>None</td>
<td>6</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BH</td>
<td>N.E.</td>
<td>7</td>
<td>DI</td>
<td>None</td>
<td>7</td>
<td>EDI</td>
<td>None</td>
<td>7</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SPL</td>
<td>Yes</td>
<td>4</td>
<td>SP</td>
<td>None</td>
<td>4</td>
<td>ESP</td>
<td>None</td>
<td>4</td>
<td>RSP</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>BPL</td>
<td>Yes</td>
<td>5</td>
<td>BP</td>
<td>None</td>
<td>5</td>
<td>EBP</td>
<td>None</td>
<td>5</td>
<td>RBP</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>SIL</td>
<td>Yes</td>
<td>6</td>
<td>SI</td>
<td>None</td>
<td>6</td>
<td>ESI</td>
<td>None</td>
<td>6</td>
<td>RSI</td>
<td>None</td>
<td>6</td>
</tr>
<tr>
<td>DIL</td>
<td>Yes</td>
<td>7</td>
<td>DI</td>
<td>None</td>
<td>7</td>
<td>EDI</td>
<td>None</td>
<td>7</td>
<td>RDI</td>
<td>None</td>
<td>7</td>
</tr>
</tbody>
</table>

Registers R8 - R15 (see below): Available in 64-Bit Mode Only
3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix)

In the Instruction Summary Table, the Opcode column presents each instruction encoded using the VEX prefix in following form (including the modR/M byte if applicable, the immediate byte if applicable):

\[
\text{VEX.}[\text{NDS}].[128,256].[66,F2,F3].0F/0F3A/0F38.[W0,W1] \text{ opcode [/r] [/ib,/is4]}
\]

- **VEX** — Indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 2.3 for more detail on the VEX prefix.

The encoding of various sub-fields of the VEX prefix is described using the following notations:

- **NDS, NDD, DDS**: Specifies that VEX.vvvv field is valid for the encoding of a register operand:
  - **VEX.NDS**: VEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved.
  - **VEX.NDD**: VEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field.
  - **VEX.DDS**: VEX.vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.
  - If none of NDS, NDD, and DDS is present, VEX.vvvv must be 1111b (i.e. VEX.vvvv does not encode an operand). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.

- **128,256**: VEX.L field can be 0 (denoted by VEX.128 or VEX.LZ) or 1 (denoted by VEX.256). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.128 or VEX.256 in the opcode column should be interpreted as follows:
  - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L = 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
  - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g. VMOVLPS).

- **VEX.LIG**: VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distin-
guished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions.

- If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.

- **66,F2,F3**: The presence or absence of these values map to the VEX.pp field encodings. If absent, this corresponds to VEX.pp=00B. If present, the corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix. The VEX.pp field may be encoded using either the 2-byte or 3-byte form of the VEX prefix.

- **0F,0F3A,0F38**: The presence maps to a valid encoding of the VEX.mmmmmm field. Only three encoded values of VEX.mmmmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid VEX.mmmmmm encoding on the ensuing opcode byte is same as if the corresponding escape byte sequence on the ensuing opcode byte for non-VEX encoded instructions. Thus a valid encoding of VEX.mmmmmm may be consider as an implies escape byte sequence of either 0FH, 0F3AH or 0F38H. The VEX.mmmmmm field must be encoded using the 3-byte form of VEX prefix.

- **0F,0F3A,0F38 and 2-byte/3-byte VEX**: The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded using the two-byte form of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.

- **W0**: VEX.W=0.
- **W1**: VEX.W=1.

The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the two-byte of VEX if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 2.3 on the subfield definitions within VEX.

- **WIG**: can use CSH form (if not requiring VEX.mmmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.

- If WIG is present, the instruction may be encoded using either the two-byte form or the three-byte form of VEX. When encoding the instruction using the three-byte form of VEX, the value of VEX.W is ignored.

- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in either imm8[7:4] (for 64-bit mode) or imm8[6:4] (for 32-bit mode), and instruction-specific payload in imm8[3:0].
- In general, the encoding of VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

**EVEX.[NDS/NDD/DDS].[128,256,512,LIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [ib]**

- **EVEX** — The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 2.6.1 for more detail on the EVEX prefix.

The encoding of various sub-fields of the EVEX prefix is described using the following notations:

- **NDS, NDD, DDS**: implies that EVEX.vvvv (and EVEX.v’) field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). DDS expresses a syntax where vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result. If both NDS and NDD absent (i.e. EVEX.vvvv does not encode an operand), EVEX.vvvv must be 1111b (and EVEX.v’ must be 1b).

- **128, 256, 512, LIG**: This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.
— 66,F2,F3: The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.

— 0F,0F3A,0F38: The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.

— W0: EVEX.W=0.
— W1: EVEX.W=1.
— WIG: EVEX.W bit ignored

* opcode — Instruction opcode.

* In general, the encoding of EVEX.R and R’, EVEX.X and X’, and EVEX.B and B’ fields are not shown explicitly in the opcode column.

### 3.1.1.3 Instruction Column in the Opcode Summary Table

The "Instruction" column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

* rel8 — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

* rel16, rel32 — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.

* ptr16:16, ptr16:32 — A far pointer, typically to a code segment different from that of the instruction. The notation 16:16 indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.

* r8 — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode.

* r16 — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.

* r32 — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.

* r64 — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15. These are available when using REX.R and 64-bit mode.

* imm8 — An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

* imm16 — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.

* imm32 — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.

* imm64 — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and –9,223,372,036,854,775,808 inclusive.
• \( r/m_8 \) — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode.

• \( r/m_{16} \) — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.

• \( r/m_{32} \) — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.

• \( r/m_{64} \) — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

• \( m \) — A 16-, 32- or 64-bit operand in memory.

• \( m_{8} \) — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.

• \( m_{16} \) — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.

• \( m_{32} \) — A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.

• \( m_{64} \) — A memory quadword operand in memory.

• \( m_{128} \) — A memory double quadword operand in memory.

• \( m_{16:16}, m_{16:32} \) — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

• \( m_{16:32}, m_{16:16}, m_{32:32}, m_{16:64} \) — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16:16 and m32:32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.

• \( m_{80bcd} \) — A Binary Coded Decimal (BCD) operand in memory, 80 bits.

• \( moffs_{8}, moffs_{16}, moffs_{32}, moffs_{64} \) — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

• \( Sreg \) — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.

• \( m_{32fp}, m_{64fp}, m_{80fp} \) — A single-precision, double-precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.

• \( m_{16int}, m_{32int}, m_{64int} \) — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.

• \( ST \) or \( ST(0) \) — The top element of the FPU register stack.

• \( ST(i) \) — The \( i \)th element from the top of the FPU register stack (\( i \leq 0 \) through 7).

• \( mm \) — An MMX register. The 64-bit MMX registers are: MM0 through MM7.
- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

- **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

- **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.

- **xmm/m32** — An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

- **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

- **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

- **<XMM0>** — Indicates implied use of the XMM0 register.

  When there is ambiguity, xmm1 indicates the first source operand using an XMM register and xmm2 the second source operand using an XMM register.

  Some instructions use the XMM0 register as the third source operand, indicated by <XMM0>. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.

- **ymm** — A YMM register. The 256-bit YMM registers are: YMM0 through YMM7; YMM8 through YMM15 are available in 64-bit mode.

- **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX instructions.

- **ymm/m256** — A YMM register or 256-bit memory operand.

- **<YMM0>** — Indicates use of the YMM0 register as an implicit argument.

- **bnd** — A 128-bit bounds register. BND0 through BND3.

- **mib** — A memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.

- **m512** — A 64-byte operand in memory.

- **zmm/m512** — A ZMM register or 512-bit memory operand.

- **{k1}{z}** — A mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.

- **{k1}** — Without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.

- **k1** — A mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.

- **mV** — A vector memory operand; the operand size is dependent on the instruction.

- **vm32{x,y,z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).

- **vm64{x,y,z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).
• **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.

• **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.

• **<ZMM0>** — Indicates use of the ZMM0 register as an implicit argument.

• **{er}** — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).

• **{sae}** — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.

• **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.

• **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.

• **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.

• **SRC** — The source in a single-source instruction.

• **DST** — the destination in an instruction. This field is encoded by reg_field.

### 3.1.1.4 Operand Encoding Column in the Instruction Summary Table

The “operand encoding” column is abbreviated as Op/En in the Instruction Summary table heading. Instruction operand encoding information is provided for each assembly instruction syntax using a letter to cross reference to a row entry in the operand encoding definition table that follows the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

EVEX encoded instructions employ compressed disp8*N encoding of the displacement bytes, where N is defined in Table 2-34 and Table 2-35, according to tupletypes. The tupletype for an instruction is listed in the operand encoding definition table where applicable.

**NOTES**

- The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.
- In the encoding definition table, the letter ‘r’ within a pair of parenthesis denotes the content of the operand will be read by the processor. The letter ‘w’ within a pair of parenthesis denotes the content of the operand will be updated by the processor.

### 3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table

The “64/32-bit Mode” column indicates whether the opcode sequence is supported in (a) 64-bit mode or (b) the Compatibility mode and other IA-32 modes that apply in conjunction with the CPUID feature flag associated specific instruction extensions.

The 64-bit mode support is to the left of the ‘slash’ and has the following notation:

- **V** — Supported.
- **I** — Not supported.
- **N.E.** — Indicates an instruction syntax is not encodable in 64-bit mode (it may represent part of a sequence of valid instructions in other modes).
- **N.P.** — Indicates the REX prefix does not affect the legacy instruction in 64-bit mode.
- **N.I.** — Indicates the opcode is treated as a new instruction in 64-bit mode.
- **N.S.** — Indicates an instruction syntax that requires an address override prefix in 64-bit mode and is not supported. Using an address override prefix in 64-bit mode may result in model-specific execution behavior.
The Compatibility/Legacy Mode support is to the right of the ‘slash’ and has the following notation:
• **V** — Supported.
• **I** — Not supported.
• **N.E.** — Indicates an Intel 64 instruction mnemonics/syntax that is not encodable; the opcode sequence is not applicable as an individual instruction in compatibility mode or IA-32 mode. The opcode may represent a valid sequence of legacy IA-32 instructions.

3.1.1.6 CPUID Support Column in the Instruction Summary Table
The fourth column holds abbreviated CPUID feature flags (e.g., appropriate bit in CPUID.1.ECX, CPUID.1.EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AESNI/PCLMULQDQ/AVX/RDRAND support) that indicate processor support for the instruction. If the corresponding flag is ‘0’, the instruction will #UD.

3.1.1.7 Description Column in the Instruction Summary Table
The “Description” column briefly explains forms of the instruction.

3.1.1.8 Description Section
Each instruction is then described by number of information sections. The “Description” section describes the purpose of the instructions and required operands in more detail.

Summary of terms that may be used in the description section:
• **Legacy SSE** — Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
• **VEX.vvvv** — The VEX bit field specifying a source or destination register (in 1’s complement form).
• **rm_field** — shorthand for the ModR/M r/m field and any REX.B
• **reg_field** — shorthand for the ModR/M reg field and any REX.R

3.1.1.9 Operation Section
The “Operation” section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:
• Comments are enclosed within the symbol pairs “(*” and “*)”.
• Compound statements are enclosed in keywords, such as: IF, THEN, ELSE and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
• A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register’s default segment (DS) or the overridden segment.
• Parentheses around the “E” in a general-purpose register name, such as (ESI), indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the “R” in a general-purpose register name, (RSI), in the presence of a 64-bit register definition such as (RSI), indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.
• Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
• A ← B indicates that the value of B is assigned to A.
• The symbols =, ≠, >, <, ≥, and ≤ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.
The expression "« COUNT" and "» COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize** and **AddressSize** — The OperandSize identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The AddressSize identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

  ```plaintext
  IF Instruction = MOVW
      THEN OperandSize ← 16;
  ELSE
      IF Instruction = MOVD
          THEN OperandSize ← 32;
      ELSE
          IF Instruction = MOVQ
              THEN OperandSize ← 64;
          Fi;
      Fi;
  Fi;
  Fi;
  ```

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for guidelines on how these attributes are determined.

- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

- **SRC** — Represents the source operand.

- **DEST** — Represents the destination operand.

- **MAXVL** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction’s encoding but is instead determined by the current value of XCR0. For details, refer to the table below. Note that the value of MAXVL is the largest of the features enabled. Future processors may define new bits in XCR0 whose setting may imply other values for MAXVL.

<table>
<thead>
<tr>
<th>XCR0 Component</th>
<th>MAXVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCR0.SSE</td>
<td>128</td>
</tr>
<tr>
<td>XCR0.AVX</td>
<td>256</td>
</tr>
<tr>
<td>XCR0.{ZMM_Hi256, Hi16_ZMM, OPMASK}</td>
<td>512</td>
</tr>
</tbody>
</table>

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.

- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
• **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

• **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).

• **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).

• **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

• **SaturateToUnsignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).

• **SaturateToUnsignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).

• **LowOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.

• **HighOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.

• **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the "Operation" subsection of the "PUSH—Push Word, Doubleword or Quadword Onto the Stack" section in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

• **Pop()** — Removes the value from the top of the stack and returns it. The statement EAX ← Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the "Operation" subsection in the "POP—Pop a Value from the Stack" section of Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

• **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.

• **Switch-Tasks** — Performs a task switch.

• **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

![Figure 3-1. Bit Offset for BIT[RAX, 21]](image-url)
If BitBase is a memory address, the BitOffset has different ranges depending on the operand size (see Table 3-2).

### Table 3-2. Range of Bit Positions Specified by Bit Offset Operands

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Immediate BitOffset</th>
<th>Register BitOffset</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0 to 15</td>
<td>$-2^{15}$ to $2^{15} - 1$</td>
</tr>
<tr>
<td>32</td>
<td>0 to 31</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
</tr>
<tr>
<td>64</td>
<td>0 to 63</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
</tr>
</tbody>
</table>

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

![Figure 3-2. Memory Bit Indexing](image)

### 3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section

The Intel C/C++ compiler intrinsic functions give access to the full power of the Intel Architecture Instruction Set, while allowing the compiler to optimize register allocation and instruction scheduling for faster execution. Most of these functions are associated with a single IA instruction, although some may generate multiple instructions or different instructions depending upon how they are used. In particular, these functions are used to invoke instructions that perform operations on vector registers that can hold multiple data elements. These SIMD instructions use the following data types.

- **__m128, __m256 and __m512** can represent 4, 8 or 16 packed single-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128 data type is also used with various single-precision floating-point scalar instructions that perform calculations using only the lowest 32 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.

- **__m128d, __m256d and __m512d** can represent 2, 4 or 8 packed double-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128d data type is also used with various double-precision floating-point scalar instructions that perform calculations using only the lowest 64 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.

- **__m128i, __m256i and __m512i** can represent integer data in bytes, words, doublewords, quadwords, and occasionally larger data types.
Each of these data types incorporates in its name the number of bits it can hold. For example, the __m128 type holds 128 bits, and because each single-precision floating-point value is 32 bits long the __m128 type holds (128/32) or four values. Normally the compiler will allocate memory for these data types on an even multiple of the size of the type. Such aligned memory locations may be faster to read and write than locations at other addresses.

These SIMD data types are not basic Standard C data types or C++ objects, so they may be used only with the assignment operator, passed as function arguments, and returned from a function call. If you access the internal members of these types directly, or indirectly by using them in a union, there may be side effects affecting optimization, so it is recommended to use them only with the SIMD instruction intrinsic functions described in this manual or the Intel C/C++ compiler documentation.

Many intrinsic functions names are prefixed with an indicator of the vector length and suffixed by an indicator of the vector element data type, although some functions do not follow the rules below. The prefixes are:

- __mm_ indicates that the function operates on 128-bit (or sometimes 64-bit) vectors.
- __mm256_ indicates the function operates on 256-bit vectors.
- __mm512_ indicates that the function operates on 512-bit vectors.

The suffixes include:

- _ps, which indicates a function that operates on packed single-precision floating-point data. Packed single-precision floating-point data corresponds to arrays of the C/C++ type float with either 4, 8 or 16 elements. Values of this type can be loaded from an array using the __mm_loadu_ps, __mm256_loadu_ps, or __mm512_loadu_ps functions, or created from individual values using __mm_set_ps, __mm256_set_ps, or __mm512_set_ps functions, and they can be stored in an array using __mm_storeu_ps, __mm256_storeu_ps, or __mm512_storeu_ps.
- _ss, which indicates a function that operates on scalar single-precision floating-point data. Single-precision floating-point data corresponds to the C/C++ type float, and values of type float can be converted to type __m128 for use with these functions using the __mm_set_ss function, and converted back using the __mm_cvtss_f32 function. When used with functions that operate on packed single-precision floating-point data the scalar element corresponds with the first packed value.
- _pd, which indicates a function that operates on packed double-precision floating-point data. Packed double-precision floating-point data corresponds to arrays of the C/C++ type double with either 2, 4, or 8 elements. Values of this type can be loaded from an array using the __mm_loadu_pd, __mm256_loadu_pd, or __mm512_loadu_pd functions, or created from individual values using __mm_set_pd, __mm256_set_pd, or __mm512_set_pd functions, and they can be stored in an array using __mm_storeu_pd, __mm256_storeu_pd, or __mm512_storeu_pd.
- _sd, which indicates a function that operates on scalar double-precision floating-point data. Double-precision floating-point data corresponds to the C/C++ type double, and values of type double can be converted to type __m128d for use with these functions using the __mm_set_sd function, and converted back using the __mm_cvtss_f64 function. When used with functions that operate on packed double-precision floating-point data the scalar element corresponds with the first packed value.
- _epi8, which indicates a function that operates on packed 8-bit signed integer values. Packed 8-bit signed integers correspond to an array of signed char with 16, 32 or 64 elements. Values of this type can be created from individual elements using __mm_set_epi8, __mm256_set_epi8, or __mm512_set_epi8 functions.
- _epi16, which indicates a function that operates on packed 16-bit signed integer values. Packed 16-bit signed integers correspond to an array of short with 8, 16 or 32 elements. Values of this type can be created from individual elements using __mm_set_epi16, __mm256_set_epi16, or __mm512_set_epi16 functions.
- _epi32, which indicates a function that operates on packed 32-bit signed integer values. Packed 32-bit signed integers correspond to an array of int with 4, 8 or 16 elements. Values of this type can be created from individual elements using __mm_set_epi32, __mm256_set_epi32, or __mm512_set_epi32 functions.
- _epi64, which indicates a function that operates on packed 64-bit signed integer values. Packed 64-bit signed integers correspond to an array of long long (or long if it is a 64-bit data type) with 2, 4 or 8 elements. Values of this type can be created from individual elements using __mm_set_epi32, __mm256_set_epi32, or __mm512_set_epi32 functions.
- _epu8, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of unsigned char with 16, 32 or 64 elements.
• \_epu16, which indicates a function that operates on packed 16-bit unsigned integer values. Packed 16-bit unsigned integers correspond to an array of unsigned short with 8, 16 or 32 elements.

• \_epu32, which indicates a function that operates on packed 32-bit unsigned integer values. Packed 32-bit unsigned integers correspond to an array of unsigned with 4, 8 or 16 elements.

• \_epu64, which indicates a function that operates on packed 64-bit unsigned integer values. Packed 64-bit unsigned integers correspond to an array of unsigned long long (or unsigned long if it is a 64-bit data type) with 2, 4 or 8 elements.

• \_si128, which indicates a function that operates on a single 128-bit value of type \_m128i.

• \_si256, which indicates a function that operates on a single a 256-bit value of type \_m256i.

• \_si512, which indicates a function that operates on a single a 512-bit value of type \_m512i.

Values of any packed integer type can be loaded from an array using the \_mm\_load\_u\_si128, \_mm\_load\_u\_si256, or \_mm\_load\_u\_si512 functions, and they can be stored in an array using \_mm\_store\_u\_si128, \_mm\_store\_u\_si256, or \_mm\_store\_u\_si512.

These functions and data types are used with the SSE, AVX, and AVX-512 instruction set extension families. In addition there are similar functions that correspond to MMX instructions. These are less frequently used because they require additional state management, and only operate on 64-bit packed integer values.

The declarations of Intel C/C++ compiler intrinsic functions may reference some non-standard data types, such as \_int64. The C Standard header stdint.h defines similar platform-independent types, and the documentation for that header gives characteristics that apply to corresponding non-standard types according to the following table.

Table 3-3. Standard and Non-standard Data Types

<table>
<thead>
<tr>
<th>Non-standard Type</th>
<th>Standard Type (from stdint.h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_int64</td>
<td>int64_t</td>
</tr>
<tr>
<td>unsigned _int64</td>
<td>uint64_t</td>
</tr>
<tr>
<td>_int32</td>
<td>int32_t</td>
</tr>
<tr>
<td>unsigned _int32</td>
<td>uint32_t</td>
</tr>
<tr>
<td>_int16</td>
<td>int16_t</td>
</tr>
<tr>
<td>unsigned _int16</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>


3.1.1.11 Flags Affected Section

The “Flags Affected” section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, “EFLAGS Cross-Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Non-conventional assignments are described in the “Operation” section. The values of flags listed as undefined may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

3.1.1.12 FPU Flags Affected Section

The floating-point instructions have an “FPU Flags Affected” section that describes how each instruction can affect the four condition code flags of the FPU status word.

3.1.1.13 Protected Mode Exceptions Section

The “Protected Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound
sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-4 associates each two-letter mnemonic with the corresponding exception vector and name. See Chapter 6, “Procedure Calls, Interrupts, and Exceptions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

### Table 3-4. Intel 64 and IA-32 General Exceptions

<table>
<thead>
<tr>
<th>Vector</th>
<th>Name</th>
<th>Source</th>
<th>Protected Mode</th>
<th>Real Address Mode</th>
<th>Virtual 8086 Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE—Divide Error</td>
<td>DIV and IDIV instructions.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>#DB—Debug</td>
<td>Any code or data reference.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>#BP—Breakpoint</td>
<td>INT3 instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>#OF—Overflow</td>
<td>INTO instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>#BR—BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>#UD—Invalid Opcode (Undefined Opcode)</td>
<td>UD instruction or reserved opcode.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>#NM—Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>#DF—Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>#TS—Invalid TSS</td>
<td>Task switch or TSS access.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>#NP—Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>#SS—Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>#GP—General Protection</td>
<td>Any memory reference and other protection checks.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>#PF—Page Fault</td>
<td>Any memory reference.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>#MF—Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>#AC—Alignment Check</td>
<td>Any data reference in memory.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>#MC—Machine Check</td>
<td>Model dependent machine check errors.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>#XM—SIMD Floating-Point Numeric Error</td>
<td>SSE/SSE2/SSE3 floating-point instructions.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.

### 3.1.1.14 Real-Address Mode Exceptions Section

The “Real-Address Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-4).

### 3.1.1.15 Virtual-8086 Mode Exceptions Section

The “Virtual-8086 Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-4).
3.1.1.16 Floating-Point Exceptions Section

The “Floating-Point Exceptions” section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, exception 16) being generated. Table 3-5 associates a one- or two-letter mnemonic with the corresponding exception name. See “Floating-Point Exception Conditions” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a detailed description of these exceptions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>#IS</td>
<td>Floating-point invalid operation:</td>
<td>- Stack overflow or underflow</td>
</tr>
<tr>
<td>#IA</td>
<td>- Invalid arithmetic operation</td>
<td>- x87 FPU stack overflow or underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Invalid FPU arithmetic operation</td>
</tr>
<tr>
<td>#Z</td>
<td>Floating-point divide-by-zero</td>
<td>Divide-by-zero</td>
</tr>
<tr>
<td>#D</td>
<td>Floating-point denormal operand</td>
<td>Source operand that is a denormal number</td>
</tr>
<tr>
<td>#O</td>
<td>Floating-point numeric overflow</td>
<td>Overflow in result</td>
</tr>
<tr>
<td>#U</td>
<td>Floating-point numeric underflow</td>
<td>Underflow in result</td>
</tr>
<tr>
<td>#P</td>
<td>Floating-point inexact result (precision)</td>
<td>Inexact result (precision)</td>
</tr>
</tbody>
</table>

3.1.1.17 SIMD Floating-Point Exceptions Section

The “SIMD Floating-Point Exceptions” section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, exception 19) being generated. Table 3-6 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to “SSE and SSE2 Exceptions”, in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>#I</td>
<td>Floating-point invalid operation</td>
<td>Invalid arithmetic operation or source operand</td>
</tr>
<tr>
<td>#Z</td>
<td>Floating-point divide-by-zero</td>
<td>Divide-by-zero</td>
</tr>
<tr>
<td>#D</td>
<td>Floating-point denormal operand</td>
<td>Source operand that is a denormal number</td>
</tr>
<tr>
<td>#O</td>
<td>Floating-point numeric overflow</td>
<td>Overflow in result</td>
</tr>
<tr>
<td>#U</td>
<td>Floating-point numeric underflow</td>
<td>Underflow in result</td>
</tr>
<tr>
<td>#P</td>
<td>Floating-point inexact result</td>
<td>Inexact result (precision)</td>
</tr>
</tbody>
</table>

3.1.1.18 Compatibility Mode Exceptions Section

This section lists exceptions that occur within compatibility mode.

3.1.1.19 64-Bit Mode Exceptions Section

This section lists exceptions that occur within 64-bit mode.
3.2 INSTRUCTIONS (A-L)

AAA—ASCII Adjust After Addition

Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode
THEN
#UD;
ELSE
IF ((AL AND 0FH) > 9) or (AF = 1)
THEN
AX ← AX + 106H;
AF ← 1;
CF ← 1;
ELSE
AF ← 0;
CF ← 0;
FI;
AL ← AL AND 0FH;
FI;

Flags Affected

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.
Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
### AAD—ASCII Adjust AX Before Division

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D5 0A</td>
<td>AAD</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>ASCII adjust AX before division.</td>
</tr>
<tr>
<td>D5 ib</td>
<td>AAD imm8</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Adjust AX before division to number base imm8.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to \((\text{AL} + (10 \times \text{AH}))\), and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the "Operation" section below), by setting the \(\text{imm8}\) byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 \(\text{imm8}\)).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

IF 64-Bit Mode THEN

#UD;

ELSE

\[
\begin{align*}
\text{tempAL} & \leftarrow \text{AL}; \\
\text{tempAH} & \leftarrow \text{AH}; \\
\text{AL} & \leftarrow (\text{tempAL} + (\text{tempAH} \times \text{imm8})) \text{ AND FFH}; \\
(* & \text{imm8 is set to 0AH for the AAD mnemonic.}) \\
\text{AH} & \leftarrow 0;
\end{align*}
\]

FI;

The immediate value (\(\text{imm8}\)) is taken from the second byte of the instruction.

### Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as protected mode.

Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
AAM—ASCII Adjust AX After Multiply

**Description**

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the \( \text{imm8} \) byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 \( \text{imm8} \)).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

**Operation**

IF 64-Bit Mode

THEN

#UD;

ELSE

\[ \text{tempAL} \leftarrow \text{AL}; \]
\[ \text{AH} \leftarrow \text{tempAL} / \text{imm8}; \] (* \( \text{imm8} \) is set to 0AH for the AAM mnemonic *)
\[ \text{AL} \leftarrow \text{tempAL} \mod \text{imm8}; \]

FI;

The immediate value (\( \text{imm8} \)) is taken from the second byte of the instruction.

**Flags Affected**

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

**Protected Mode Exceptions**

#DE If an immediate value of 0 is used.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as protected mode.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

---

**Opcode Instruction Op/En 64-bit Mode Compat/ Leg Mode Description**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4 0A</td>
<td>AAM</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>ASCII adjust AX after multiply.</td>
</tr>
<tr>
<td>D4 ( \text{ib} )</td>
<td>AAM ( \text{imm8} )</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Adjust AX after multiply to number base ( \text{imm8} ).</td>
</tr>
</tbody>
</table>
Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
AAS—ASCII Adjust AL After Subtraction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F</td>
<td>AAS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>ASCII adjust AL after subtraction.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top four bits set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-bit mode
    THEN
    #UD;
    ELSE
    IF ((AL AND 0FH) > 9) or (AF = 1)
        THEN
            AX ← AX - 6;
            AH ← AH - 1;
            AF ← 1;
            CF ← 1;
            AL ← AL AND 0FH;
        ELSE
            CF ← 0;
            AF ← 0;
            AL ← AL AND 0FH;
    ELSE
    Fi;
    Fi;

Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as protected mode.

Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.
ADC—Add with Carry

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.*
The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[ \text{DEST} \leftarrow \text{DEST} + \text{SRC} + \text{CF} ; \]

**Intel C/C++ Compiler Intrinsic Equivalent**

ADC: `extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);`

ADC: `extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);`

ADC: `extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *sum_out);`

ADC: `extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);`

**Flags Affected**

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

**Protected Mode Exceptions**

- #GP(0) If the destination is located in a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 F6 r</td>
<td>RM</td>
<td>V/V</td>
<td>ADX</td>
<td>Unsigned addition of r32 with CF, r/m32 to r32, writes CF.</td>
</tr>
<tr>
<td>ADCX r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 REX.w 0F 38 F6 /r</td>
<td>RM</td>
<td>V/NE</td>
<td>ADX</td>
<td>Unsigned addition of r64 with CF, r/m64 to r64, writes CF.</td>
</tr>
<tr>
<td>ADCX r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

ADCX executes normally either inside or outside a transaction region.

Note: ADCX defines the OF flag differently than the ADD/ADC instructions as defined in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

**Operation**

IF OperandSize is 64-bit

FI;

**Flags Affected**

CF is updated based on result. OF, SF, ZF, AF and PF flags are unmodified.

**Intel C/C++ Compiler Intrinsic Equivalent**

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

**SIMD Floating-Point Exceptions**

None

**Protected Mode Exceptions**

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) For an illegal address in the SS segment.
#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) For an illegal address in the SS segment.

#GP(0) If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) For an illegal address in the SS segment.

#GP(0) If any part of the operand lies outside the effective address space from 0 to FFFFH.

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
### ADD—Add

**Description**

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the CF and OF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Notes:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

---

* **opcode**
  * **Instruction**
  * **Op/En**
  * **64-bit Mode**
  * **Compat/ Leg Mode**
  * **Description**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 ib</td>
<td>ADD AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Add imm8 to AL.</td>
</tr>
<tr>
<td>05 iw</td>
<td>ADD AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Add imm16 to AX.</td>
</tr>
<tr>
<td>05 id</td>
<td>ADD EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Add imm32 to EAX.</td>
</tr>
<tr>
<td>REX.W + 05 id</td>
<td>ADD RAX, imm32</td>
<td>I</td>
<td>N.E.</td>
<td>Add imm32 sign-extended to 64-bits to RAX.</td>
<td></td>
</tr>
<tr>
<td>80 /0 lb</td>
<td>ADD r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add imm8 to r/m8.</td>
</tr>
<tr>
<td>REX + 80 /0 ib</td>
<td>ADD r/m8, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Add sign-extended imm8 to r/m8.</td>
<td></td>
</tr>
<tr>
<td>81 /0 iw</td>
<td>ADD r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add imm16 to r/m16.</td>
</tr>
<tr>
<td>81 /0 id</td>
<td>ADD r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add imm32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /0 id</td>
<td>ADD r/m64, imm32</td>
<td>MI</td>
<td>N.E.</td>
<td>Add imm32 sign-extended to 64-bits to r/m64.</td>
<td></td>
</tr>
<tr>
<td>83 /0 lb</td>
<td>ADD r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add sign-extended imm8 to r/m16.</td>
</tr>
<tr>
<td>83 /0 lb</td>
<td>ADD r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add sign-extended imm8 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 83 /0 lb</td>
<td>ADD r/m64, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Add sign-extended imm8 to r/m64.</td>
<td></td>
</tr>
<tr>
<td>00 /r</td>
<td>ADD r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Add r8 to r/m8.</td>
</tr>
<tr>
<td>REX + 00 /r</td>
<td>ADD r/m8, r8</td>
<td>MR</td>
<td>N.E.</td>
<td>Add r8 to r/m8.</td>
<td></td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Add r16 to r/m16.</td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Add r32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 01 /r</td>
<td>ADD r/m64, r64</td>
<td>MR</td>
<td>N.E.</td>
<td>Add r64 to r/m64.</td>
<td></td>
</tr>
<tr>
<td>02 /r</td>
<td>ADD r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Add r/m8 to r8.</td>
</tr>
<tr>
<td>REX + 02 /r</td>
<td>ADD r8, r/m8</td>
<td>RM</td>
<td>N.E.</td>
<td>Add r/m8 to r8.</td>
<td></td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Add r/m16 to r16.</td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Add r/m32 to r32.</td>
</tr>
<tr>
<td>REX.W + 03 /r</td>
<td>ADD r64, r/m64</td>
<td>RM</td>
<td>N.E.</td>
<td>Add r/m64 to r64.</td>
<td></td>
</tr>
</tbody>
</table>
This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.
In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
DEST ← DEST + SRC;

Flags Affected
The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions
- #GP(0) If the destination is located in a non-writable segment.
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #GP(0) If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions
- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
### ADDPD—Add Packed Double-Precision Floating-Point Values

#### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 58 /r ADDPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed double-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed double-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed double-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 58 /r VADDPD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 58 /r VADDPD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 58 /r VADDPD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Add packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Add two, four or eight packed double-precision floating-point values from the first source operand to the second source operand, and stores the packed double-precision floating-point results in the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
**Operation**

**VADDPD (EVEX encoded versions) when src2 operand is a vector register**

For $(\text{KL, VL}) = (2, 128), (4, 256), (8, 512)$

If $(\text{VL} = 512)$ AND $(\text{EVEX.b} = 1)$

- THEN 
  
  SET_RM(EVEX.RC);

- ELSE 
  
  SET_RM(MXCSR.RM);

FI;

FOR $j \leftarrow 0 \text{ TO KL-1}$

- $i \leftarrow j \times 64$

  IF $k1[j]$ OR *no writemask*

  THEN 
    
    $\text{DEST}[i+63:i] \leftarrow \text{SRC1}[i+63:i] + \text{SRC2}[i+63:i]$

  ELSE 

    IF *merging-masking* ; merging-masking

    THEN *DEST*[i+63:i] remains unchanged*

    ELSE ; zeroing-masking

      $\text{DEST}[i+63:i] \leftarrow 0$

    FI

  FI

ENDFOR

$\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0$

**VADDPD (EVEX encoded versions) when src2 operand is a memory source**

For $(\text{KL, VL}) = (2, 128), (4, 256), (8, 512)$

FOR $j \leftarrow 0 \text{ TO KL-1}$

- $i \leftarrow j \times 64$

  IF $k1[j]$ OR *no writemask*

  THEN 

    IF (EVEX.b = 1)

    THEN 
    
      $\text{DEST}[i+63:i] \leftarrow \text{SRC1}[i+63:i] + \text{SRC2}[63:0]$

    ELSE 

      $\text{DEST}[i+63:i] \leftarrow \text{SRC1}[i+63:i] + \text{SRC2}[i+63:i]$

    FI;

  ELSE 

    IF *merging-masking* ; merging-masking

    THEN *DEST*[i+63:i] remains unchanged*

    ELSE ; zeroing-masking

      $\text{DEST}[i+63:i] \leftarrow 0$

    FI

  FI

ENDFOR

$\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0$

**VADDPD (VEX.256 encoded version)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]$

$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$

$\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] + \text{SRC2}[191:128]$

$\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] + \text{SRC2}[255:192]$

$\text{DEST}[\text{MAXVL-1:256}] \leftarrow 0$

.
VADDPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[MAXVL-1:128] ← 0

ADDPD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] + SRC[63:0]
DEST[127:64] ← DEST[127:64] + SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VADDPD __m512d _mm512_add_pd (__m512d a, __m512d b);
VADDPD __m512d _mm512_mask_add_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VADDPD __m512d _mm512_maskz_add_pd (__mmask8 k, __m512d a, __m512d b);
VADDPD __m256d _mm256_mask_add_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VADDPD __m256d _mm256_maskz_add_pd (__mmask8 k, __m256d a, __m256d b);
VADDPD __m128d _mm_mask_add_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDPD __m128d _mm_maskz_add_pd (__mmask8 k, __m128d a, __m128d b);
VADDPD __m512d _mm512_add_round_pd (__m512d a, __m512d b, int);
VADDPD __m512d _mm512_mask_add_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VADDPD __m512d _mm512_maskz_add_round_pd (__mmask8 k, __m512d a, __m512d b, int);
ADDPD __m256d _mm256_add_pd (__m256d a, __m256d b);
ADDPD __m128d _mm_add_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
ADDPS—Add Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 58 /r ADDPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Add packed single-precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed single-precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed single-precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 58 /r VADDPS xmm1 (k1){x}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 58 /r VADDPS ymm1 (k1){x}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 58 /r VADDPS zmm1 (k1){x}, zmm2, zmm3/m512/m32bcst (er)</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Add packed single-precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Add four, eight or sixteen packed single-precision floating-point values from the first source operand with the second source operand, and stores the packed single-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
**Operation**

VADDPS (EVEX encoded versions) when src2 operand is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VADDPS (EVEX encoded versions) when src2 operand is a memory source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
        ELSE
          DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
**VADDPS (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32] \\
\text{DEST}[95:64] & \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64] \\
\text{DEST}[127:96] & \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96] \\
\text{DEST}[159:128] & \leftarrow \text{SRC1}[159:128] + \text{SRC2}[159:128] \\
\text{DEST}[191:160] & \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160] \\
\text{DEST}[223:192] & \leftarrow \text{SRC1}[223:192] + \text{SRC2}[223:192] \\
\text{DEST}[255:224] & \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224].
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0
\]

**VADDPS (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32] \\
\text{DEST}[95:64] & \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64] \\
\text{DEST}[127:96] & \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0
\end{align*}
\]

**ADDPS (128-bit Legacy SSE version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32] \\
\text{DEST}[95:64] & \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64] \\
\text{DEST}[127:96] & \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & \text{(Unmodified)}
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\begin{align*}
\text{VADDPS } &_{\text{m512}} \ _{\text{mm512}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{m512}} \ a, _{\text{m512}} \ b); \\
\text{VADDPS } &_{\text{m512}} \ _{\text{mm512}} \ _{\text{mask}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{m512}} \ s, _{\text{mmask16}} \ k, _{\text{m512}} \ a, _{\text{m512}} \ b); \\
\text{VADDPS } &_{\text{m512}} \ _{\text{mm512}} \ _{\text{mask2}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{mmask16}} \ k, _{\text{m512}} \ a, _{\text{m512}} \ b); \\
\text{VADDPS } &_{\text{m256}} \ _{\text{mm256}} \ _{\text{mask}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{m256}} \ s, _{\text{mmask8}} \ k, _{\text{m256}} \ a, _{\text{m256}} \ b); \\
\text{VADDPS } &_{\text{m256}} \ _{\text{mm256}} \ _{\text{mask2}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{mmask8}} \ k, _{\text{m256}} \ a, _{\text{m256}} \ b); \\
\text{VADDPS } &_{\text{m128}} \ _{\text{mm128}} \ _{\text{mask}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{mmask8}} \ k, _{\text{m128}} \ a, _{\text{m128}} \ b); \\
\text{VADDPS } &_{\text{m128}} \ _{\text{mm128}} \ _{\text{add}} \ _{\text{round}} \ _{\text{ps}} \ (_{\text{m128}} \ a, _{\text{m128}} \ b, \text{int}); \\
\text{VADDPS } &_{\text{m128}} \ _{\text{mm128}} \ _{\text{mask}} \ _{\text{add}} \ _{\text{round}} \ _{\text{ps}} \ (_{\text{mmask16}} \ k, _{\text{m128}} \ a, _{\text{m128}} \ b, \text{int}); \\
\text{ADDPS } &_{\text{m256}} \ _{\text{mm256}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{m256}} \ a, _{\text{m256}} \ b); \\
\text{ADDPS } &_{\text{m128}} \ _{\text{mm128}} \ _{\text{add}} \ _{\text{ps}} \ (_{\text{m128}} \ a, _{\text{m128}} \ b);
\end{align*}
\]

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
ADDSD—Add Scalar Double-Precision Floating-Point Values

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Adds the low double-precision floating-point values from the second source operand and the first source operand and stores the double-precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VADDSD (EVEX encoded version)
IF (EVEX.b = 1) AND SRC2 *is a register*
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
THEN DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
ELSE
  IF *merging-masking*
  THEN *DEST[63:0] remains unchanged*
  ELSE
    THEN DEST[63:0] ← 0
  FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

VADDSD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

ADDSD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VADDSD __m128d _mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d _mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d _mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
ADDSS—Add Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 58 /r ADDSS xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F:W1G 58 /r VADDSS xmm1,xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 58 /r VADDSS xmm1<a href="z">k1</a>, xmm2, xmm3/m32[er]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Add the low single-precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Adds the low single-precision floating-point values from the second source operand and the first source operand, and stores the double-precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:32) of the corresponding the destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
**Operation**

**VADDSS (EVEX encoded versions)**

IF (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;

IF k1[0] or *no writemask*
  THEN
    DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*  
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;

DEST[MAXVL-1:128] ← 0

**VADDSS DEST, SRC1, SRC2 (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[MAXVL-1:128] ← 0

**ADDSS DEST, SRC (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0] + SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
ADDSUBPD—Packed Double-FP Add/Subtract

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F D0 /r ADDSUBPD xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Add/subtract double-precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG D0 /r VADDSUBPD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add/subtract packed double-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG D0 /r VADDSUBPD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add/subtract packed double-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Adds odd-numbered double-precision floating-point values of the first source operand (second operand) with the corresponding double-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered double-precision floating-point values from the second source operand from the corresponding double-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
**ADDSUBPD—Packed Double-FP Add/Subtract**

### Operation

**ADDSUBPD (128-bit Legacy SSE version)**

- DEST[63:0] ← DEST[63:0] - SRC[63:0]
- DEST[MAXVL-1:128] (Unmodified)

**VADDSUBPD (VEX.128 encoded version)**

- DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
- DEST[MAXVL-1:128] ← 0

**VADDSUBPD (VEX.256 encoded version)**

- DEST[63:0] ← SRC1[63:0] - SRC2[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

- ADDSUBPD: `__m128d _mm_addsub_pd(__m128d a, __m128d b)`
- VADDSUBPD: `__m256d _mm256_addsub_pd(__m256d a, __m256d b)`

### Exceptions

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.
**ADDSUBPS—Packed Single-FP Add/Subtract**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F D0 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Add/subtract single-precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>ADDSUBPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.F2.0F.WIG D0 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add/subtract single-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1.</td>
</tr>
<tr>
<td>VADDSUBPS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.F2.0F.WIG D0 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add / subtract single-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1.</td>
</tr>
<tr>
<td>VADDSUBPS ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating point values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
**Operation**

**ADDSUBPS (128-bit Legacy SSE version)**
- \( \text{DEST}[31:0] \leftarrow \text{DEST}[31:0] - \text{SRC}[31:0] \)
- \( \text{DEST}[95:64] \leftarrow \text{DEST}[95:64] - \text{SRC}[95:64] \)
- \( \text{DEST}[127:96] \leftarrow \text{DEST}[127:96] + \text{SRC}[127:96] \)
- \( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)

**VADDSUBPS (VEX.128 encoded version)**
- \( \text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0] \)
- \( \text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32] \)
- \( \text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64] \)
- \( \text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96] \)
- \( \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \)

**VADDSUBPS (VEX.256 encoded version)**
- \( \text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0] \)
- \( \text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32] \)
- \( \text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64] \)
- \( \text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96] \)
- \( \text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC2}[159:128] \)
- \( \text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160] \)
- \( \text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] - \text{SRC2}[223:192] \)
- \( \text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224] \).

**Intel C/C++ Compiler Intrinsic Equivalent**

- ADDSUBPS: \( \_\_m128 \_\_mm_addsub_ps(\_\_m128 a, \_\_m128 b) \)
- VADDSUBPS: \( \_\_m256 \_\_mm256_addsub_ps(\_\_m256 a, \_\_m256 b) \)

**Exceptions**

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.
**SIMD Floating-Point Exceptions**
Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**
See Exceptions Type 2.
ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 F6 r</td>
<td>RM</td>
<td>V/V</td>
<td>ADX</td>
<td>Unsigned addition of r32 with OF, r/m32 to r32, writes OF.</td>
</tr>
<tr>
<td>ADOX r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 REX.w 0F 38 F6 /r</td>
<td>RM</td>
<td>V/NE</td>
<td>ADX</td>
<td>Unsigned addition of r64 with OF, r/m64 to r64, writes OF.</td>
</tr>
<tr>
<td>ADOX r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands.

The ADOX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we execute an instruction to zero the OF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64-bits.

ADOX executes normally either inside or outside a transaction region.

Note: ADOX defines the CF and OF flags differently than the ADD/ADC instructions as defined in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

### Operation

IF OperandSize is 64-bit

ELSE OF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + OF;
FI;

### Flags Affected

OF is updated based on result. CF, SF, ZF, AF and PF flags are unmodified.

### Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

### SIMD Floating-Point Exceptions

None
Protected Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) For an illegal address in the SS segment.
#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) For an illegal address in the SS segment.
#GP(0) If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) For an illegal address in the SS segment.
#GP(0) If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
AESDEC—Perform One Round of an AES Decryption Flow

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DE /r AESDEC xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AES</td>
<td>Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDECLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

**AESDEC**

STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)

**VAESDEC**

STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC: __m128i _mm_aesdec (__m128i, __m128i)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4.
AESDECLAST—Perform Last Round of an AES Decryption Flow

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DF /r</td>
<td>RM</td>
<td>V/V</td>
<td>AES</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.</td>
</tr>
<tr>
<td>VAESDECLAST xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

AESDECLAST

STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)

VAESDECLAST

STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDECLAST: __m128i _mm_aesdeclast (__m128i, __m128i)
SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4.
AESENC—Perform One Round of an AES Encryption Flow

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DC /r</td>
<td>RM</td>
<td>V/V</td>
<td>AES</td>
<td>Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG DC /r</td>
<td>RVM</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from the xmm3/m128; store the result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM reg (r, w)</td>
<td>ModRM r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs a single round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENC-CLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

**AESENC**

STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)

**VAESENC**

STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENC: __m128i _mm_aesenc (__m128i, __m128i)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4.
**AESENCLAST—Perform Last Round of an AES Encryption Flow**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DD /r</td>
<td>RM</td>
<td>V/V</td>
<td>AES</td>
<td>Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.</td>
</tr>
<tr>
<td>VAESSENCLAST xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128 bit round key from xmm3/m128; store the result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation**

**AESENCLAST**

STATE ← SRC1;  
RoundKey ← SRC2;  
STATE ← ShiftRows( STATE );  
STATE ← SubBytes( STATE );  
DEST[127:0] ← STATE XOR RoundKey;  
DEST[MAXVL-1:128] (Unmodified)

**VAESSENCLAST**

STATE ← SRC1;  
RoundKey ← SRC2;  
STATE ← ShiftRows( STATE );  
STATE ← SubBytes( STATE );  
DEST[127:0] ← STATE XOR RoundKey;  
DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)AESENCLAST: __m128i _mm_aesenclast(__m128i __m128i)
**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
See Exceptions Type 4.
### AESIMC—Perform the AES InvMixColumn Transformation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DB /r AESIMC xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AES</td>
<td>Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.</td>
</tr>
<tr>
<td>VAESIMC xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Perform the InvMixColumns transformation on the source operand and store the result in the destination operand. The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

Note: the AESIMC instruction should be applied to the expanded AES round keys (except for the first and last round key) in order to prepare them for decryption using the "Equivalent Inverse Cipher" (defined in FIPS 197).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

#### Operation

**AESIMC**

DEST[127:0] ← InvMixColumns( SRC );

DEST[MAXVL-1:128] (Unmodified)

**VAESIMC**

DEST[127:0] ← InvMixColumns( SRC );

DEST[MAXVL-1:128] ← 0;

#### Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC: __m128i _mm_aesimc(__m128i)

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.
AESKEYGENASSIST—AES Round Key Generation Assist

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A DF /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AES</td>
<td>Assist in AES round key generation using an 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG DF /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Assist in AES round key generation using 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

AESKEYGENASSIST

X3[31:0] ← SRC[127:96];
X2[31:0] ← SRC[95:64];
X1[31:0] ← SRC[63:32];
X0[31:0] ← SRC[31:0];
RCON[31:0] ← ZeroExtend(imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32] ← RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord( SubWord(X3) ) XOR RCON;
DEST[MAXVL-1:128] (Unmodified)
VAESKEYGENASSIST
X3[31:0] ← SRC [127:96];
X2[31:0] ← SRC [95:64];
X1[31:0] ← SRC [63:32];
X0[31:0] ← SRC [31:0];
RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32 ] ← RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord( SubWord(X3) ) XOR RCON;
DEST[MAXVL-1:128] ← 0;

Intel C/C++ Compiler Intrinsic Equivalent
(V)AESKEYGENASSIST: __m128i _mm_aeskeygenassist (__m128i, const int)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4; additionally
#UD If VEX.vvvv ≠ 1111B.
AND—Logical AND

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg(r, w)</td>
<td>ModRMreg(r, w)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMreg(r, w)</td>
<td>ModRMreg(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMreg(r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow it to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**NOTES:**

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.*
Operation

DEST ← DEST AND SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0) If the destination operand points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
### ANDN — Logical AND NOT

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.LZ.0F38.W0 F2 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>BMI1</td>
<td>Bitwise AND of inverted r32b with r/m32, store result in r32a.</td>
</tr>
<tr>
<td>VEX.NDS.LZ.0F38.W1 F2 /r</td>
<td>RVM</td>
<td>V/NE</td>
<td>BMI1</td>
<td>Bitwise AND of inverted r64b with r/m64, store result in r64a.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

\[
\text{DEST} \leftarrow (\text{NOT SRC1}) \text{ bitwiseAND SRC2};
\]

\[
\text{SF} \leftarrow \text{DEST}[\text{OperandSize} - 1];
\]

\[
\text{ZF} \leftarrow (\text{DEST} = 0);
\]

### Flags Affected

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

### Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally #UD If VEX.W = 1.
ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 54 /r ANDPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical AND of packed double-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical AND of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
**Operation**

**VANDPD (EVEX encoded versions)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 64\)

IF \(k1[j]\) OR *no writemask*

THEN

IF (EVEX.b == 1) AND (SRC2 *is memory*)

THEN

\(\text{DEST}[i+63:j] \leftarrow \text{SRC1}[i+63:j] \text{ BITWISE AND SRC2}[63:0]\)

ELSE

\(\text{DEST}[i+63:j] \leftarrow \text{SRC1}[i+63:j] \text{ BITWISE AND SRC2}[i+63:0]\)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST\([i+63:j]\) remains unchanged*

ELSE ; zeroing-masking

\(\text{DEST}[i+63:j] = 0\)

FI;

ENDIFFOR

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

**VANDPD (VEX.256 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] \text{ BITWISE AND SRC2}[63:0]\)

\(\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] \text{ BITWISE AND SRC2}[127:64]\)

\(\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] \text{ BITWISE AND SRC2}[191:128]\)

\(\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] \text{ BITWISE AND SRC2}[255:192]\)

\(\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0\)

**VANDPD (VEX.128 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] \text{ BITWISE AND SRC2}[63:0]\)

\(\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] \text{ BITWISE AND SRC2}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0\)

**ANDPD (128-bit Legacy SSE version)**

\(\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] \text{ BITWISE AND SRC}[63:0]\)

\(\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] \text{ BITWISE AND SRC}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] \text{(Unmodified)}\)

**Intel C/C++ Compiler Intrinsic Equivalent**

\(\text{VANDPD } \_\_m512d \_\_mm512\_and\_pd (\_\_m512d a, \_\_m512d b);\)

\(\text{VANDPD } \_\_m512d \_\_mm512\_mask\_and\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);\)

**SIMD Floating-Point Exceptions**

None
Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 54 ( r ) ANDPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F 54 ( r ) VANDPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F 54 ( r ) VANDPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.0W 54 ( r ) VANDPS xmm1 ( <a href="r">k1</a> ), xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.0W 54 ( r ) VANDPS ymm1 ( <a href="r">k1</a> ), ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.0W 54 ( r ) VANDPS zmm1 ( <a href="r">k1</a> ), zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical AND of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

VANDPS (EVEX encoded versions)

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \) TO \(KL-1\)
  \(i \leftarrow j \times 32\)
  IF \(k1[j]\) OR *no writemask*
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN
        DEST\([i+63:i]\) \leftarrow SRC1\([i+31:i]\) BITWISE AND SRC2\([31:0]\)
      ELSE
        DEST\([i+31:i]\) \leftarrow SRC1\([i+31:i]\) BITWISE AND SRC2\([i+31:i]\)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST\([i+31:i]\) remains unchanged*
    ELSE ; zeroing-masking
      DEST\([i+31:i]\) \leftarrow 0
    FI;
  FI;
ENDFOR

DEST\([MAXVL-1:VL]\) \leftarrow 0;

VANDPS (VEX.256 encoded version)

DEST\([31:0]\) \leftarrow SRC1\([31:0]\) BITWISE AND SRC2\([31:0]\)
DEST\([63:32]\) \leftarrow SRC1\([63:32]\) BITWISE AND SRC2\([63:32]\)
DEST\([95:64]\) \leftarrow SRC1\([95:64]\) BITWISE AND SRC2\([95:64]\)
DEST\([127:96]\) \leftarrow SRC1\([127:96]\) BITWISE AND SRC2\([127:96]\)
DEST\([159:128]\) \leftarrow SRC1\([159:128]\) BITWISE AND SRC2\([159:128]\)
DEST\([191:160]\) \leftarrow SRC1\([191:160]\) BITWISE AND SRC2\([191:160]\)
DEST\([223:192]\) \leftarrow SRC1\([223:192]\) BITWISE AND SRC2\([223:192]\)
DEST\([255:224]\) \leftarrow SRC1\([255:224]\) BITWISE AND SRC2\([255:224]\)
DEST\([MAXVL-1:256]\) \leftarrow 0;

VANDPS (VEX.128 encoded version)

DEST\([31:0]\) \leftarrow SRC1\([31:0]\) BITWISE AND SRC2\([31:0]\)
DEST\([63:32]\) \leftarrow SRC1\([63:32]\) BITWISE AND SRC2\([63:32]\)
DEST\([95:64]\) \leftarrow SRC1\([95:64]\) BITWISE AND SRC2\([95:64]\)
DEST\([127:96]\) \leftarrow SRC1\([127:96]\) BITWISE AND SRC2\([127:96]\)
DEST\([MAXVL-1:128]\) \leftarrow 0;

ANDPS (128-bit Legacy SSE version)

DEST\([31:0]\) \leftarrow DEST\([31:0]\) BITWISE AND SRC\([31:0]\)
DEST\([63:32]\) \leftarrow DEST\([63:32]\) BITWISE AND SRC\([63:32]\)
DEST\([95:64]\) \leftarrow DEST\([95:64]\) BITWISE AND SRC\([95:64]\)
DEST\([127:96]\) \leftarrow DEST\([127:96]\) BITWISE AND SRC\([127:96]\)
DEST\([MAXVL-1:128]\) (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent
VANDPS __m512_mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512_mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512_mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256_mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256_mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128_mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m256_mm256_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256_mm_and_ps (__m256 a, __m256 b);
ANDPS __m128_mm_and_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
**ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 55 /r ANDNPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:W1 55 /r VANDNPD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:W1 55 /r VANDNPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:W1 55 /r VANDNPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND NOT of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise logical AND NOT of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

VANDNPD (EVEX encoded versions)

\(KL, VL = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0 \) \( \text{TO} \) \(KL-1\)

\(i \leftarrow j \times 64\)

IF \(k1[j] \) OR *no writemask*

IF (EVEX.b == 1) AND (SRC2 *is memory*)

THEN

\(\text{DEST}[i+63:j] \leftarrow (\text{NOT}(\text{SRC1}[i+63:j])) \text{ BITWISE AND} \text{ SRC2}[63:0]\)

ELSE

\(\text{DEST}[i+63:j] \leftarrow (\text{NOT}(\text{SRC1}[i+63:j])) \text{ BITWISE AND} \text{ SRC2}[i+63:j]\)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+63:j] remains unchanged*

ELSE ; zeroing-masking

\(\text{DEST}[i+63:j] = 0\)

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

VANDNPD (VEX.256 encoded version)

\(\text{DEST}[63:0] \leftarrow (\text{NOT}(\text{SRC1}[63:0])) \text{ BITWISE AND} \text{ SRC2}[63:0]\)

\(\text{DEST}[127:64] \leftarrow (\text{NOT}(\text{SRC1}[127:64])) \text{ BITWISE AND} \text{ SRC2}[127:64]\)

\(\text{DEST}[191:128] \leftarrow (\text{NOT}(\text{SRC1}[191:128])) \text{ BITWISE AND} \text{ SRC2}[191:128]\)

\(\text{DEST}[255:192] \leftarrow (\text{NOT}(\text{SRC1}[255:192])) \text{ BITWISE AND} \text{ SRC2}[255:192]\)

\(\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0\)

VANDNPD (VEX.128 encoded version)

\(\text{DEST}[63:0] \leftarrow (\text{NOT}(\text{SRC1}[63:0])) \text{ BITWISE AND} \text{ SRC2}[63:0]\)

\(\text{DEST}[127:64] \leftarrow (\text{NOT}(\text{SRC1}[127:64])) \text{ BITWISE AND} \text{ SRC2}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0\)

ANDNPD (128-bit Legacy SSE version)

\(\text{DEST}[63:0] \leftarrow (\text{NOT}(\text{DEST}[63:0])) \text{ BITWISE AND} \text{ SRC}[63:0]\)

\(\text{DEST}[127:64] \leftarrow (\text{NOT}(\text{DEST}[127:64])) \text{ BITWISE AND} \text{ SRC}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPD __m512d _mm512__andnot_pd (__m512d a, __m512d b);
VANDNPD __m512d _mm512__mask__andnot_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDNPD __m512d _mm512__maskz__andnot_pd (__mmask8 k, __m512d a, __m512d b);
VANDNPD __m256d _mm256__mask__andnot_pd (__mmask8 k, __m256d a, __m256d b);
VANDNPD __m256d _mm256__maskz__andnot_pd (__mmask8 k, __m256d a, __m256d b);
VANDNPD __m128d _mm128__mask__andnot_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDNPD __m128d _mm128__maskz__andnot_pd (__mmask8 k, __m128d a, __m128d b);
VANDNPD __m256d _mm256__andnot_pd (__m256d a, __m256d b);
ANDNPD __m128d _mm128__andnot_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None
Other Exceptions
VEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 55 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F 55 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F 55 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F:W0 55 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F:W0 55 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F:W0 55 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv ModRM:r/m (r)</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv ModRM:r/m (r)</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise logical AND NOT of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

**VANDNPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        IF (EVEX.b == 1) AND (SRC2 *is memory*)
            THEN
                DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]
            ELSE
                DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR

**VANDNPS (VEX.256 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[159:128] ← (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]
DEST[MAXVL-1:256] ← 0

**VANDNPS (VEX.128 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] ← 0

**ANDNPS (128-bit Legacy SSE version)**

DEST[31:0] ← (NOT(DST[31:0])) BITWISE AND SRC[31:0]
DEST[95:64] ← (NOT(DST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] ← (NOT(DST[127:96])) BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
**Intel C/C++ Compiler Intrinsic Equivalent**

VANDNPS __m512 _mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 _mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 _mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 _mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 _mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 _mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 _mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS _mm256_andnot_ps (__m256 a, __m256 b);
VANDNPS _mm128_andnot_ps (__m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
ARPL—Adjust RPL Field of Segment Selector

**Description**

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program’s code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then ensures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program’s code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode. See "Checking Caller Access Privileges" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information about the use of this instruction.

**Operation**

IF 64-BIT MODE
  THEN
  See MOVSXD;
ELSE
  IF DEST[RPL] < SRC[RPL]
    THEN
      ZF ← 1;
      DEST[RPL] ← SRC[RPL];
    ELSE
      ZF ← 0;
    FI;
  FI;

**Flags Affected**

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, it is set to 0.

---

**Opcode Instruction Op/En**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>ModRM/r/m (w)</td>
<td>ModRM/reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Adjust RPL of r/m16 to not less than RPL of r16.
Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The ARPL instruction is not recognized in real-address mode.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD The ARPL instruction is not recognized in virtual-8086 mode.
If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Not applicable.
BLENDPD — Blend Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 0D /r ib BLENDPD xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select packed DP-FP values from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.WIG OD /r ib VBLENDPD xmm1, xmm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed double-precision floating-point Values from xmm2 and xmm3/m128 from mask in imm8 and store the values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F3A.WIG OD /r ib VBLENDPD ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed double-precision floating-point Values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8[3:0]</td>
</tr>
</tbody>
</table>

**Description**

Double-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [3:0] determine whether the corresponding double-precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is “1”, then the double-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Operation**

**BLENDPD (128-bit Legacy SSE version)**

IF (IMM8[0] = 0) THEN DEST[63:0] ← DEST[63:0]
ELSE DEST[63:0] ← SRC[63:0] FI

IF (IMM8[1] = 0) THEN DEST[127:64] ← DEST[127:64]
ELSE DEST[127:64] ← SRC[127:64] FI

DEST[MAXVL-1:128] ← Unmodified

**VBLENDPD (VEX.128 encoded version)**

IF (IMM8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
ELSE DEST[63:0] ← SRC2[63:0] FI

IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
ELSE DEST[127:64] ← SRC2[127:64] FI

DEST[MAXVL-1:128] ← 0
VBLENDPD (VEX.256 encoded version)
IF (IMM8[0] = 0) THEN DEST[63:0] <- SRC1[63:0]
ELSE DEST[63:0] <- SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] <- SRC1[127:64]
ELSE DEST[127:64] <- SRC2[127:64] FI

Intel C/C++ Compiler Intrinsic Equivalent
BLENDPD: __m128d __mm_blend_pd (__m128d v1, __m128d v2, const int mask);
VBLENDPD: __m256d __mm256_blend_pd (__m256d a, __m256d b, const int mask);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4.
BEXTR — Bit Field Extract

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 -bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.LZ.0F38.W0 F7 / r BEXTR r32a, r/m32, r32b</td>
<td>RMV</td>
<td>V/V</td>
<td>BM1</td>
<td>Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.</td>
</tr>
<tr>
<td>VEX.NDS.LZ.0F38.W1 F7 / r BEXTR r64a, r/m64, r64b</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BM1</td>
<td>Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>VEX.vvvv (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the second source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize - 1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

START ← SRC2[7:0];
LEN ← SRC2[15:8];
TEMP ← ZERO_EXTEND_TO_512 (SRC1);
DEST ← ZERO_EXTEND(TEMP[START+LEN -1: START]);
ZF ← (DEST = 0);

**Flags Affected**

ZF is updated based on the result. AF, SF, and PF are undefined. All other flags are cleared.

**Intel C/C++ Compiler Intrinsic Equivalent**

BEXTR: unsigned __int32 _bextr_u32(unsigned __int32 src, unsigned __int32 start, unsigned __int32 len);
BEXTR: unsigned __int64 _bextr_u64(unsigned __int64 src, unsigned __int32 start, unsigned __int32 len);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD If VEX.W = 1.
BLENDPS — Blend Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 0C /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select packed single precision floating-point values from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.WIG 0C /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed single-precision floating-point values from xmm2 and xmm3/m128 from mask in imm8 and store the values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F3A.WIG 0C /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed single-precision floating-point values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

Packed single-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [7:0] determine whether the corresponding single precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is “1”, then the single-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

**BLENDPS (128-bit Legacy SSE version)**

\[
\begin{align*}
\text{IF (IMM8[0] = 0) THEN DEST}[31:0] & \leftarrow \text{DEST}[31:0] \\
\text{ELSE DEST}[31:0] & \leftarrow \text{SRC}[31:0] \\
\text{IF (IMM8[1] = 0) THEN DEST}[63:32] & \leftarrow \text{DEST}[63:32] \\
\text{ELSE DEST}[63:32] & \leftarrow \text{SRC}[63:32] \\
\text{IF (IMM8[2] = 0) THEN DEST}[95:64] & \leftarrow \text{DEST}[95:64] \\
\text{ELSE DEST}[95:64] & \leftarrow \text{SRC}[95:64] \\
\text{IF (IMM8[3] = 0) THEN DEST}[127:96] & \leftarrow \text{DEST}[127:96] \\
\text{ELSE DEST}[127:96] & \leftarrow \text{SRC}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & \text{(Unmodified)}
\end{align*}
\]
VBLENDPS (VEX.128 encoded version)
IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
ELSE DEST[31:0] ← SRC2[31:0] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
ELSE DEST[95:64] ← SRC2[95:64] FI
DEST[MAXVL-1:128] ← 0

VBLENDPS (VEX.256 encoded version)
IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
ELSE DEST[31:0] ← SRC2[31:0] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
ELSE DEST[95:64] ← SRC2[95:64] FI

Intel C/C++ Compiler Intrinsic Equivalent
BLENDPS: __m128 _mm_blend_ps (__m128 v1, __m128 v2, const int mask);
VBLENDPS: __m256 _mm256_blend_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4.
BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3B 15 /r</td>
<td>RM0</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select packed DP FP values from xmm1 and xmm2/m128 from mask specified in XMM0 and store the values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.W0 4B /r /is4</td>
<td>RVMR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally copy double-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the mask operand, xmm4.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 4B /r /is4</td>
<td>RVMR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally copy double-precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the mask operand, ymm4.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>RM0</td>
</tr>
<tr>
<td>RVMR</td>
</tr>
</tbody>
</table>

**Description**

Conditionally copy each quadword data element of double-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each quadword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding quadword element in the second source operand, if a mask bit is “1”; or
- the corresponding quadword element in the first source operand, if a mask bit is “0”

The register assignment of the implicit mask operand for BLENDVPD is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPD permits the mask to be any XMM or YMM register. In contrast, BLENDVPD treats XMM0 implicitly as the mask and do not support non-destructive destination operation.
**Operation**

**BLENDVPD (128-bit Legacy SSE version)**

MASK ← XMM0
IF (MASK[63] = 0) THEN DEST[63:0] ← DEST[63:0]
ELSE DEST[63:0] ← SRC[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← DEST[127:64]
ELSE DEST[127:64] ← SRC[127:64] FI
DEST[MAXVL-1:128] (Unmodified)

**VBLENDVPD (VEX.128 encoded version)**

MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
ELSE DEST[63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
ELSE DEST[127:64] ← SRC2[127:64] FI
DEST[MAXVL-1:128] ← 0

**VBLENDVPD (VEX.256 encoded version)**

MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
ELSE DEST[63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
ELSE DEST[127:64] ← SRC2[127:64] FI
IF (MASK[255] = 0) THEN DEST[255:192] ← SRC1[255:192]

**Intel C/C++ Compiler Intrinsic Equivalent**

BLENDVPD: _m128d _mm_blendv_pd(_m128d v1, _m128d v2, _m128d v3);
VBLENDVPD: _m128 _mm_blendv_pd (_m128d a, _m128d b, _m128d mask);
VBLENDVPD: _m256 _mm256_blendv_pd (_m256d a, _m256d b, _m256d mask);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.W = 1.
BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values

**Opcode/ Instruction** | **Op/ En** | **64/32-bit Mode** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
66 0F 38 14 /r | RMO | V/V | SSE4_1 | Select packed single precision floating-point values from xmm1 and xmm2/m128 from mask specified in XMM0 and store the values into xmm1.
VEX.NDS.128.66.0F3A.W0 4A /r /is4 | RVMR | V/V | AVX | Conditionally copy single-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the specified mask operand, xmm4.
VEX.NDS.256.66.0F3A.W0 4A /r /is4 | RVMR | V/V | AVX | Conditionally copy single-precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the specified mask register, ymm4.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
--- | --- | --- | --- | ---
RM0 | ModRM:reg (r, w) | ModRM:r/m (r) | implicit XMM0 | NA |
RVMR | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8[7:4] |

**Description**

Conditionally copy each dword data element of single-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each dword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding dword element in the second source operand, if a mask bit is “1”; or
- the corresponding dword element in the first source operand, if a mask bit is “0”

The register assignment of the implicit mask operand for BLENDVPS is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source operand, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source operand, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPS permits the mask to be any XMM or YMM register. In contrast, BLENDVPS treats XMM0 implicitly as the mask and do not support non-destructive destination operation.
Operation

BLENDVPS (128-bit Legacy SSE version)

MASK ← XMM0
IF (MASK[31] = 0) THEN DEST[31:0] ← DEST[31:0]
ELSE DEST[31:0] ← SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← DEST[63:32]
IF (MASK[95] = 0) THEN DEST[95:64] ← DEST[95:64]
ELSE DEST[95:64] ← SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← DEST[127:96]
DEST[MAXVL-1:128] ← 0

VBLENDVPS (VEX.128 encoded version)

MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
ELSE DEST[31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
ELSE DEST[95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
DEST[MAXVL-1:128] ← 0

VBLENDVPS (VEX.256 encoded version)

MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
ELSE DEST[31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
ELSE DEST[95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
IF (MASK[159] = 0) THEN DEST[159:128] ← SRC1[159:128]
IF (MASK[255] = 0) THEN DEST[255:224] ← SRC1[255:224]

Intel C/C++ Compiler Intrinsic Equivalent

BLENDVPS: __m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3);
VBLENDVPS: __m128 _mm_blendv_ps (__m128 a, __m128 b, __m128 mask);
VBLENDVPS: __m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);

SIMD Floating-Point Exceptions

None
Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.
BLSI — Extract Lowest Set Isolated Bit

**Description**
Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**
\[\text{temp} \leftarrow (-\text{SRC}) \text{ bitwiseAND} \ (\text{SRC}); \]
\[\text{SF} \leftarrow \text{temp[OperandSize -1]}; \]
\[\text{ZF} \leftarrow (\text{temp} = 0); \]
\[\text{IF SRC} = 0 \]
\[\quad \text{CF} \leftarrow 0; \]
\[\text{ELSE} \]
\[\quad \text{CF} \leftarrow 1; \]
\[\text{FI} \]
\[\text{DEST} \leftarrow \text{temp}; \]

**Flags Affected**
ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**
BLSI: \[\text{unsigned \_int32 \_blsi\textunderscore u32(unsigned \_int32 \text{src});} \]
BLSI: \[\text{unsigned \_int64 \_blsi\textunderscore u64(unsigned \_int64 \text{src});} \]

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally #UD if VEX.W = 1.
## BLSMSK — Get Mask Up to Lowest Set Bit

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDD.LZ.0F38.W0 F3 /2 BLSMSK r32, r/m32</td>
<td>VM</td>
<td>V/V</td>
<td>BM1</td>
<td>Set all lower bits in r32 to “1” starting from bit 0 to lowest set bit in r/m32.</td>
</tr>
<tr>
<td>VEX.NDD.LZ.0F38.W1 F3 /2 BLSMSK r64, r/m64</td>
<td>VM</td>
<td>V/N.E.</td>
<td>BM1</td>
<td>Set all lower bits in r64 to “1” starting from bit 0 to lowest set bit in r/m64.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>VEX.vvvv (w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Sets all the lower bits of the destination operand to “1” up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

\[
\text{temp} \leftarrow (\text{SRC}-1) \oplus \text{SRC}; \\
\text{SF} \leftarrow \text{temp}[\text{OperandSize}-1]; \\
\text{ZF} \leftarrow 0; \\
\text{IF SRC} = 0 \\
\quad \text{CF} \leftarrow 1; \\
\text{ELSE} \\
\quad \text{CF} \leftarrow 0; \\
\text{FI} \\
\text{DEST} \leftarrow \text{temp};
\]

### Flags Affected

SF is updated based on the result. CF is set if the source if zero. ZF and OF flags are cleared. AF and PF flag are undefined.

### Intel C/C++ Compiler Intrinsic Equivalent

BLSMSK:    unsigned __int32 _blsmsk_u32(unsigned __int32 src);
BLSMSK:    unsigned __int64 _blsmsk_u64(unsigned __int64 src);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally #UD if VEX.W = 1.
**BLSR — Reset Lowest Set Bit**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDD.LZ.0F38.W0 F3 /I</td>
<td>VM</td>
<td>V/V</td>
<td>BMI1</td>
<td>Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.</td>
</tr>
<tr>
<td>VEX.NDD.LZ.0F38.W1 F3 /I</td>
<td>VM</td>
<td>V/N.E.</td>
<td>BMI1</td>
<td>Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>VEX.vvvv (w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

\[
\text{temp} \leftarrow (\text{SRC}-1) \text{ bitwiseAND } (\text{SRC}); \\
\text{SF} \leftarrow \text{temp}[\text{OperandSize}-1]; \\
\text{ZF} \leftarrow (\text{temp} = 0); \\
\text{IF SRC} = 0 \\
\quad \text{CF} \leftarrow 1; \\
\text{ELSE} \\
\quad \text{CF} \leftarrow 0; \\
\text{FI} \\
\text{DEST} \leftarrow \text{temp};
\]

**Flags Affected**

ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**

- unsigned __int32 __blsr_u32(unsigned __int32 src);
- unsigned __int64 __blsr_u64(unsigned __int64 src);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally #UD If VEX.W = 1.
BNDCL—Check Lower Bound

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 1A /r</td>
<td>RM</td>
<td>NE/V</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m32 is lower than the lower bound in bnd.LB.</td>
</tr>
<tr>
<td>BNDCL bnd, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F 1A /r</td>
<td>RM</td>
<td>V/NE</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m64 is lower than the lower bound in bnd.LB.</td>
</tr>
<tr>
<td>BNDCL bnd, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compare the address in the second operand with the lower bound in bnd. The second operand can be either a register or memory operand. If the address is lower than the lower bound in bnd.LB, it will set BNDSTATUS to 01H and signal a #BR exception.

This instruction does not cause any memory access, and does not read or write any flags.

**Operation**

**BNDCL BND, reg**

IF reg < BND.LB Then
   BNDSTATUS &lt; 01H;
   #BR;
   Fl;

**BNDCL BND, mem**

TEMP &lt; LEA(mem);
IF TEMP < BND.LB Then
   BNDSTATUS &lt; 01H;
   #BR;
   Fl;

**Intel C/C++ Compiler Intrinsic Equivalent**

BNDCL void _bnd_chk_ptr_lbounds(const void *q)

**Flags Affected**

None

**Protected Mode Exceptions**

#BR If lower bound check fails.
#UD If the LOCK prefix is used.
   If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 67H prefix is not used and CS.D=0.
   If 67H prefix is used and CS.D=1.
Real-Address Mode Exceptions
#BR If lower bound check fails.
#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.

Virtual-8086 Mode Exceptions
#BR If lower bound check fails.
#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
Same exceptions as in protected mode.
**BNDCU/BNDCN—Check Upper Bound**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 1A r</td>
<td>RM</td>
<td>NE/V</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnd.UB in 1’s complement form).</td>
</tr>
<tr>
<td>BNDCU bnd, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 0F 1A r</td>
<td>RM</td>
<td>V/NE</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnd.UB in 1’s complement form).</td>
</tr>
<tr>
<td>BNDCU bnd, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 0F 1B r</td>
<td>RM</td>
<td>NE/V</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnd.UB not in 1’s complement form).</td>
</tr>
<tr>
<td>BNDCN bnd, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 0F 1B r</td>
<td>RM</td>
<td>V/NE</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnd.UB not in 1’s complement form).</td>
</tr>
<tr>
<td>BNDCN bnd, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BNDCU perform 1’s complement operation on the upper bound of bnd first before proceeding with address comparison. BNDCN perform address comparison directly using the upper bound in bnd that is already reverted out of 1’s complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA

**Operation**

**BNDCU BND, reg**

IF reg > NOT(BND.UB) Then
   BNDSTATUS ← 01H;
   #BR;
   Fi;

**BNDCU BND, mem**

TEMP ← LEA(mem);
IF TEMP > NOT(BND.UB) Then
   BNDSTATUS ← 01H;
   #BR;
   Fi;

**BNDCN BND, reg**

IF reg > BND.UB Then
   BNDSTATUS ← 01H;
   #BR;
   Fi;

**BNDCN BND, mem**

TEMP ← LEA(mem);
IF TEMP > NOT(BND.UB) Then
   BNDSTATUS ← 01H;
   #BR;
   Fi;
BNDCN BND, mem
TEMP ← LEA(mem);
IF TEMP > BND.UB Then
    BNDSTATUS ← 01H;
    #BR;
    FI;

Intel C/C++ Compiler Intrinsic Equivalent
BNDCU .void _bnd_chk_ptr_ubounds(const void *q)

Flags Affected
None

Protected Mode Exceptions
#BR If upper bound check fails.
#UD If the LOCK prefix is used.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 67H prefix is not used and CS.D=0.
    If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions
#BR If upper bound check fails.
#UD If the LOCK prefix is used.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 16-bit addressing is used.

Virtual-8086 Mode Exceptions
#BR If upper bound check fails.
#UD If the LOCK prefix is used.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 16-bit addressing is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
Same exceptions as in protected mode.
BNDLDX—Load Extended Bounds Using Address Translation

**Opcode/Instruction** | **Op/En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
NP 0F 1A /r | RM | V/V | MPX | Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 |
--- | --- | --- | --- |
RM | ModRM:reg (w) | SIB.base (r): Address of pointer SIB.index(r) | NA |

**Description**

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

**Operation**

base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;

**Outside 64-bit mode**

A_BT[31:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
  BNDSTATUS ← A_BDE | 02H;
  #BR;
  FI;
Temp_lb[31:0] ← LoadFrom(A_BTE);
Temp_ub[31:0] ← LoadFrom(A_BTE + 4);
Temp_ptr[31:0] ← LoadFrom(A_BTE + 8);
IF Temp_ptr equal ptr_value Then
  BND.LB ← Temp_lb;
  BND.UB ← Temp_ub;
  #BR;
  FI;
ELSE
   BND.LB ← 0;
   BND.UB ← 0;
FI;

In 64-bit mode
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
   BNDSTATUS ← A_BDE | 02H;
   #BR;
FI;
Temp_lb[63:0] ← LoadFrom(A_BTE);
Temp_ub[63:0] ← LoadFrom(A_BTE + 8);
Temp_ptr[63:0] ← LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
   BND.LB ← Temp_lb;
   BND.UB ← Temp_ub;
ELSE
   BND.LB ← 0;
   BND.UB ← 0;
FI;

Intel C/C++ Compiler Intrinsic Equivalent
BNLDX: Generated by compiler as needed.

Flags Affected
None

Protected Mode Exceptions
#BR If the bound directory entry is invalid.
#UD If the LOCK prefix is used.
   If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 67H prefix is not used and CS.D=0.
   If 67H prefix is used and CS.D=1.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.
   If DS register contains a NULL segment selector.
#PF(fault code) If a page fault occurs.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
   If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 16-bit addressing is used.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.

¹. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
   If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 16-bit addressing is used.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code) If a page fault occurs.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR If the bound directory entry is invalid.
#UD If ModRM is RIP relative.
   If the LOCK prefix is used.
   If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0) If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code) If a page fault occurs.
**BNDMK—Make Bounds**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 1B /r</td>
<td>RM</td>
<td>NE/V</td>
<td>MPX</td>
<td>Make lower and upper bounds from m32 and store them in bnd.</td>
</tr>
<tr>
<td>BNDMK bnd, m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F 1B /r</td>
<td>RM</td>
<td>V/NE</td>
<td>MPX</td>
<td>Make lower and upper bounds from m64 and store them in bnd.</td>
</tr>
<tr>
<td>BNDMK bnd, m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

The instruction causes an invalid-opcode exception (#UD) if executed in 64-bit mode with RIP-relative addressing.

### Operation

\[
\text{BND.LB} \leftarrow \text{SRCMEM.base};
\]

\[
\begin{align*}
\text{IF 64-bit mode Then} & \quad \text{BND.UB} \leftarrow \text{NOT}(\text{LEA.64_bits}(\text{SRCMEM})); \\
\text{ELSE} & \quad \text{BND.UB} \leftarrow \text{Zero_Extend.64_bits(} \text{NOT(} \text{LEA.32_bits(} \text{SRCMEM} \text{)}) \text{)};
\end{align*}
\]

### Intel C/C++ Compiler Intrinsic Equivalent

\[
\text{BNDMK void } \_\text{bnd_set_ptr_bounds(} \text{const void } * q, \text{size_t } \text{size});
\]

### Flags Affected

None

### Protected Mode Exceptions

- **#UD**  
  - If the LOCK prefix is used.  
  - If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.  
  - If 67H prefix is not used and CS.D=0.  
  - If 67H prefix is used and CS.D=1.

### Real-Address Mode Exceptions

- **#UD**  
  - If the LOCK prefix is used.  
  - If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.  
  - If 16-bit addressing is used.
Virtual-8086 Mode Exceptions

#UD  If the LOCK prefix is used.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD  If the LOCK prefix is used.
    If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
    If RIP-relative addressing is used.

#SS(0)  If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.

Same exceptions as in protected mode.
BNDMOV—Move Bounds

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 1A /r</td>
<td>RM</td>
<td>NE/V</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2/m64 to bound register bnd1.</td>
</tr>
<tr>
<td>BNDMOV bnd1, bnd2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 1A /r</td>
<td>RM</td>
<td>V/NE</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2/m128 to bound register bnd1.</td>
</tr>
<tr>
<td>BNDMOV bnd1, bnd2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 1B /r</td>
<td>MR</td>
<td>NE/V</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2 to bnd1/m64.</td>
</tr>
<tr>
<td>BNDMOV bnd1/m64, bnd2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 1B /r</td>
<td>MR</td>
<td>V/NE</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2 to bound register bnd1/m128.</td>
</tr>
<tr>
<td>BNDMOV bnd1/m128, bnd2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

BNDMOV moves a pair of lower and upper bound values from the source operand (the second operand) to the destination (the first operand). Each operation is 128-bit move. The exceptions are same as the MOV instruction. The memory format for loading/store bounds in 64-bit mode is shown in Figure 3-5.

![Figure 3-5. Memory Layout of BNDMOV to/from Memory](image)

This instruction does not change flags.

Operation

BNDMOV register to register

DEST.LB ← SRC.LB;
DEST.UB ← SRC.UB;
BNDMOV from memory
IF 64-bit mode THEN
    DEST.LB ← LOAD_QWORD(SRC);
    DEST.UB ← LOAD_QWORD(SRC+8);
ELSE
    DEST.LB ← LOAD_DWORD_ZERO_EXT(SRC);
    DEST.UB ← LOAD_DWORD_ZERO_EXT(SRC+4);
FI;

BNDMOV to memory
IF 64-bit mode THEN
    DEST[63:0] ← SRC.LB;
    DEST[127:64] ← SRC.UB;
ELSE
    DEST[31:0] ← SRC.LB;
    DEST[63:32] ← SRC.UB;
FI;

Intel C/C++ Compiler Intrinsic Equivalent
BNDMOV void * _bnd_copy_ptr_bounds(const void *q, const void *r)

Flags Affected
None

Protected Mode Exceptions
#UD If the LOCK prefix is used but the destination is not a memory operand.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 67H prefix is not used and CS.D=0.
    If 67H prefix is used and CS.D=1.
#SS(0) If the memory operand effective address is outside the SS segment limit.
#GP(0) If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the destination operand points to a non-writable segment
    If the DS, ES, FS, or GS segment register contains a NULL segment selector.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code) If a page fault occurs.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used but the destination is not a memory operand.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 16-bit addressing is used.
#GP(0) If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If the memory operand effective address is outside the SS segment limit.
Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used but the destination is not a memory operand.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.

#GP(0) If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If the memory operand effective address is outside the SS segment limit.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code) If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used but the destination is not a memory operand.
If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

#SS(0) If the memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code) If a page fault occurs.
**BNDSTX—Store Extended Bounds Using Address Translation**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 1B /r BNDSTX mib, bnd</td>
<td>MR</td>
<td>V/V</td>
<td>MPX</td>
<td>Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>SIB.base (r): Address of pointer SIB.index(r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

**Operation**

base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;

**Outside 64-bit mode**

A_BT[31:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
  BNDSTATUS ← A_BDE | 02H;
  #BR;
FI;
A_DEST[8][31:0] ← ptr_value;
A_DEST[0][31:0] ← BND.LB;
A_DEST[4][31:0] ← BND.UB;
In 64-bit mode
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[16][63:0] ← ptr_value;
A_DEST[0][63:0] ← BND.LB;
A_DEST[8][63:0] ← BND.UB;

Intel C/C++ Compiler Intrinsic Equivalent
BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);

Flags Affected
None

Protected Mode Exceptions
#BR If the bound directory entry is invalid.
#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 67H prefix is not used and CS.D=0.
If 67H prefix is used and CS.D=1.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.
If DS register contains a NULL segment selector.
If the destination operand points to a non-writable segment
#PF(fault code) If a page fault occurs.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code) If a page fault occurs.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

---

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
64-Bit Mode Exceptions

#BR If the bound directory entry is invalid.

#UD If ModRM is RIP relative.
If the LOCK prefix is used.
If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

#GP(0) If the memory address (A_BDE or A_BTE) is in a non-canonical form.
If the destination operand points to a non-writable segment

#PF(fault code) If a page fault occurs.
BOUND—Check Array Index Against Bounds

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Comp/M Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>62 Ir</td>
<td>BOUND r16, m16&amp;16</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Check if r16 (array index) is within bounds specified by m16&amp;16.</td>
</tr>
<tr>
<td>62 Ir</td>
<td>BOUND r32, m32&amp;32</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Check if r32 (array index) is within bounds specified by m32&amp;32.</td>
</tr>
</tbody>
</table>

**Description**

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

**Operation**

If 64bit Mode

THEN

#UD;

ELSE

IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound) THEN

(* Below lower bound or above upper bound *)

IF <equation for PL enabled> THEN BNDSTATUS ← 0

#BR;

FI;

FI;

**Flags Affected**

None.
Protected Mode Exceptions

#BR If the bounds test fails.
#UD If second operand is not a memory location.
If the LOCK prefix is used.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#BR If the bounds test fails.
#UD If second operand is not a memory location.
If the LOCK prefix is used.

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#BR If the bounds test fails.
#UD If second operand is not a memory location.
If the LOCK prefix is used.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.
BSF—Bit Scan Forward

Instruction Operand Encoding

### Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← 0;
        WHILE Bit(SRC, temp) = 0
            temp ← temp + 1;
        OD;
        DEST ← temp;
FI;

### Flags Affected

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF flags are undefined.

### Protected Mode Exceptions

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.

---

**BSF r16, r/m16**

**BSF r32, r/m32**

**BSF r64, r/m64**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Op/En: Operands 1 and 2.
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
BSR—Bit Scan Reverse

**Description**
Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF SRC = 0
THEN
    ZF ← 1;
    DEST is undefined;
ELSE
    ZF ← 0;
    temp ← OperandSize - 1;
    WHILE Bit(SRC, temp) = 0
    DO
        temp ← temp - 1;
    OD;
    DEST ← temp;
FI;

**Flags Affected**
The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F BD /r</td>
<td>BSR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Bit scan reverse on r/m16.</td>
</tr>
<tr>
<td>0F BD /r</td>
<td>BSR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Bit scan reverse on r/m32.</td>
</tr>
<tr>
<td>REX.W + 0F BD /r</td>
<td>BSR r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Bit scan reverse on r/m64.</td>
</tr>
</tbody>
</table>
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
BSWAP—Byte Swap

Description
Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Legacy Compatibility
The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

Operation
TEMP ← DEST
IF 64-bit mode AND OperandSize = 64 THEN
    DEST[7:0] ← TEMP[63:56];
    DEST[15:8] ← TEMP[55:48];
    DEST[23:16] ← TEMP[47:40];
    DEST[39:32] ← TEMP[31:24];
    DEST[47:40] ← TEMP[23:16];
    DEST[55:48] ← TEMP[15:8];
    DEST[63:56] ← TEMP[7:0];
ELSE
    DEST[7:0] ← TEMP[31:24];
    DEST[15:8] ← TEMP[23:16];
    DEST[23:16] ← TEMP[15:8];
    DEST[31:24] ← TEMP[7:0];
FI;

Flags Affected
None.

Exceptions (All Operating Modes)
#UD If the LOCK prefix is used.
BT—Bit Test

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Comp/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F A3 /r</td>
<td>BT r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>0F A3 /r</td>
<td>BT r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>REX.W + 0F A3 /r</td>
<td>BT r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>0F BA /4 ib</td>
<td>BT r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>0F BA /4 ib</td>
<td>BT r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>REX.W + 0F BA /4 ib</td>
<td>BT r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Store selected bit in CF flag.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: Bit(BitBase, BitOffset) on page 3-12.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the higher order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

\[
\text{Effective Address} + (4 \times \text{(BitOffset DIV 32)})
\]

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

\[
\text{Effective Address} + (2 \times \text{(BitOffset DIV 16)})
\]

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

\[
\text{CF} \leftarrow \text{Bit(BitBase, BitOffset)};
\]
Flags Affected
The CF flag contains the value of the selected bit. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**BTC—Bit Test and Complement**

**Description**

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.

- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-12.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[
\begin{align*}
CF & \leftarrow \text{Bit(BitBase, BitOffset)}; \\
\text{Bit(BitBase, BitOffset)} & \leftarrow \text{NOT Bit(BitBase, BitOffset)};
\end{align*}
\]

**Flags Affected**

The CF flag contains the value of the selected bit before it is complemented. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.
Protected Mode Exceptions
#GP(0)  If the destination operand points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions
#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS  If a memory operand effective address is outside the SS segment limit.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used but the destination is not a memory operand.
BTR—Bit Test and Reset

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: Bit(BitBase, BitOffset) on page 3-12.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

\[
\begin{align*}
\text{CF} & \leftarrow \text{Bit(BitBase, BitOffset)}; \\
\text{Bit(BitBase, BitOffset)} & \leftarrow 0;
\end{align*}
\]

### Flags Affected

The CF flag contains the value of the selected bit before it is cleared. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.
Protected Mode Exceptions
#GP(0) If the destination operand points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
BTS—Bit Test and Set

Description
Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: Bit(BitBase, BitOffset) on page 3-12.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
\[
\begin{align*}
\text{CF} & \leftarrow \text{Bit(BitBase, BitOffset)}; \\
\text{Bit(BitBase, BitOffset)} & \leftarrow 1;
\end{align*}
\]

Flags Affected
The CF flag contains the value of the selected bit before it is set. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.
Protected Mode Exceptions

#GP(0)  If the destination operand points to a non-writable segment.
        If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
        If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS  If a memory operand effective address is outside the SS segment limit.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used but the destination is not a memory operand.
BZHI — Zero High Bits Starting with Specified Bit Position

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.LZ.0F38.W0 F5 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Zero bits in r/m32 starting with the position in r32b, write result to r32a.</td>
</tr>
<tr>
<td>VEX.NDS.LZ.0F38.W1 F5 /r</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Zero bits in r/m64 starting with the position in r64b, write result to r64a.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>VEX.vvvv (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize -1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize -1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

\[
\begin{align*}
N & \leftarrow \text{SRC2}[7:0] \\
\text{DEST} & \leftarrow \text{SRC1} \\
\text{IF (N < OperandSize)} & \\
\quad \text{DEST}[\text{OperandSize}-1:N] & \leftarrow 0 \\
\text{FI} \\
\text{IF (N > OperandSize - 1)} & \\
\quad CF & \leftarrow 1 \\
\text{ELSE} & \\
\quad CF & \leftarrow 0 \\
\text{FI}
\end{align*}
\]

Flags Affected

ZF, CF and SF flags are updated based on the result. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BZHI: `unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);`
BZHI: `unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD If VEX.W = 1.
CALL—Call Procedure

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E8 cw</td>
<td>CALL rel16</td>
<td>M</td>
<td>N.S.</td>
<td>Valid</td>
<td>Call near, relative, displacement relative to next instruction.</td>
</tr>
<tr>
<td>E8 cd</td>
<td>CALL rel32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.</td>
</tr>
<tr>
<td>FF /2</td>
<td>CALL r/m16</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Call near, absolute indirect, address given in r/m16.</td>
</tr>
<tr>
<td>FF /2</td>
<td>CALL r/m32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Call near, absolute indirect, address given in r/m32.</td>
</tr>
<tr>
<td>FF /2</td>
<td>CALL r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Call near, absolute indirect, address given in r/m64.</td>
</tr>
<tr>
<td>9A cd</td>
<td>CALL ptr16:16</td>
<td>D</td>
<td>Invalid</td>
<td>Valid</td>
<td>Call far, absolute, address given in operand.</td>
</tr>
<tr>
<td>9A cp</td>
<td>CALL ptr16:32</td>
<td>D</td>
<td>Invalid</td>
<td>Valid</td>
<td>Call far, absolute, address given in operand.</td>
</tr>
<tr>
<td>FF /3</td>
<td>CALL m16:16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Call far, absolute indirect address given in m16:16. In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.</td>
</tr>
<tr>
<td>FF /3</td>
<td>CALL m16:32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.</td>
</tr>
<tr>
<td>REX.W + FF /3</td>
<td>CALL m16:64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Offset</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 7, “Task Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for information on performing task switches with the CALL instruction.
Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16, r/m32, or r/m64). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (rel16 or rel32) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)
On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure’s stack, an optional set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure’s code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task’s code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for information on the mechanics of a task switch.

When you execute a task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS’s previous task link field is loaded with the old task’s TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking in Compatibility Mode" in Chapter 7 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

**Mixing 16-Bit and 32-Bit Calls.** When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, for more information.

**Far Calls in Compatibility Mode.** When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target
operand can specify the call gate segment selector either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure’s stack and the segment selector and instruction pointer for the calling procedure’s code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

**Near/(Far) Calls in 64-bit Mode.** When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (m16:16, m16:32 or m16:64). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (m16:16, m16:32 or m16:64). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure’s stack and the segment selector and instruction pointer for the calling procedure’s code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.
Operation

IF near call
  THEN IF near relative call
    THEN
      IF OperandSize = 64
        THEN
          tempDEST ← SignExtend(DEST); (* DEST is rel32 *)
          tempRIP ← RIP + tempDEST;
          IF stack not large enough for a 8-byte return address
            THEN #SS(0); Fi;
          Push(RIP);
          RIP ← tempRIP;
        FI;
      IF OperandSize = 32
        THEN
          tempEIP ← EIP + DEST; (* DEST is rel32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); Fi;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); Fi;
          Push(EIP);
          EIP ← tempEIP;
        FI;
      IF OperandSize = 16
        THEN
          tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
          IF tempEIP is not within code segment limit THEN #GP(0); Fi;
          IF stack not large enough for a 2-byte return address
            THEN #SS(0); Fi;
          Push(IP);
          EIP ← tempEIP;
        FI;
    ELSE (* Near absolute call *)
      IF OperandSize = 64
        THEN
          tempRIP ← DEST; (* DEST is r/m64 *)
          IF stack not large enough for a 8-byte return address
            THEN #SS(0); Fi;
          Push(RIP);
          RIP ← tempRIP;
        FI;
      IF OperandSize = 32
        THEN
          tempEIP ← DEST; (* DEST is r/m32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); Fi;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); Fi;
          Push(EIP);
          EIP ← tempEIP;
        FI;
      IF OperandSize = 16
        THEN
          tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
          IF tempEIP is not within code segment limit THEN #GP(0); Fi;
IF stack not large enough for a 2-byte return address
   THEN #SS(0); FI;
PUSH(IP);
   EIPI ← tempEIP;
FI;
FI; rel/abs
FI; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
THEN
   IF OperandSize = 32
      THEN
         IF stack not large enough for a 6-byte return address
            THEN #SS(0); FI;
         IF DEST[31:16] is not zero THEN #GP(0); FI;
         Push(CS); (* Padded with 16 high-order bits *)
         Push(EIP);
         CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
         EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
      ELSE (* OperandSize = 16 *)
         IF stack not large enough for a 4-byte return address
            THEN #SS(0); FI;
         Push(CS);
         Push(IP);
         CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
         EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
      FI;
   FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
THEN
   IF segment selector in target operand NULL
      THEN #GP(0); FI;
   IF segment selector index not within descriptor table limits
      THEN #GP(new code segment selector); FI;
   Read type and access rights of selected segment descriptor;
   IF IA32_EFER.LMA = 0
      THEN
         IF segment type is not a conforming or nonconforming code segment, call gate, task gate, or TSS
            THEN #GP(segment selector); FI;
      ELSE
         IF segment type is not a conforming or nonconforming code segment or 64-bit call gate,
            THEN #GP(segment selector); FI;
   FI;
   Depending on type and access rights:
      GO TO CONFORMING-CODE-SEGMENT;
      GO TO NONCONFORMING-CODE-SEGMENT;
      GO TO CALL-GATE;
      GO TO TASK-GATE;
      GO TO TASK-STATE-SEGMENT;
   FI;
CONFORMING-CODE-SEGMENT:
  IF L-bit = 1 and D-bit = 1 and IA32_EFER.LMA = 1
      THEN GP(new code segment selector); Fl;
  IF DPL > CPL
      THEN #GP(new code segment selector); Fl;
  IF segment not present
      THEN #NP(new code segment selector); Fl;
  IF stack not large enough for return address
      THEN #SS(0); Fl;
  tempEIP ← DEST(Offset);
  IF target mode = Compatibility mode
      THEN tempEIP ← tempEIP AND 00000000_FFFFFFFFH; Fl;
  IF OperandSize = 16
      THEN
          tempEIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
          IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
              THEN #GP(0); Fl;
          IF tempEIP is non-canonical
              THEN #GP(0); Fl;
          IF OperandSize = 32
              THEN
                  Push(CS); (* Padded with 16 high-order bits *)
                  Push(EIP);
                  CS ← DEST(CodeSegmentSelector);
                  (* Segment descriptor information also loaded *)
                  CS(RPL) ← CPL;
                  EIP ← tempEIP;
              ELSE
                  IF OperandSize = 16
                      THEN
                          Push(CS);
                          Push(IP);
                          CS ← DEST(CodeSegmentSelector);
                          (* Segment descriptor information also loaded *)
                          CS(RPL) ← CPL;
                          EIP ← tempEIP;
                      ELSE (* OperandSize = 64 *)
                          Push(CS); (* Padded with 48 high-order bits *)
                          Push(RIP);
                          CS ← DEST(CodeSegmentSelector);
                          (* Segment descriptor information also loaded *)
                          CS(RPL) ← CPL;
                          RIP ← tempEIP;
                  ELSE Fl;
              END;
          END;
      END;
  END;

NONCONFORMING-CODE-SEGMENT:
  IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
      THEN GP(new code segment selector); Fl;
  IF (RPL > CPL) or (DPL ≠ CPL)
      THEN #GP(new code segment selector); Fl;
  IF segment not present
THEN #NP(new code segment selector); Fl;
IF stack not large enough for return address
THEN #SS(0); Fl;
tempEIP ← DEST(Offset);
IF target mode = Compatibility mode
THEN tempEIP ← tempEIP AND 00000000_FFFFFFFFH; Fl;
IF OperandSize = 16
THEN tempEIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
THEN #GP(0); Fl;
IF tempEIP is non-canonical
THEN #GP(0); Fl;
IF OperandSize = 32
THEN
Push(CS); (* Padded with 16 high-order bits *)
Push(EIP);
CS ← DEST(CodeSegmentSelector);
(* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
ELSE
IF OperandSize = 16
THEN
Push(CS);
Push(IP);
CS ← DEST(CodeSegmentSelector);
(* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
ELSE (* OperandSize = 64 *)
Push(CS); (* Padded with 48 high-order bits *)
Push(RIP);
CS ← DEST(CodeSegmentSelector);
(* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
RIP ← tempEIP;
ELSE (* OperandSize = 64 *)
Push(CS); (* Padded with 48 high-order bits *)
Push(RIP);
CS ← DEST(CodeSegmentSelector);
(* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
RIP ← tempEIP;
Fl;
END;

CALL-GATE:
IF call gate (DPL < CPL) or (RPL > DPL)
THEN #GP(call-gate selector); Fl;
IF call gate not present
THEN #NP(call-gate selector); Fl;
IF call-gate code-segment selector is NULL
THEN #GP(0); Fl;
IF call-gate code-segment selector index is outside descriptor table limits
THEN #GP(call-gate code-segment selector); Fl;
Read call-gate code-segment descriptor;
IF call-gate code-segment descriptor does not indicate a code segment
or call-gate code-segment descriptor DPL > CPL
THEN #GP(call-gate code-segment selector); Fl;
CALL—Call Procedure

INSTRUCTION SET REFERENCE, A-L

IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
THEN #GP(call-gate code-segment selector); Fi;
IF call-gate code segment not present
THEN #NP(call-gate code-segment selector); Fi;
IF call-gate code segment is non-conforming and DPL < CPL
THEN go to MORE-PRIVILEGE;
ELSE go to SAME-PRIVILEGE;
FI;
END;

MORE-PRIVILEGE:
IF current TSS is 32-bit
THEN
   TSSstackAddress ← (new code-segment DPL * 8) + 4;
   IF (TSSstackAddress + 5) > current TSS limit
      THEN #TS(current TSS selector); Fi;
   NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
   NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
ELSE
   IF current TSS is 16-bit
      THEN
         TSSstackAddress ← (new code-segment DPL * 4) + 2
         IF (TSSstackAddress + 3) > current TSS limit
            THEN #TS(current TSS selector); Fi;
         NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
         NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
      ELSE (* current TSS is 64-bit *)
         TSSstackAddress ← (new code-segment DPL * 8) + 4;
         IF (TSSstackAddress + 7) > current TSS limit
            THEN #TS(current TSS selector); Fi;
         NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
         NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
      FI;
   FI;
IF IA32_EFER.LMA = 0 and NewSS is NULL
THEN #TS(NewSS); Fi;
Read new stack-segment descriptor;
IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a writable data segment)
THEN #TS(NewSS); Fi
IF IA32_EFER.LMA = 0 and new stack segment not present
THEN #SS(NewSS); Fi;
IF CallGateSize = 32
THEN
   IF new stack does not have room for parameters plus 16 bytes
      THEN #SS(NewSS); Fi;
   IF CallGate(InstructionPointer) not within new code-segment limit
      THEN #GP(0); Fi;
   SS ← newSS; (* Segment descriptor information also loaded *)
   ESP ← newESP;
   CS:EIP ← CallGate(CS:InstructionPointer);
   (* Segment descriptor information also loaded *)
Push(oldSS:oldESP); (* From calling procedure *)
  temp ← parameter count from call gate, masked to 5 bits;
  Push(parameters from calling procedure's stack, temp)
  Push(oldCS:oldEIP); (* Return address to calling procedure *)
ELSE
  IF CallGateSize = 16
    THEN
      IF new stack does not have room for parameters plus 8 bytes
        THEN #SS(NewSS); FI;
      IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
        THEN #GP(0); FI;
      SS ← newSS; (* Segment descriptor information also loaded *)
      ESP ← newESP;
      CS:IP ← CallGate(CS:InstructionPointer);
      (* Segment descriptor information also loaded *)
      Push(oldSS:oldESP); (* From calling procedure *)
      temp ← parameter count from call gate, masked to 5 bits;
      Push(parameters from calling procedure's stack, temp)
      Push(oldCS:oldEIP); (* Return address to calling procedure *)
  ELSE (* CallGateSize = 64 *)
    IF pushing 32 bytes on the stack would use a non-canonical address
      THEN #SS(0); FI;
    IF (CallGate(InstructionPointer) is non-canonical)
      THEN #GP(0); FI;
    SS ← NewSS; (* NewSS is NULL)
    ESP ← NewESP;
    CS:IP ← CallGate(CS:InstructionPointer);
    (* Segment descriptor information also loaded *)
    Push(oldSS:oldESP); (* From calling procedure *)
  FI;
  FI;
CPL ← CodeSegment(DPL)
CS(RPL) ← CPL
END;
SAME-PRIVILEGE:
  IF CallGateSize = 32
    THEN
      IF stack does not have room for 8 bytes
        THEN #SS(0); FI;
      IF CallGate(InstructionPointer) not within code segment limit
        THEN #GP(0); FI;
      CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
      Push(oldCS:oldEIP); (* Return address to calling procedure *)
  ELSE
    IF CallGateSize = 16
      THEN
        IF stack does not have room for 4 bytes
          THEN #SS(0); FI;
        IF CallGate(InstructionPointer) not within code segment limit
          THEN #GP(0); FI;
        CS:IP ← CallGate(CS:Instruction Pointer);
        (* Segment descriptor information also loaded *)
Push(oldCS:oldIP); (* Return address to calling procedure *)
ELSE (* CallGateSize = 64)
    IF pushing 16 bytes on the stack touches non-canonical addresses
THEN #SS(0); FI;
    IF RIP non-canonical
THEN #GP(0); FI;
    CS:IP ← CallGate(CS:instruction pointer);
(* Segment descriptor information also loaded *)
    Push(oldCS:oldIP); (* Return address to calling procedure *)
    FI;
FI;
CS(RPL) ← CPL
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
        or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF descriptor is not a TSS segment
        THEN #GP(TSS selector); FI;
    IF TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
        or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

**Flags Affected**

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.
Protected Mode Exceptions

#GP(0) If the target offset in destination operand is beyond the new code segment limit.
If the segment selector in the destination operand is NULL.
If the code segment selector in the gate is NULL.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#GP(selector) If a code segment or gate or TSS selector index is outside descriptor table limits.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment’s segment selector is greater than the CPL.
If the DPL for a conforming-code segment is greater than the CPL.
If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS’s segment selector.
If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.
If the segment selector from a call gate is beyond the descriptor table limits.
If the DPL for a code-segment obtained from a call gate is greater than the CPL.
If the segment selector for a TSS has its local/global bit set for local.
If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.
If a memory operand effective address is outside the SS segment limit.

#SS(selector) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.
If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.
If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.

#NP(selector) If a code segment, data segment, call gate, task gate, or TSS is not present.

#TS(selector) If the new stack segment selector and ESP are beyond the end of the TSS.
If the new stack segment selector is NULL.
If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.
If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.
If the new stack segment is not a writable data segment.
If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the target offset is beyond the code segment limit.

#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the target offset is beyond the code segment limit.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.

#UD  If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)  If a memory address accessed by the selector is in non-canonical space.

#GP(0)  If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0)  If a memory address is non-canonical.
If target offset in destination operand is non-canonical.
If the segment selector in the destination operand is NULL.
If the code segment selector in the 64-bit gate is NULL.

#GP(selector)  If code segment or 64-bit call gate is outside descriptor table limits.
If code segment or 64-bit call gate overlaps non-canonical space.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.
If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L-bit set.
If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment’s segment selector is greater than the CPL.
If the DPL for a conforming-code segment is greater than the CPL.
If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
If the upper type field of a 64-bit call gate is not 0x0.
If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.
If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.

#SS(0)  If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.
If a memory operand effective address is outside the SS segment limit.
If the stack address is in a non-canonical form.

#SS(selector)  If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.

#NP(selector)  If a code segment or 64-bit call gate is not present.

#TS(selector)  If the load of the new RSP exceeds the limit of the TSS.

#UD  (64-bit mode only) If a far call is direct to an absolute address in memory.
If the LOCK prefix is used.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>CBW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>AX ← sign-extend of AL.</td>
</tr>
<tr>
<td>98</td>
<td>CWDE</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>EAX ← sign-extend of AX.</td>
</tr>
<tr>
<td>REX.W + 98</td>
<td>CDQE</td>
<td>ZO</td>
<td>N.E.</td>
<td></td>
<td>RAX ← sign-extend of EAX.</td>
</tr>
</tbody>
</table>

Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register.

CBW and CWDE reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16; CWDE is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size. Others may treat these two mnemonics as synonyms (CBW/CWDE) and use the setting of the operand-size attribute to determine the size of values to be converted.

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

Operation

IF OperandSize = 16 (* Instruction = CBW *)
    THEN
        AX ← SignExtend(AL);
    ELSE IF (OperandSize = 32, Instruction = CWDE)
        EAX ← SignExtend(AX); Fl;
    ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
        RAX ← SignExtend(EAX);
    Fl;

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
### CLAC—Clear AC Flag in EFLAGS Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 CA CLAC</td>
<td>ZO</td>
<td>V/V</td>
<td>SMAP</td>
<td>Clear the AC flag in the EFLAGS register.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

### Operation

EFLAGS.AC ← 0;

### Flags Affected

AC cleared. Other flags are unaffected.

### Protected Mode Exceptions

- **#UD**
  - If the LOCK prefix is used.
  - If the CPL > 0.
  - If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Real-Address Mode Exceptions

- **#UD**
  - If the LOCK prefix is used.
  - If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Virtual-8086 Mode Exceptions

- **#UD**
  - The CLAC instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

- **#UD**
  - If the LOCK prefix is used.
  - If the CPL > 0.
  - If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### 64-Bit Mode Exceptions

- **#UD**
  - If the LOCK prefix is used.
  - If the CPL > 0.
  - If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.
CLC—Clear Carry Flag

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>CLC</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Clear CF flag.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Clears the CF flag in the EFLAGS register. Operation is the same in all modes.

**Operation**

\[ CF \leftarrow 0; \]

**Flags Affected**

The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD If the LOCK prefix is used.
CLD—Clear Direction Flag

**Opcode** | **Instruction** | **Op/En** | **64-bit Mode** | **Compat/ Leg Mode** | **Description**  
---|---|---|---|---|---  
FC | CLD | ZO | Valid | Valid | Clear DF flag.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all modes.

**Operation**

DF ← 0;

**Flags Affected**

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD If the LOCK prefix is used.
CLFLUSH—Flush Cache Line

### Opcode Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE 77</td>
<td>CLFLUSH m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Flushes cache line containing m8.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (CPUID.01H:EDX[bit 19]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, fence instructions, and executions of CLFLUSHOPT to the same cache line.\(^1\) They are not ordered with respect to executions of CLFLUSHOPT to different cache lines.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSH instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSH instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSH instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in 1A-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

#### Operation

Flush_Cache_Line(SRC);

#### Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSH: void _mm_clflush(void const *p)

---

1. Earlier versions of this manual specified that executions of the CLFLUSH instruction were ordered only by the MFENCE instruction. All processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.
**Protected Mode Exceptions**

- **#GP(0)**: For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
- **#SS(0)**: For an illegal address in the SS segment.
- **#PF(fault-code)**: For a page fault.
- **#UD**: If CPUID.01H:EDX.CLFSH[bit 19] = 0.
  If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP**: If any part of the operand lies outside the effective address space from 0 to FFFFH.
- **#UD**: If CPUID.01H:EDX.CLFSH[bit 19] = 0.
  If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

**Compatibly Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)**: If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)**: If the memory address is in a non-canonical form.
- **#PF(fault-code)**: For a page fault.
- **#UD**: If CPUID.01H:EDX.CLFSH[bit 19] = 0.
  If the LOCK prefix is used.
CLFLUSHOPT—Flush Cache Line Optimized

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F AE /7</td>
<td>CLFLUSHOPT m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Flushes cache line containing m8.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSHOPT is indicated by the presence of the CPUID feature flag CLFLUSHOPT (CPUID.(EAX=7,ECX=0):EBX[bit 23]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSHOPT instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to the following accesses to the cache line being invalidated: writes, executions of CLFLUSH, and executions of CLFLUSHOPT. They are not ordered with respect to writes, executions of CLFLUSH, or executions of CLFLUSHOPT that access other cache lines; to enforce ordering with such an operation, software can insert an SFENCE instruction between CFLUSHOPT and that operation.

CLFLUSHOPT operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

Flush_Cache_Line_Optimized(SRC);

**Intel C/C++ Compiler Intrinsic Equivalents**

CLFLUSHOPT: void _mm_clflushopt(void const *p)
Protected Mode Exceptions

#GP(0)  For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)  For an illegal address in the SS segment.
#PF(fault-code) For a page fault.
#UD If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0.
If the LOCK prefix is used.
If an instruction prefix F2H or F3H is used.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0.
If the LOCK prefix is used.
If an instruction prefix F2H or F3H is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.
#PF(fault-code) For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#UD If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0.
If the LOCK prefix is used.
If an instruction prefix F2H or F3H is used.
CLI — Clear Interrupt Flag

In most cases, CLI clears the IF flag in the EFLAGS register and no other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3 and either VME mode or PVI mode is active, CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 3-7 indicates the action of the CLI instruction depending on the processor operating mode, IOPL, and CPL.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

In most cases, CLI clears the IF flag in the EFLAGS register and no other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

### Table 3-7. Decision Table for CLI Results

<table>
<thead>
<tr>
<th>Mode</th>
<th>IOPL</th>
<th>CLI Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-address</td>
<td>X</td>
<td>IF = 0</td>
</tr>
<tr>
<td>Protected, not PVI</td>
<td>≥ CPL</td>
<td>IF = 0</td>
</tr>
<tr>
<td>Protected, PVI</td>
<td>&lt; CPL</td>
<td>#GP fault</td>
</tr>
<tr>
<td>Protected, PVI</td>
<td>3</td>
<td>IF = 0</td>
</tr>
<tr>
<td>Virtual-8086, not VME</td>
<td>3</td>
<td>VIF = 0</td>
</tr>
<tr>
<td>Virtual-8086, VME</td>
<td>3</td>
<td>IF = 0</td>
</tr>
<tr>
<td></td>
<td>0–2</td>
<td>#GP fault</td>
</tr>
<tr>
<td></td>
<td>0–2</td>
<td>VIF = 0</td>
</tr>
</tbody>
</table>

### NOTES:

1. X = This setting has no effect on instruction operation.
2. For this table, “protected mode” applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.
**Operation**

IF CR0.PE = 0
    THEN IF ← 0; (* Reset Interrupt Flag *)
ELS
    IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
        THEN IF ← 0; (* Reset Interrupt Flag *)
    ELSE
        IF VME mode OR PVI mode
            THEN VIF ← 0; (* Reset Virtual Interrupt Flag *)
        ELSE #GP(0);

FI;
FI;

**Flags Affected**
Either the IF flag or the VIF flag is cleared to 0. Other flags are unaffected.

**Protected Mode Exceptions**
- #GP(0) If CPL is greater than IOPL and PVI mode is not active.
- #GP(0) If CPL is greater than IOPL and less than 3.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**
- #GP(0) If IOPL is less than 3 and VME mode is not active.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**
Same exceptions as in protected mode.
CLTS—Clear Task-Switched Flag in CR0

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 06</td>
<td>CLTS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Clears TS flag in CR0.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 25, “VMX Non-Root Operation,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

CR0.TS[bit 3] ← 0;

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) CLTS is not recognized in virtual-8086 mode.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #GP(0) If the CPL is greater than 0.
- #UD If the LOCK prefix is used.
CLWB—Cache Line Write Back

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F AE /6 CLWB m8</td>
<td>M</td>
<td>V/V</td>
<td>CLWB</td>
<td>Writes back modified cache line containing m8, and may retain the line in cache hierarchy in non-modified state.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

The availability of CLWB instruction is indicated by the presence of the CPUID feature flag CLWB (bit 24 of the EBX register, see “CPUID — CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLWB instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLWB instruction that references the cache line).

CLWB instruction is ordered only by store-fencing operations. For example, software can use an SFENCE, MFENCE, XCHG, or LOCK-prefixed instructions to ensure that previous stores are included in the write-back. CLWB instruction need not be ordered by another CLWB or CLFLUSHOPT instruction. CLWB is implicitly ordered with older stores executed by the logical processor to the same address.

For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use CLWB (with appropriate fencing) instead of CLFLUSH or CLFLUSHOPT for improved performance.

The CLWB instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLWB instruction sets the accessed flag but not the dirty flag in the page tables.

In some implementations, the CLWB instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLWB instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLWB instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

Operation

Cache Line Write Back(m8);

Flags Affected

None.

1. ModRM.MOD != 011B
C/C++ Compiler Intrinsic Equivalent
CLWB void _mm_clwb(void const *p);

Protected Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0) For an illegal address in the SS segment.
#PF(fault-code) For a page fault.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.
#PF(fault-code) For a page fault.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
CMC—Complement Carry Flag

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>CMC</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Complement CF flag.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

Operation
EFLAGS.CF[bit 0] ← NOT EFLAGS.CF[bit 0];

Flags Affected
The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)
#UD If the LOCK prefix is used.
## CMOVcc—Conditional Move

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compats/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 47 /r</td>
<td>CMOVA r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 47 /r</td>
<td>CMOVA r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 47 /r</td>
<td>CMOVA r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 43 /r</td>
<td>CMOVAE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above or equal (CF=0).</td>
</tr>
<tr>
<td>0F 43 /r</td>
<td>CMOVAE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above or equal (CF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 43 /r</td>
<td>CMOVAE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if above or equal (CF=0).</td>
</tr>
<tr>
<td>0F 42 /r</td>
<td>COMVB r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>0F 42 /r</td>
<td>COMVB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 42 /r</td>
<td>COMVB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>0F 46 /r</td>
<td>CMOVB r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 46 /r</td>
<td>CMOVB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 46 /r</td>
<td>CMOVB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 42 /r</td>
<td>CMOVBE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if carry (CF=1).</td>
</tr>
<tr>
<td>0F 42 /r</td>
<td>CMOVBE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if carry (CF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 42 /r</td>
<td>CMOVBE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if carry (CF=1).</td>
</tr>
<tr>
<td>0F 44 /r</td>
<td>CMOVF r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less (SF ≠ OF).</td>
</tr>
<tr>
<td>0F 44 /r</td>
<td>CMOVF r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less (SF ≠ OF).</td>
</tr>
<tr>
<td>REX.W + 0F 44 /r</td>
<td>CMOVF r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if less (SF ≠ OF).</td>
</tr>
<tr>
<td>0F 4F /r</td>
<td>CMOVG r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 4F /r</td>
<td>CMOVG r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4F /r</td>
<td>CMOVG r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.A.</td>
<td>Move if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 4D /r</td>
<td>CMOVGE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 4D /r</td>
<td>CMOVGE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4D /r</td>
<td>CMOVGE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 4C /r</td>
<td>CMOVLC r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less (SF ≠ OF).</td>
</tr>
<tr>
<td>0F 4C /r</td>
<td>CMOVLC r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less (SF ≠ OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4C /r</td>
<td>CMOVLC r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if less (SF ≠ OF).</td>
</tr>
<tr>
<td>0F 4E /r</td>
<td>CMOVEL r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less or equal (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>0F 4E /r</td>
<td>CMOVEL r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less or equal (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4E /r</td>
<td>CMOVEL r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if less or equal (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>0F 46 /r</td>
<td>CMOVNA r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 46 /r</td>
<td>CMOVNA r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 46 /r</td>
<td>CMOVNA r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 42 /r</td>
<td>CMOVNAE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 42 /r</td>
<td>CMOVNAE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 42 /r</td>
<td>CMOVNAE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 43 /r</td>
<td>CMOVNB r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below (CF=0).</td>
</tr>
<tr>
<td>0F 43 /r</td>
<td>CMOVNB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below (CF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 43 /r</td>
<td>CMOVNB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not below (CF=0).</td>
</tr>
<tr>
<td>0F 47 /r</td>
<td>CMOVNBE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Instruction</td>
<td>Op/En</td>
<td>64-Bit Mode</td>
<td>Comp/ Leg Mode</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------</td>
<td>-------</td>
<td>-------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>0F 47/r</td>
<td>CMOVNB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 47/r</td>
<td>CMOVNB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 43/r</td>
<td>CMOVNC r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not carry (CF=0).</td>
</tr>
<tr>
<td>0F 43/r</td>
<td>CMOVNC r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not carry (CF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 43/r</td>
<td>CMOVNC r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not carry (CF=0).</td>
</tr>
<tr>
<td>0F 45/r</td>
<td>CMOVNE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 45/r</td>
<td>CMOVNE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not equal (ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 45/r</td>
<td>CMOVNE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 4E/r</td>
<td>CMOVNG r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>0F 4E/r</td>
<td>CMOVNG r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4E/r</td>
<td>CMOVNG r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not greater (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>0F 4C/r</td>
<td>CMOVNGE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater or equal (SF≠ OF).</td>
</tr>
<tr>
<td>0F 4C/r</td>
<td>CMOVNGE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater or equal (SF≠ OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4C/r</td>
<td>CMOVNGE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not greater or equal (SF≠ OF).</td>
</tr>
<tr>
<td>0F 4D/r</td>
<td>CMOVNL r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 4D/r</td>
<td>CMOVNL r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less (SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4D/r</td>
<td>CMOVNL r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 4F/r</td>
<td>CMOVNLE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 4F/r</td>
<td>CMOVNLE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4F/r</td>
<td>CMOVNLE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 41/r</td>
<td>CMOVNO r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not overflow (OF=0).</td>
</tr>
<tr>
<td>0F 41/r</td>
<td>CMOVNO r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not overflow (OF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 41/r</td>
<td>CMOVNO r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not overflow (OF=0).</td>
</tr>
<tr>
<td>0F 4B/r</td>
<td>CMOVNP r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not parity (PF=0).</td>
</tr>
<tr>
<td>0F 4B/r</td>
<td>CMOVNP r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not parity (PF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 4B/r</td>
<td>CMOVNP r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not parity (PF=0).</td>
</tr>
<tr>
<td>0F 49/r</td>
<td>CMOVNS r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not sign (SF=0).</td>
</tr>
<tr>
<td>0F 49/r</td>
<td>CMOVNS r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not sign (SF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 49/r</td>
<td>CMOVNS r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not sign (SF=0).</td>
</tr>
<tr>
<td>0F 45/r</td>
<td>CMOVNZ r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not zero (ZF=0).</td>
</tr>
<tr>
<td>0F 45/r</td>
<td>CMOVNZ r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not zero (ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 45/r</td>
<td>CMOVNZ r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not zero (ZF=0).</td>
</tr>
<tr>
<td>0F 40/r</td>
<td>CMovo r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if overflow (OF=1).</td>
</tr>
<tr>
<td>0F 40/r</td>
<td>CMovo r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if overflow (OF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 40/r</td>
<td>CMovo r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if overflow (OF=1).</td>
</tr>
<tr>
<td>0F 4A/r</td>
<td>CMovp r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity (PF=1).</td>
</tr>
<tr>
<td>0F 4A/r</td>
<td>CMovp r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity (PF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 4A/r</td>
<td>CMovp r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if parity (PF=1).</td>
</tr>
<tr>
<td>0F 4A/r</td>
<td>CMovpe r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity even (PF=1).</td>
</tr>
<tr>
<td>0F 4A/r</td>
<td>CMovpe r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity even (PF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 4A/r</td>
<td>CMovpe r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if parity even (PF=1).</td>
</tr>
</tbody>
</table>
The CMOVcc instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOVcc mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOVcc instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Operation

```
temp ← SRC
IF condition TRUE
    THEN
        DEST ← temp;
    FI;
ELSE
    IF (OperandSize = 32 and IA-32e mode active)
        THEN
            DEST[63:32] ← 0;
        FI;
FI;
```

CMOVcc—Conditional Move
Flags Affected
None.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
   current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
   current privilege level is 3.
#UD If the LOCK prefix is used.
**CMP—Compare Two Operands**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3C ib</td>
<td>CMP AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with AL.</td>
</tr>
<tr>
<td>3D iw</td>
<td>CMP AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm16 with AX.</td>
</tr>
<tr>
<td>3D id</td>
<td>CMP EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm32 with EAX.</td>
</tr>
<tr>
<td>REX.W + 3D id</td>
<td>CMP RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare imm32 sign-extended to 64-bits with RAX.</td>
</tr>
<tr>
<td>80 /7 ib</td>
<td>CMP r/m8, imm8</td>
<td>Mi</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with r/m8.</td>
</tr>
<tr>
<td>REX + 80 /7 ib</td>
<td>CMP r/m8, imm8</td>
<td>Mi</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare imm8 with r/m8.</td>
</tr>
<tr>
<td>81 /7 iw</td>
<td>CMP r/m16, imm16</td>
<td>Mi</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm16 with r/m16.</td>
</tr>
<tr>
<td>81 /7 id</td>
<td>CMP r/m32, imm32</td>
<td>Mi</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm32 with r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /7 id</td>
<td>CMP r/m64, imm32</td>
<td>Mi</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare imm32 sign-extended to 64-bits with r/m64.</td>
</tr>
<tr>
<td>83 /7 ib</td>
<td>CMP r/m16, imm8</td>
<td>Mi</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with r/m16.</td>
</tr>
<tr>
<td>83 /7 ib</td>
<td>CMP r/m32, imm8</td>
<td>Mi</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with r/m32.</td>
</tr>
<tr>
<td>REX.W + 83 /7 ib</td>
<td>CMP r/m64, imm8</td>
<td>Mi</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare imm8 with r/m64.</td>
</tr>
<tr>
<td>38 /r</td>
<td>CMP r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r8 with r/m8.</td>
</tr>
<tr>
<td>REX + 38 /r</td>
<td>CMP r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r8 with r/m8.</td>
</tr>
<tr>
<td>39 /r</td>
<td>CMP r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r16 with r/m16.</td>
</tr>
<tr>
<td>39 /r</td>
<td>CMP r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r32 with r/m32.</td>
</tr>
<tr>
<td>REX.W + 39 /r</td>
<td>CMP r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r64 with r/m64.</td>
</tr>
<tr>
<td>3A /r</td>
<td>CMP r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r/m8 with r8.</td>
</tr>
<tr>
<td>REX + 3A /r</td>
<td>CMP r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r/m8 with r8.</td>
</tr>
<tr>
<td>3B /r</td>
<td>CMP r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r/m16 with r16.</td>
</tr>
<tr>
<td>3B /r</td>
<td>CMP r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r/m32 with r32.</td>
</tr>
<tr>
<td>REX.W + 3B /r</td>
<td>CMP r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r/m64 with r64.</td>
</tr>
</tbody>
</table>

**NOTES:**
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX (r)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the Jcc, CMOVcc, and SETcc instructions are based on the results of a CMP instruction. Appendix B, "EFLAGS Condition Codes," in the *Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.
In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[ \text{temp} \leftarrow \text{SRC1} - \text{SignExtend}(\text{SRC2}); \]
\[ \text{ModifyStatusFlags}; (* \text{Modify status flags in the same manner as the SUB instruction}*) \]

**Flags Affected**
The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.
CMPPD—Compare Packed Double-Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F C2 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed double-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.NDS.128:66.0F:WIG C2 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed double-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.NDS.256:66.0F:WIG C2 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed double-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>EVEX.NDS.128:66.0F:W1 C2 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed double-precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256:66.0F:W1 C2 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed double-precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512:66.0F:W1 C2 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed double-precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM :reg (r, w)</td>
<td>ModRM :r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM :reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM :r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM :reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM :r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

**EVEX encoded versions:** The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

**VEX.256 encoded version:** The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

**128-bit Legacy SSE version:** The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).
VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:
• For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
• For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

### Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions

<table>
<thead>
<tr>
<th>Predicate</th>
<th>imm8 Value</th>
<th>Description</th>
<th>Result: A Is 1st Operand, B Is 2nd Operand</th>
<th>Signals #IA on QNAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ_OQ (EQ)</td>
<td>0H</td>
<td>Equal (ordered, non-signaling)</td>
<td>False False True False False No</td>
<td></td>
</tr>
<tr>
<td>LT_OS (LT)</td>
<td>1H</td>
<td>Less-than (ordered, signaling)</td>
<td>False True False False Yes</td>
<td></td>
</tr>
<tr>
<td>LE_OS (LE)</td>
<td>2H</td>
<td>Less-than-or-equal (ordered, signaling)</td>
<td>False True True False Yes</td>
<td></td>
</tr>
<tr>
<td>UNORD_Q (UNORD)</td>
<td>3H</td>
<td>Unordered (non-signaling)</td>
<td>False False False True No</td>
<td></td>
</tr>
<tr>
<td>NEQ_UQ (NEQ)</td>
<td>4H</td>
<td>Not-equal (unordered, non-signaling)</td>
<td>True True False False No</td>
<td></td>
</tr>
<tr>
<td>NLT_US (NLT)</td>
<td>5H</td>
<td>Not-less-than (unordered, signaling)</td>
<td>True False False True Yes</td>
<td></td>
</tr>
<tr>
<td>NLE_US (NLE)</td>
<td>6H</td>
<td>Not-less-than-or-equal (unordered, signaling)</td>
<td>True False True True Yes</td>
<td></td>
</tr>
<tr>
<td>ORD_OQ (ORD)</td>
<td>7H</td>
<td>Ordered (non-signaling)</td>
<td>True True True False No</td>
<td></td>
</tr>
<tr>
<td>EQ_UQ</td>
<td>8H</td>
<td>Equal (unordered, non-signaling)</td>
<td>False False True True Yes</td>
<td></td>
</tr>
<tr>
<td>NGE_US (NGE)</td>
<td>9H</td>
<td>Not-greater-than-or-equal (unordered, signaling)</td>
<td>False True False True Yes</td>
<td></td>
</tr>
<tr>
<td>NGT_US (NGT)</td>
<td>AH</td>
<td>Not-greater-than (unordered, signaling)</td>
<td>False True True True Yes</td>
<td></td>
</tr>
<tr>
<td>FALSE_OQ (FALSE)</td>
<td>8H</td>
<td>False (ordered, non-signaling)</td>
<td>False False False False No</td>
<td></td>
</tr>
<tr>
<td>NEQ_OQ</td>
<td>CH</td>
<td>Not-equal (ordered, non-signaling)</td>
<td>True True False False No</td>
<td></td>
</tr>
<tr>
<td>GE_OS (GE)</td>
<td>DH</td>
<td>Greater-than-or-equal (ordered, signaling)</td>
<td>True False True False Yes</td>
<td></td>
</tr>
<tr>
<td>GT_OQ (GT)</td>
<td>EH</td>
<td>Greater-than (ordered, signaling)</td>
<td>True False False False Yes</td>
<td></td>
</tr>
<tr>
<td>TRUE_UQ (TRUE)</td>
<td>FH</td>
<td>True (unordered, non-signaling)</td>
<td>True True True True No</td>
<td></td>
</tr>
<tr>
<td>EQ_OO</td>
<td>10H</td>
<td>Equal (ordered, signaling)</td>
<td>False False True False Yes</td>
<td></td>
</tr>
<tr>
<td>LT_OQ</td>
<td>11H</td>
<td>Less-than (ordered, nonsignaling)</td>
<td>False True False False No</td>
<td></td>
</tr>
<tr>
<td>LE_OO</td>
<td>12H</td>
<td>Less-than-or-equal (ordered, nonsignaling)</td>
<td>False True False False No</td>
<td></td>
</tr>
<tr>
<td>UNORD_S</td>
<td>13H</td>
<td>Unordered (signaling)</td>
<td>False False False True No</td>
<td></td>
</tr>
<tr>
<td>NEQ_US</td>
<td>14H</td>
<td>Not-equal (unordered, signaling)</td>
<td>True True False True Yes</td>
<td></td>
</tr>
<tr>
<td>NLT_UQ</td>
<td>15H</td>
<td>Not-less-than (unordered, nonsignaling)</td>
<td>True False True True No</td>
<td></td>
</tr>
<tr>
<td>NLE_UQ</td>
<td>16H</td>
<td>Not-less-than-or-equal (unordered, nonsignaling)</td>
<td>True False True True No</td>
<td></td>
</tr>
<tr>
<td>ORD_S</td>
<td>17H</td>
<td>Ordered (signaling)</td>
<td>True True True False Yes</td>
<td></td>
</tr>
<tr>
<td>EQ_US</td>
<td>18H</td>
<td>Equal (unordered, signaling)</td>
<td>False False True True Yes</td>
<td></td>
</tr>
<tr>
<td>NGE_UQ</td>
<td>19H</td>
<td>Not-greater-than-or-equal (unordered, nonsignaling)</td>
<td>False True False True No</td>
<td></td>
</tr>
</tbody>
</table>
The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-2. Compiler should treat reserved Imm8 values as illegal syntax.

### Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

<table>
<thead>
<tr>
<th>Predicate</th>
<th>imm8 Value</th>
<th>Description</th>
<th>Result: A is 1st Operand, B is 2nd Operand</th>
<th>Signals #IA on QNAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGT_UQ</td>
<td>1AH</td>
<td>Not-greater-than (unordered, nonsignaling)</td>
<td>False True True True</td>
<td>No</td>
</tr>
<tr>
<td>FALSE_OS</td>
<td>1BH</td>
<td>False (ordered, signaling)</td>
<td>False True False False</td>
<td>Yes</td>
</tr>
<tr>
<td>NEQ_OS</td>
<td>1CH</td>
<td>Not-equal (ordered, signaling)</td>
<td>True True False False</td>
<td>Yes</td>
</tr>
<tr>
<td>GE_OQ</td>
<td>1DH</td>
<td>Greater-than-or-equal (ordered, nonsignaling)</td>
<td>True False True False</td>
<td>No</td>
</tr>
<tr>
<td>GT_OQ</td>
<td>1EH</td>
<td>Greater-than (ordered, nonsignaling)</td>
<td>True False False False</td>
<td>No</td>
</tr>
<tr>
<td>TRUE_US</td>
<td>1FH</td>
<td>True (unordered, signaling)</td>
<td>True True True True</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**NOTES:**
1. If either operand A or B is a NaN.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-2. Compiler should treat reserved Imm8 values as illegal syntax.

### Table 3-2. Pseudo-Op and CMPPD Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPLTPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPLEPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPUNORDPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPNEQPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPNLTPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNLEPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-3, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal
syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 3-3.

Table 3-3. Pseudo-Op and VCMPPD Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPEQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMPLTPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPLEPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPPNORDPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPEQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMPLTPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMPLEPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMPPNORDPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPGEQ_UQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMPNGEPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMPNGTPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 0AH</td>
</tr>
<tr>
<td>VCMPPNORD_UQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 0BH</td>
</tr>
<tr>
<td>VCMPPNGEPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 0CH</td>
</tr>
<tr>
<td>VCMPGEQ_OSUSPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 0DH</td>
</tr>
<tr>
<td>VCMPNGEPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 0EH</td>
</tr>
<tr>
<td>VCMPPNORD_USPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 0FH</td>
</tr>
<tr>
<td>VCMPGEQ_OSUSPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 10H</td>
</tr>
<tr>
<td>VCMPNGEPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 11H</td>
</tr>
<tr>
<td>VCMPGEQ_OQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 12H</td>
</tr>
<tr>
<td>VCMPGEQ_UQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 13H</td>
</tr>
<tr>
<td>VCMPPNORD_OSUSPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 14H</td>
</tr>
<tr>
<td>VCMPPNORD_UQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 15H</td>
</tr>
<tr>
<td>VCMPPNORD_USUSPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 16H</td>
</tr>
<tr>
<td>VCMPPNORD_USPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 17H</td>
</tr>
<tr>
<td>VCMPGEQ_OSUSPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 18H</td>
</tr>
<tr>
<td>VCMPNGEPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 19H</td>
</tr>
<tr>
<td>VCMPNGTPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 1AH</td>
</tr>
<tr>
<td>VCMPPNORD_OSUSPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 1BH</td>
</tr>
<tr>
<td>VCMPPNORD_UQPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 1CH</td>
</tr>
<tr>
<td>VCMPPNORD_USUSPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 1EH</td>
</tr>
<tr>
<td>VCMPPNORD_USPD reg1, reg2, reg3</td>
<td>VCMPPD reg1, reg2, reg3, 1FH</td>
</tr>
</tbody>
</table>
### Operation

CASE (COMPARISON PREDICATE) OF

<table>
<thead>
<tr>
<th>CASE</th>
<th>OP3</th>
<th>OP5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OP3 ← EQ_OQ; OP5 ← EQ_OQ;</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>OP3 ← LT_OS; OP5 ← LT_OS;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OP3 ← LE_OS; OP5 ← LE_OS;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OP3 ← UNORD_Q; OP5 ← UNORD_Q;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>OP3 ← NLT_US; OP5 ← NLT_US;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>OP3 ← NLE_US; OP5 ← NLE_US;</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>OP3 ← ORD_Q; OP5 ← ORD_Q;</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>OP5 ← EQ_UQ;</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>OP5 ← NGE_US;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>OP5 ← NGT_US;</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>OP5 ← FALSE_OQ;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>OP5 ← NEQ_OQ;</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>OP5 ← GE_OS;</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>OP5 ← GT_OS;</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>OP5 ← TRUE_UQ;</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>OP5 ← EQ_OS;</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>OP5 ← LT_OQ;</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>OP5 ← LE_OQ;</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>OP5 ← UNORD_S;</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>OP5 ← NEQ_US;</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>OP5 ← NLT_UQ;</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>OP5 ← NLE_UQ;</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>OP5 ← ORD_S;</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>OP5 ← EQ_US;</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>OP5 ← NGE_UQ;</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>OP5 ← NGT_UQ;</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>OP5 ← FALSE_OS;</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>OP5 ← NEQ_OS;</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>OP5 ← GE_OQ;</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>OP5 ← GT_OQ;</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>OP5 ← TRUE_US;</td>
<td></td>
</tr>
</tbody>
</table>

DEFAULT: Reserved;

ESAC
VCMPPD (EVEX encoded versions)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
i \leftarrow j \times 64 \\
\text{IF } k2[j] \text{ OR } \text{ *no writemask*} \\
\text{THEN} \\
\quad \text{IF } (EVEX.b = 1) \text{ AND } \text{ (SRC2 *is memory*)} \\
\quad \text{THEN} \\
\quad \quad \text{CMP} \leftarrow \text{SRC1}[i+63:i] \text{ OP5 SRC2}[63:0] \\
\quad \text{ELSE} \\
\quad \quad \text{CMP} \leftarrow \text{SRC1}[i+63:i] \text{ OP5 SRC2}[i+63:i] \\
\quad \text{FI;} \\
\quad \text{IF CMP = TRUE} \\
\quad \quad \text{DEST}[j] \leftarrow 1; \\
\quad \text{ELSE DEST}[j] \leftarrow 0; \text{FI;} \\
\text{ELSE DEST}[j] \leftarrow 0 ; \text{zeroing-masking only} \\
\text{FI;} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX}_K-1:KL] \leftarrow 0
\]

VCMPPD (VEX.256 encoded version)

\[
\begin{align*}
\text{CMP0} & \leftarrow \text{SRC1}[63:0] \text{ OP5 SRC2}[63:0]; \\
\text{CMP1} & \leftarrow \text{SRC1}[127:64] \text{ OP5 SRC2}[127:64]; \\
\text{CMP2} & \leftarrow \text{SRC1}[191:128] \text{ OP5 SRC2}[191:128]; \\
\text{CMP3} & \leftarrow \text{SRC1}[255:192] \text{ OP5 SRC2}[255:192]; \\
\text{IF CMP0 = TRUE} \\
\quad \text{THEN DEST}[63:0] \leftarrow \text{FFFFFFFFFFFFFFFFH;} \\
\quad \text{ELSE DEST}[63:0] \leftarrow \text{0000000000000000H;} \text{FI;} \\
\text{IF CMP1 = TRUE} \\
\quad \text{THEN DEST}[127:64] \leftarrow \text{FFFFFFFFFFFFFFFFH;} \\
\quad \text{ELSE DEST}[127:64] \leftarrow \text{0000000000000000H;} \text{FI;} \\
\text{IF CMP2 = TRUE} \\
\quad \text{THEN DEST}[191:128] \leftarrow \text{FFFFFFFFFFFFFFFFH;} \\
\quad \text{ELSE DEST}[191:128] \leftarrow \text{0000000000000000H;} \text{FI;} \\
\text{IF CMP3 = TRUE} \\
\quad \text{THEN DEST}[255:192] \leftarrow \text{FFFFFFFFFFFFFFFFH;} \\
\quad \text{ELSE DEST}[255:192] \leftarrow \text{0000000000000000H;} \text{FI;} \\
\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0
\end{align*}
\]

VCMPPD (VEX.128 encoded version)

\[
\begin{align*}
\text{CMP0} & \leftarrow \text{SRC1}[63:0] \text{ OP5 SRC2}[63:0]; \\
\text{CMP1} & \leftarrow \text{SRC1}[127:64] \text{ OP5 SRC2}[127:64]; \\
\text{IF CMP0 = TRUE} \\
\quad \text{THEN DEST}[63:0] \leftarrow \text{FFFFFFFFFFFFFFFFH;} \\
\quad \text{ELSE DEST}[63:0] \leftarrow \text{0000000000000000H;} \text{FI;} \\
\text{IF CMP1 = TRUE} \\
\quad \text{THEN DEST}[127:64] \leftarrow \text{FFFFFFFFFFFFFFFFH;} \\
\quad \text{ELSE DEST}[127:64] \leftarrow \text{0000000000000000H;} \text{FI;} \\
\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\end{align*}
\]
CMPPD (128-bit Legacy SSE version)

CMP0 \leftarrow \text{SRC1}[63:0] \text{ OP3 SRC2}[63:0];
CMP1 \leftarrow \text{SRC1}[127:64] \text{ OP3 SRC2}[127:64];

\begin{align*}
\text{IF CMP0 = TRUE} & & \text{THEN } \text{DEST}[63:0] \leftarrow \text{FFFFFFFFFFFFFFFFH}; \\
& & \text{ELSE } \text{DEST}[63:0] \leftarrow \text{0000000000000000H}; \text{FI}; \\
\text{IF CMP1 = TRUE} & & \text{THEN } \text{DEST}[127:64] \leftarrow \text{FFFFFFFFFFFFFFFFH}; \\
& & \text{ELSE } \text{DEST}[127:64] \leftarrow \text{0000000000000000H}; \text{FI}; \\
\text{DEST}[\text{MAXVL-1:128}] \text{ (Unmodified)}
\end{align*}

\textbf{Intel C/C++ Compiler Intrinsic Equivalent}

\begin{align*}
\text{VCMPPD \_mmask8 \_mm512\_cmp\_pd\_mask( \_m512d a, \_m512d b, int imm);} \\
\text{VCMPPD \_mmask8 \_mm512\_cmp\_round\_pd\_mask( \_m512d a, \_m512d b, int imm, int sae);} \\
\text{VCMPPD \_mmask8 \_mm512\_mask\_cmp\_pd\_mask( \_mmask8 k1, \_m512d a, \_m512d b, int imm);} \\
\text{VCMPPD \_mmask8 \_mm512\_mask\_cmp\_round\_pd\_mask( \_mmask8 k1, \_m512d a, \_m512d b, int imm, int sae);} \\
\text{VCMPPD \_mmask8 \_mm256\_cmp\_pd\_mask( \_m256d a, \_m256d b, int imm);} \\
\text{VCMPPD \_mmask8 \_mm256\_mask\_cmp\_pd\_mask( \_mmask8 k1, \_m256d a, \_m256d b, int imm);} \\
\text{VCMPPD \_mmask8 \_mm\_cmp\_pd\_mask( \_m128d a, \_m128d b, int imm);} \\
\text{VCMPPD \_mmask8 \_mm\_mask\_cmp\_pd\_mask( \_mmask8 k1, \_m128d a, \_m128d b, int imm);} \\
\text{(V)CMPPD \_m128 \_mm\_cmp\_pd( \_m256d a, \_m256d b, int imm)}
\end{align*}

\textbf{SIMD Floating-Point Exceptions}

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.
Denormal

\textbf{Other Exceptions}

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
CMPPS—Compare Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare packed single-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed single-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed single-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 C2 /r ib VCMPPS k1 [k2], xmm2, xmm3/m128/m32bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed single-precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 C2 /r ib VCMPPS k1 [k2], ymm2, ymm3/m256/m32bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed single-precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 C2 /r ib VCMPPS k1 [k2], zmm2, zmm3/m512/m32bcst(sae), imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed single-precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).
VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the “greater-than”, “greater-than-or-equal”, “not-greater-than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-4. Compiler should treat reserved Imm8 values as illegal syntax.

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPLTPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPLEPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPUNORDPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPNEQPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPLTPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNLEPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-5.
<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPLEQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMPLTAPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPLEAPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPUNORDPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPLTAPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMPPNLEPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMPPORDPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMPEQ_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPNQEPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMPEQ_OSPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMPPUNORD_SPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0AH</td>
</tr>
<tr>
<td>VCMPPGAPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0BH</td>
</tr>
<tr>
<td>VCMPPGSPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0CH</td>
</tr>
<tr>
<td>VCMPPORD_SPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0DH</td>
</tr>
<tr>
<td>VCMPTUEPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0EH</td>
</tr>
<tr>
<td>VCMPPG_SPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0FH</td>
</tr>
<tr>
<td>VCMPLT_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 10H</td>
</tr>
<tr>
<td>VCMPLE_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 11H</td>
</tr>
<tr>
<td>VCMPPNORD_OPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 12H</td>
</tr>
<tr>
<td>VCMPPNLT_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 13H</td>
</tr>
<tr>
<td>VCMPPNLE_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 14H</td>
</tr>
<tr>
<td>VCMPPORD_OPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 15H</td>
</tr>
<tr>
<td>VCMPPTRUESPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 16H</td>
</tr>
<tr>
<td>VCMPEQ_SPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 17H</td>
</tr>
<tr>
<td>VCMPPGINQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 18H</td>
</tr>
<tr>
<td>VCMPPG_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 19H</td>
</tr>
<tr>
<td>VCMPPG_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1AH</td>
</tr>
<tr>
<td>VCMPPGEQ_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1BH</td>
</tr>
<tr>
<td>VCMPPG_TRUE_OPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1CH</td>
</tr>
<tr>
<td>VCMPPG_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1DH</td>
</tr>
<tr>
<td>VCMPPG_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1EH</td>
</tr>
<tr>
<td>VCMPPG_TRUE_OPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1FH</td>
</tr>
</tbody>
</table>
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP3 ← EQ_OQ; OP5 ← EQ_OQ;
  1: OP3 ← LT_OQ; OP5 ← LT_OQ;
  2: OP3 ← LE_OQ; OP5 ← LE_OQ;
  3: OP3 ← UNORD_Q; OP5 ← UNORD_Q;
  4: OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;
  5: OP3 ← NLT_US; OP5 ← NLT_US;
  6: OP3 ← NLE_US; OP5 ← NLE_US;
  7: OP3 ← ORD_Q; OP5 ← ORD_Q;
  8: OP5 ← EQ_UQ;
  9: OP5 ← NGE_US;
 10: OP5 ← NGT_US;
 11: OP5 ← FALSE_OQ;
 12: OP5 ← NEQ_OQ;
 13: OP5 ← GE_OQ;
 14: OP5 ← GT_OQ;
 15: OP5 ← TRUE_UQ;
 16: OP5 ← EQ_OQ;
 17: OP5 ← LT_OQ;
 18: OP5 ← LE_OQ;
 19: OP5 ← UNORD_S;
 20: OP5 ← NEQ_US;
 21: OP5 ← NLT_UQ;
 22: OP5 ← NLE_UQ;
 23: OP5 ← ORD_S;
 24: OP5 ← EQ_US;
 25: OP5 ← NGE_UQ;
 26: OP5 ← NGT_UQ;
 27: OP5 ← FALSE_OS;
 28: OP5 ← NEQ_OS;
 29: OP5 ← GE_OQ;
 30: OP5 ← GT_OQ;
 31: OP5 ← TRUE_US;
DEFAULT: Reserved
ESAC;
VCMPPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    CMP ← SRC1[i+31:i] OP5 SRC2[31:0]
                ELSE
                    CMP ← SRC1[i+31:i] OP5 SRC2[i+31:i]
                FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE DEST[j] ← 0; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VCMPPS (VEX.256 encoded version)
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
CMP4 ← SRC1[159:128] OP5 SRC2[159:128];
CMP5 ← SRC1[191:160] OP5 SRC2[191:160];
CMP6 ← SRC1[223:192] OP5 SRC2[223:192];
CMP7 ← SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 000000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] ← FFFFFFFFH;
    ELSE DEST[159:128] ← 000000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] ← FFFFFFFFH;
    ELSE DEST[191:160] ← 000000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] ← FFFFFFFFH;
    ELSE DEST[223:192] ← 000000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] ← FFFFFFFFH;
    ELSE DEST[255:224] ← 000000000H; FI;
DEST[MAXVL-1:256] ← 0
VCMPPS (VEX.128 encoded version)
CMPO ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
  THEN DEST[31:0] ← FFFFFFFFH;
  ELSE DEST[31:0] ← 000000000H; Fl;
IF CMP1 = TRUE
  THEN DEST[63:32] ← FFFFFFFFH;
  ELSE DEST[63:32] ← 000000000H; Fl;
IF CMP2 = TRUE
  THEN DEST[95:64] ← FFFFFFFFH;
  ELSE DEST[95:64] ← 000000000H; Fl;
IF CMP3 = TRUE
  THEN DEST[127:96] ← FFFFFFFFH;
  ELSE DEST[127:96] ← 000000000H; Fl;
DEST[MAXVL-1:128] ← 0

CMPPS (128-bit Legacy SSE version)
CMPO ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
  THEN DEST[31:0] ← FFFFFFFFH;
  ELSE DEST[31:0] ← 000000000H; Fl;
IF CMP1 = TRUE
  THEN DEST[63:32] ← FFFFFFFFH;
  ELSE DEST[63:32] ← 000000000H; Fl;
IF CMP2 = TRUE
  THEN DEST[95:64] ← FFFFFFFFH;
  ELSE DEST[95:64] ← 000000000H; Fl;
IF CMP3 = TRUE
  THEN DEST[127:96] ← FFFFFFFFH;
  ELSE DEST[127:96] ← 000000000H; Fl;
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCMPPS __mmask16 _mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 _mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPDP __mmask8 _mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 __mm256_cmp_ps( __m256 a, __m256 b, int imm)
CMPPS __m128 __mm_cmp_ps( __m128 a, __m128 b, int imm)

SIMD Floating-Point Exceptions
Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.
Denormal
Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6</td>
<td>CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPS m16, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPS m32, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R</td>
</tr>
<tr>
<td>REX.W + A7</td>
<td>CMPS m64, m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compares quadword at address (R</td>
</tr>
<tr>
<td>A6</td>
<td>CMPSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R</td>
</tr>
<tr>
<td>REX.W + A7</td>
<td>CMPSQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compares quadword at address (R</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operand form is provided to allow documentation. However, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords, quadwords), but they do not have to specify the correct loca-
Locations of the source operands are always specified by the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E/R)SI and (E/R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E/R)SI and (E/R)DI register increment; if the DF flag is 1, the registers decrement.) The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See "REP/REPE/REPN /REPNZ—Repeat String Operation Prefix" in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, for a description of the REP prefix.

In 64-bit mode, the instruction's default address size is 64 bits, 32 bit address size is supported using the prefix 67H. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[
\text{temp} \leftarrow \text{SRC1} - \text{SRC2};
\]

SetStatusFlags(temp);

IF (64-Bit Mode)
THEN
IF (Byte comparison)
THEN IF DF = 0
THEN
\( (R|E)SI \leftarrow (R|E)SI + 1; \)
\( (R|E)DI \leftarrow (R|E)DI + 1; \)
ELSE
\( (R|E)SI \leftarrow (R|E)SI - 1; \)
\( (R|E)DI \leftarrow (R|E)DI - 1; \)
FI;
ELSE IF (Word comparison)
THEN IF DF = 0
THEN
\( (R|E)SI \leftarrow (R|E)SI + 2; \)
\( (R|E)DI \leftarrow (R|E)DI + 2; \)
ELSE
\( (R|E)SI \leftarrow (R|E)SI - 2; \)
\( (R|E)DI \leftarrow (R|E)DI - 2; \)
FI;
ELSE IF (Doubleword comparison)
THEN IF DF = 0
THEN
\( (R|E)SI \leftarrow (R|E)SI + 4; \)
\( (R|E)DI \leftarrow (R|E)DI + 4; \)
ELSE
\( (R|E)SI \leftarrow (R|E)SI - 4; \)
\( (R|E)DI \leftarrow (R|E)DI - 4; \)
FI;
ELSE (* Quadword comparison *)
   THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 8;
      (R|E)DI ← (R|E)DI + 8;
   ELSE
      (R|E)SI ← (R|E)SI - 8;
      (R|E)DI ← (R|E)DI - 8;
   FI;
   FI;
ELSE (* Non-64-bit Mode *)
   IF (byte comparison)
      THEN IF DF = 0
         THEN
            (E)SI ← (E)SI + 1;
            (E)DI ← (E)DI + 1;
         ELSE
            (E)SI ← (E)SI - 1;
            (E)DI ← (E)DI - 1;
         FI;
      ELSE IF (Word comparison)
         THEN IF DF = 0
            THEN
               (E)SI ← (E)SI + 2;
               (E)DI ← (E)DI + 2;
            ELSE
               (E)SI ← (E)SI - 2;
               (E)DI ← (E)DI - 2;
            FI;
         ELSE (* Doubleword comparison *)
            THEN IF DF = 0
               THEN
                  (E)SI ← (E)SI + 4;
                  (E)DI ← (E)DI + 4;
               ELSE
                  (E)SI ← (E)SI - 4;
                  (E)DI ← (E)DI - 4;
               FI;
            FI;
   FI;

Flags Affected
The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
      If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**CMPSD—Compare Scalar Double-Precision Floating-Point Value**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F C2 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare low double-precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F.WIG C2 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W1 C2 /r ib VCMPSD k1 [k2], xmm2, xmm3/m64{sae}, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Compares the low double-precision floating-point values in the second source operand and the first source operand and returns the results in of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the “greater-than”, “greater-than-or-equal”, “not-greater-than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison)
or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-6. Compiler should treat reserved Imm8 values as illegal syntax.

### Table 3-6. Pseudo-Op and CMPSD Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPLTSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPEQSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPNORDSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPNEQSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPNLTSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNLESQD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-7, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-7.

### Table 3-7. Pseudo-Op and VCMPSD Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>VCMPSD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMPLTSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPPORDSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPPNEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMPPNLTSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMPPNLESQD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMPPORDSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPEQ_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMPPNGESD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMPPNGTSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, OAH</td>
</tr>
<tr>
<td>VCMPPNALSED reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 0BH</td>
</tr>
<tr>
<td>VCMPPNEQ_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, OCH</td>
</tr>
<tr>
<td>VCMPPGESD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, ODH</td>
</tr>
</tbody>
</table>
Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

CASE (COMPARISON PREDICATE) OF
0: OP3 ← EQ_OQ; OP5 ← EQ_OQ;
1: OP3 ← LT_OS; OP5 ← LT_OS;
2: OP3 ← LE_OQ; OP5 ← LE_OQ;
3: OP3 ← UNORD_Q; OP5 ← UNORD_Q;
4: OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;
5: OP3 ← NLT_US; OP5 ← NLT_US;
6: OP3 ← NLE_US; OP5 ← NLE_US;
7: OP3 ← ORD_Q; OP5 ← ORD_Q;
8: OP5 ← EQ_UQ;
9: OP5 ← NGE_US;
10: OP5 ← NGT_US;
11: OP5 ← FALSE_OQ;
12: OP5 ← NEQ_UQ;
13: OP5 ← GE_OS;
14: OP5 ← GT_OQ;
15: OP5 ← TRUE_UQ;
16: OP5 ← EQ_OS;
17: OP5 ← LT_OQ;
18: OP5 ← LE_OQ;
19: OP5 ← UNORD_S;
20: OP5 ← NEQ_US;
21: OP5 ← NLT_UQ;
CMPSD—Compare Scalar Double-Precision Floating-Point Value

22: OP5 ← NLE_UQ;
23: OP5 ← ORD_S;
24: OP5 ← EQ_US;
25: OP5 ← NGE_UQ;
26: OP5 ← NGT_UQ;
27: OP5 ← FALSE_OS;
28: OP5 ← NEQ_OS;
29: OP5 ← GE_OQ;
30: OP5 ← GT_OQ;
31: OP5 ← TRUE_US;
DEFAULT: Reserved

ESAC;

VCMPSD (EVEX encoded version)
CMPO ← SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or *no writemask*
    THEN IF CMPO = TRUE
         THEN DEST[0] ← 1;
         ELSE DEST[0] ← 0; FI;
    ELSE DEST[0] ← 0 ; zeroing-masking only
    FI;
DEST[MAX_KL-1:1] ← 0

CMPSD (128-bit Legacy SSE version)
CMPO ← DEST[63:0] OP3 SRC[63:0];
IF CMPO = TRUE
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0000000000000000H; FI;
DEST[MAXVL-1:64] (Unmodified)

VCMPSD (VEX.128 encoded version)
CMPO ← SRC1[63:0] OP5 SRC2[63:0];
IF CMPO = TRUE
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0000000000000000H; FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCMPSD __mmask8 __mm_cmp_sd_mask(__m128d a, __m128d b, int imm);
VCMPSD __mmask8 __mm_cmp_round_sd_mask(__m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 __mm_mask_cmp_sd_mask(__mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 __mm_mask_cmp_round_sd_mask(__mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d __mm_cmp_sd(__m128d a, __m128d b, const int imm)

SIMD Floating-Point Exceptions
Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1 Denormal.

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
CMPSS—Compare Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare low single-precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W0 C2 /r ib VCMPSS k1 [k2], xmm2, xmm3/m32{sae}, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Compares the low single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 128:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either
by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

### Table 3-8. Pseudo-Op and CMPSS Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPFLTSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPLESS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPUNORDSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPEQSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPNLTSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNEQSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 3-9, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSS instructions in a similar fashion by extending the syntax listed in Table 3-9.

### Table 3-9. Pseudo-Op and VCMPSS Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPEQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMPFLTSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPLESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPUNORDSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPEQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMPNLTSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMPNEQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMPORDSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPEQ_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMPNGESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMPNGTSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0AH</td>
</tr>
<tr>
<td>VCMPPFALSESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0BH</td>
</tr>
<tr>
<td>VCMPEQ_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0CH</td>
</tr>
<tr>
<td>VCMPPGESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0DH</td>
</tr>
</tbody>
</table>
Software should ensure VCMPSS is encoded with VEX.L=0. Encoding VCMPSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

CASE (COMPARISON PREDICATE) OF
- 0: OP3 ← EQ_OQ; OP5 ← EQ_OQ;
- 1: OP3 ← LT_OS; OP5 ← LT_OS;
- 2: OP3 ← LE_OS; OP5 ← LE_OS;
- 3: OP3 ← UNORD_Q; OP5 ← UNORD_Q;
- 4: OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;
- 5: OP3 ← NLT_US; OP5 ← NLT_US;
- 6: OP3 ← NLE_US; OP5 ← NLE_US;
- 7: OP3 ← ORD_Q; OP5 ← ORD_Q;
- 8: OP5 ← EQ_UQ;
- 9: OP5 ← NGE_US;
- 10: OP5 ← NLT_US;
- 11: OP5 ← FALSE_OQ;
- 12: OP5 ← NEQ_UQ;
- 13: OP5 ← GE_OS;
- 14: OP5 ← GT_OS;
- 15: OP5 ← TRUE_UQ;
- 16: OP5 ← EQ_OS;
- 17: OP5 ← LT_OS;
- 18: OP5 ← LE_OS;
- 19: OP5 ← UNORD_S;
- 20: OP5 ← NEQ_US;
- 21: OP5 ← NLT_UQ;

**Table 3-9. Pseudo-Op and VCMPSS Implementation**

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPGTSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0EH</td>
</tr>
<tr>
<td>VCMPTRUESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0FH</td>
</tr>
<tr>
<td>VCMPEQ_OSSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0EH</td>
</tr>
<tr>
<td>VCMPGT_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1NH</td>
</tr>
<tr>
<td>CMPLE_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 12H</td>
</tr>
<tr>
<td>VCMPEQ_OSSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 13H</td>
</tr>
<tr>
<td>VCMPGT_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 14H</td>
</tr>
<tr>
<td>VCMPNLT_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 15H</td>
</tr>
<tr>
<td>VCMPLT_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 16H</td>
</tr>
<tr>
<td>VCMPOEQ_USSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 17H</td>
</tr>
<tr>
<td>VCMPNLT_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 18H</td>
</tr>
<tr>
<td>VCMPGT_SSSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 19H</td>
</tr>
<tr>
<td>VCMPGE_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1AH</td>
</tr>
<tr>
<td>VCMPGT_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1BH</td>
</tr>
<tr>
<td>VCMPGE_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1CH</td>
</tr>
<tr>
<td>VCMPEQ_USSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1DH</td>
</tr>
<tr>
<td>VCMPEQ_USSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1FH</td>
</tr>
</tbody>
</table>
22: OP5 ← NLE_UQ;
23: OP5 ← ORD_S;
24: OP5 ← EQ_US;
25: OP5 ← NGE_UQ;
26: OP5 ← NGT_UQ;
27: OP5 ← FALSE_OS;
28: OP5 ← NEQ_OS;
29: OP5 ← GE_OQ;
30: OP5 ← GT_OQ;
31: OP5 ← TRUE_US;
DEFAULT: Reserved
ESAC;

VCMPSS (EVEX encoded version)
CMPO ← SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or *no writemask*
   THEN IF CMPO = TRUE
       THEN DEST[0] ← 1;
       ELSE DEST[0] ← 0; FI;
   ELSE DEST[0] ← 0; zeroing-masking only
   FI;
DEST[MAX_KL-1:1] ← 0

VCMPSS (128-bit Legacy SSE version)
CMPO ← DEST[31:0] OP3 SRC[31:0];
IF CMPO = TRUE
   THEN DEST[31:0] ← FFFFFFFFH;
   ELSE DEST[31:0] ← 00000000H; FI;
DEST[MAXVL-1:32] (Unmodified)

VCMPSS (VEX.128 encoded version)
CMPO ← SRC1[31:0] OP5 SRC2[31:0];
IF CMPO = TRUE
   THEN DEST[31:0] ← FFFFFFFFH;
   ELSE DEST[31:0] ← 00000000H; FI;
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCMPSS __mmask8 __m128 _mm_cmp_ss_mask(__m128 a, __m128 b, int imm);
VCMPSS __mmask8 __m128 _mm_cmp_round_ss_mask(__m128 a, __m128 b, int imm, int sae);
VCMPSS __mmask8 __m128 _mm_mask_cmp_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __mmask8 __m128 _mm_mask_cmp_round_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __m128 __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions
Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
# CMPXCHG—Compare and Exchange

## Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor’s bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

## Operation

(\* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed \*)

```
TEMP ← DEST
IF accumulator = TEMP
    THEN
        ZF ← 1;
        DEST ← SRC;
    ELSE
        ZF ← 0;
        accumulator ← TEMP;
        DEST ← TEMP;
FI;
```
Flags Affected
The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

Protected Mode Exceptions
- #GP(0) If the destination is located in a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions
- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

**Description**
Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64 bits of a 128-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see page 3-209.

**IA-32 Architecture Compatibility**
This instruction encoding is not supported on Intel processors earlier than the Pentium processors.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C7 71 m64</td>
<td>M</td>
<td>Valid</td>
<td>Valid*</td>
<td>Compare EDX:EAX with m64. If equal, set ZF and load ECX:EBX into m64. Else, clear ZF and load m64 into EDX:EAX.</td>
</tr>
<tr>
<td>CMPXCHG8B m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REX.W + 0F C7 71 m128</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare RDX:RAX with m128. If equal, set ZF and load RCX:RBX into m128. Else, clear ZF and load m128 into RDX:RAX.</td>
</tr>
<tr>
<td>CMPXCHG16B m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
*See IA-32 Architecture Compatibility section below.*

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Operation

IF (64-Bit Mode and OperandSize = 64)
    THEN
        TEMP128 ← DEST
        IF (RDX:RAX = TEMP128)
            THEN
                ZF ← 1;
                DEST ← RCX:RBX;
            ELSE
                ZF ← 0;
                RDX:RAX ← TEMP128;
                DEST ← TEMP128;
            FI;
        FI
    ELSE
        TEMP64 ← DEST;
        IF (EDX:EAX = TEMP64)
            THEN
                ZF ← 1;
                DEST ← ECX:EBX;
            ELSE
                ZF ← 0;
                EDX:EAX ← TEMP64;
                DEST ← TEMP64;
            FI;
        FI;
    FI;

Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Protected Mode Exceptions

#UD If the destination is not a memory operand.
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD If the destination operand is not a memory location.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
Virtual-8086 Mode Exceptions

#UD If the destination operand is not a memory location.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary.
If CPUID.01H:ECX.CMPXCHG16B[bit 13] = 0.
#UD If the destination operand is not a memory location.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2F /r COMISD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F:WIG 2F /r VCOMISD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F:W1 2F /r VCOMISD xmm1, xmm2/m64{sa}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Compares the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

**COMISD (all versions)**

\[
\text{RESULT} \leftarrow \text{OrderedCompare(DEST[63:0] <> SRC[63:0])}
\]

(* Set EFLAGS *) CASE (RESULT) OF

- UNORDERED: ZF,PF,CF \( \leftarrow 111; \)
- GREATER_THAN: ZF,PF,CF \( \leftarrow 000; \)
- LESS_THAN: ZF,PF,CF \( \leftarrow 001; \)
- EQUAL: ZF,PF,CF \( \leftarrow 100; \)

ESAC;

OF, AF, SF \( \leftarrow 0; \)


Intel C/C++ Compiler Intrinsic Equivalent
VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd(__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd(__m128d a, __m128d b)
VCOMISD int _mm_comile_sd(__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd(__m128d a, __m128d b)
VCOMISD int _mm_comige_sd(__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions
Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3;
EVEX-encoded instructions, see Exceptions Type E3NF.
#UD If VEX.vvv != 111B or EVEX.vvv != 111B.
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>VEX.1G.0F.W1 2F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>EVEX.1G.0F.W0 2F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compares the single-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

**COMISS (all versions)**

RESULT ← OrderedCompare([DEST[31:0] <> SRC[31:0]]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED: ZF,PF,CF ← 111;
  GREATER_THAN: ZF,PF,CF ← 000;
  LESS_THAN: ZF,PF,CF ← 001;
  EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0;}
**Intel C/C++ Compiler Intrinsic Equivalent**

- `VCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);`
- `VCOMISS int _mm_comieq_ss (__m128 a, __m128 b)`
- `VCOMISS int _mm_comilt_ss (__m128 a, __m128 b)`
- `VCOMISS int _mm_comile_ss (__m128 a, __m128 b)`
- `VCOMISS int _mm_comigt_ss (__m128 a, __m128 b)`
- `VCOMISS int _mm_comige_ss (__m128 a, __m128 b)`
- `VCOMISS int _mm_comineq_ss (__m128 a, __m128 b)`

**SIMD Floating-Point Exceptions**

Invalid (if SNaN or QNaN operands), Denormal.

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CPUID—CPU Identification

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF A2</td>
<td>CPUID</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers. The instruction’s output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

- CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
- CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
- CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)
- CPUID.EAX = 0CH (* INVALID: Returns the same information as CPUID.EAX = 0BH. *)
- CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
- CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

"Serializing Instructions” in Chapter 8, "Multiple-Processor Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

"Caching Translation Information” in Chapter 4, “Paging,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

---

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

3-190 Vol. 2A

CPUID—CPU Identification
### Table 3-8. Information Returned by CPUID Instruction

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0H</strong></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td>Maximum Input Value for Basic CPUID Information.</td>
</tr>
<tr>
<td>EBX</td>
<td>&quot;Genu&quot;</td>
</tr>
<tr>
<td>ECX</td>
<td>&quot;Intel&quot;</td>
</tr>
<tr>
<td>EDX</td>
<td>&quot;Intel&quot;</td>
</tr>
<tr>
<td><strong>01H</strong></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td>Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).</td>
</tr>
</tbody>
</table>
| EBX               | Bits 07 - 00: Brand Index. Bits 15 - 08: CLFLUSH line size (Value * 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23 - 16: Maximum number of addressable IDs for logical processors in this physical package*.
| ECX               | Feature Information (see Figure 3-7 and Table 3-10). |
| EDX               | Feature Information (see Figure 3-8 and Table 3-11). |
| **NOTES:**        | * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1. |
| **02H**           |                                        |
| EAX               | Cache and TLB Information (see Table 3-12). |
| EBX               | Cache and TLB Information. |
| ECX               | Cache and TLB Information. |
| EDX               | Cache and TLB Information. |
| **03H**           |                                        |
| EAX               | Reserved. |
| EBX               | Reserved. |
| ECX               | Bits 00 - 31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) |
| EDX               | Bits 32 - 63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) |
| **NOTES:**        | Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature. |
| **CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.** |
| **Deterministic Cache Parameters Leaf** | |
| **04H**           |                                        |
| EAX               | Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved. |

**NOTES:**

Leaf 04H output depends on the initial value in ECX.*

See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 217.
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial EAX Value</strong></td>
<td><strong>Information Provided about the Processor</strong></td>
</tr>
<tr>
<td>Bits 07 - 05: Cache Level (starts at 1). Bit 08: Self Initializing cache level (does not need SW initialization). Bit 09: Fully Associative cache.</td>
<td>Bits 13 - 10: Reserved. Bits 25 - 14: Maximum number of addressable IDs for logical processors sharing this cache**; *<strong>. Bits 31 - 26: Maximum number of addressable IDs for processor cores in the physical package</strong>; ****; *****.</td>
</tr>
<tr>
<td>EBX Bits 11 - 00: L = System Coherency Line Size**. Bits 21 - 12: P = Physical Line partitions**. Bits 31 - 22: W = Ways of associativity**.</td>
<td>ECX Bits 31-00: S = Number of Sets**.</td>
</tr>
<tr>
<td>EDX Bit 00: Write-Back Invalidate/Invalidate. 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache. 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 02: Complex Cache Indexing. 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31 - 03: Reserved = 0.</td>
<td>NOTES: * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0. ** Add one to the return value to get the result. *** The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache. **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</td>
</tr>
<tr>
<td><strong>MONITOR/MWAIT Leaf</strong></td>
<td></td>
</tr>
<tr>
<td>05H</td>
<td>EAX Bits 15 - 00: Smallest monitor-line size in bytes (default is processor’s monitor granularity). Bits 31 - 16: Reserved = 0. EBX Bits 15 - 00: Largest monitor-line size in bytes (default is processor’s monitor granularity). Bits 31 - 16: Reserved = 0. ECX Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported. Bit 01: Supports treating interrupts as break-event for MWait, even when interrupts disabled. Bits 31 - 02: Reserved.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| EDX               | Bits 03 - 00: Number of C0* sub C-states supported using MWAIT.  
|                   | Bits 07 - 04: Number of C1* sub C-states supported using MWAIT.  
|                   | Bits 11 - 08: Number of C2* sub C-states supported using MWAIT.  
|                   | Bits 15 - 12: Number of C3* sub C-states supported using MWAIT.  
|                   | Bits 19 - 16: Number of C4* sub C-states supported using MWAIT.  
|                   | Bits 23 - 20: Number of C5* sub C-states supported using MWAIT.  
|                   | Bits 27 - 24: Number of C6* sub C-states supported using MWAIT.  
|                   | Bits 31 - 28: Number of C7* sub C-states supported using MWAIT.  
| NOTE:            | * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states. |

#### Thermal and Power Management Leaf

| 06H EAX          | Bit 00: Digital temperature sensor is supported if set.  
|                  | Bit 01: Intel Turbo Boost Technology available (see description of IA32_MISC_ENABLE[38]).  
|                  | Bit 02: ARAT. APIC-Timer-always-running feature is supported if set.  
|                  | Bit 03: Reserved.  
|                  | Bit 04: PLN. Power limit notification controls are supported if set.  
|                  | Bit 05: ECMO. Clock modulation duty cycle extension is supported if set.  
|                  | Bit 06: PTM. Package thermal management is supported if set.  
|                  | Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set.  
|                  | Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set.  
|                  | Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set.  
|                  | Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set.  
|                  | Bit 12: Reserved.  
|                  | Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREADSTALL MSR is supported if set.  
|                  | Bit 14: Intel Turbo Boost Max Technology 3.0 available.  
|                  | Bit 15: HWP Capabilities. Highest Performance change is supported if set.  
|                  | Bit 16: HWP PECl override is supported if set.  
|                  | Bit 17: Flexible HWP is supported if set.  
|                  | Bit 18: Fast access mode for the IA32_HWP_REQUEST MSR is supported if set.  
|                  | Bit 19: Reserved.  
|                  | Bit 20: Ignoring Idle Logical Processor HWP request is supported if set.  
|                  | Bits 31 - 21: Reserved.  
| EBX               | Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor.  
|                  | Bits 31 - 04: Reserved.  
| ECX               | Bits 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency.  
|                  | Bits 02 - 01: Reserved = 0.  
|                  | Bits 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H).  
|                  | Bits 31 - 04: Reserved = 0.  
| EDX               | Reserved = 0.  

CPUID—CPU Identification

Vol. 2A 3-193
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>07H</strong></td>
<td>Sub-leaf 0 (Input ECX = 0). *</td>
</tr>
</tbody>
</table>

**EAX**
- Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.
- Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/wRFSBASE/wRGSBASE if 1.
- Bit 01: IA32_TSC_ADJUST MSR is supported if 1.
- Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1.
- Bit 03: BMI1.
- Bit 04: HLE.
- Bit 05: AVX2.
- Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1.
- Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1.
- Bit 08: BMI2.
- Bit 09: Supports Enhanced REP MOVSB/STOSB if 1.
- Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers.
- Bit 11: RTM.
- Bit 13: Deprecated FPU CS and FPU DS values if 1.
- Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1.
- Bit 16: AVX512F.
- Bit 17: AVX512FDQ.
- Bit 18: RDSEED.
- Bit 19: ADX.
- Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1.
- Bit 21: AVX512_IFMA.
- Bit 22: Reserved.
- Bit 23: CLFLUSHOPT.
- Bit 24: CLWB.
- Bit 25: Intel Processor Trace.
- Bit 26: AVX512PF. (Intel® Xeon Phi™ only.)
- Bit 27: AVX512ER. (Intel® Xeon Phi™ only.)
- Bit 28: AVX512CD.
- Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1.
- Bit 30: AVX512BW.
- Bit 31: AVX512VL.

**EBX**
- Bit 00: PREFETCHWT1. (Intel® Xeon Phi™ only.)
- Bit 01: AVX512_VBMI.
- Bit 02: UMIP. Supports user-mode instruction prevention if 1.
- Bit 03: PKU. Supports protection keys for user-mode pages if 1.
- Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRUR/WRPKRU instructions).
- Bits 16 - 5: Reserved.
- Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode.
- Bit 22: RDPID and IA32_TSC_AUX are available if 1.
- Bits 29 - 23: Reserved.
- Bit 30: SGX_LA. Supports SGX Launch Configuration if 1.
- Bit 31: Reserved.
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>NOTE:</strong></td>
<td>* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.</td>
</tr>
</tbody>
</table>

#### Direct Cache Access Information Leaf

<table>
<thead>
<tr>
<th>09H EAX</th>
<th>Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

#### Architectural Performance Monitoring Leaf

<table>
<thead>
<tr>
<th>0AH EAX</th>
<th>Bits 07 - 00: Version ID of architectural performance monitoring.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor.</td>
</tr>
<tr>
<td></td>
<td>Bits 23 - 16: Bit width of general-purpose, performance monitoring counter.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 00: Core cycle event not available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 01: Instruction retired event not available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 02: Reference cycles event not available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 03: Last-level cache reference event not available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 04: Last-level cache misses event not available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 05: Branch instruction retired event not available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 06: Branch mispredict retired event not available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 07: Reserved = 0.</td>
</tr>
<tr>
<td>ECX</td>
<td>Reserved = 0.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 04 - 00: Number of fixed-function performance counters (if Version ID &gt; 1).</td>
</tr>
<tr>
<td></td>
<td>Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID &gt; 1).</td>
</tr>
<tr>
<td></td>
<td>Bits 14 - 13: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>Bit 15: AnyThread deprecation.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 16: Reserved = 0.</td>
</tr>
</tbody>
</table>

#### Extended Topology Enumeration Leaf

<table>
<thead>
<tr>
<th>0BH EAX</th>
<th>Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All logical processors with the same next level ID share current level.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 05: Reserved.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 16: Reserved.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| ECX               | Bits 07 - 00: Level number. Same value in ECX input.  
                          Bits 15 - 08: Level type***.  
                          Bits 31 - 16: Reserved.  
| EDX               | Bits 31- 00: x2APIC ID the current logical processor.  
                          **NOTES:**  
                          * Software should use this field (EAX[4:0]) to enumerate processor topology of the system.  
                          ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.  
                          *** The value of the “level type” field is not related to level numbers in any way, higher “level type” values do not mean higher levels. Level type field has the following encoding:  
                          0: Invalid.  
                          1: SMT.  
                          2: Core.  
                          3-255: Reserved.  

### Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)

| ODH | NOTES:  
        Leaf 0DH main leaf (ECX = 0).  
| EAX | Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1.  
                          Bit 00: x87 state.  
                          Bit 01: SSE state.  
                          Bit 02: AVX state.  
                          Bits 04 - 03: MPX state.  
                          Bits 07 - 05: AVX-512 state.  
                          Bit 08: Used for IA32_XSS.  
                          Bit 09: PKRU state.  
                          Bits 12 - 10: Reserved.  
                          Bit 13: Used for IA32_XSS.  
                          Bits 31 - 14: Reserved.  
| EBX | Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.  
| ECX | Bit 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0.  
| EDX | Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1.  
                          Bits 31 - 00: Reserved.  

### Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)

| ODH | EAX | Bit 00: XSAVEOPT is available.  
                          Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set.  
                          Bit 02: Supports XGETBV with ECX = 1 if set.  
                          Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set.  
                          Bits 31 - 04: Reserved.  
| EBX | Bits 31 - 00: The size in bytes of the XSAVE area containing all states enabled by XCR0 | IA32_XSS.
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ECX</strong></td>
<td>Bits 31 - 00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07 - 00: Used for XCR0. Bit 08: PT state. Bit 09: Used for XCR0. Bits 12 - 10: Reserved. Bit 13: HWP state. Bits 31 - 14: Reserved.</td>
</tr>
<tr>
<td><strong>EDX</strong></td>
<td>Bits 31 - 00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.</td>
</tr>
</tbody>
</table>

**Processor Extended State Enumeration Sub-leaves (EAX = ODH, ECX = n, n > 1)**

- **0DH**
  - **NOTES:** Leaf ODH output depends on the initial value in ECX.
  - Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR.
  - *If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (0 ≤ n ≤ 31) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (32 ≤ n ≤ 63) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].

| EAX | Bits 31 - 0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, n. |
| EBX | Bits 31 - 0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCR0 register*. |
| ECX | Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCR0. Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31 - 02 are reserved. This field reports 0 if the sub-leaf index, n, is invalid*. |
| EDX | This field reports 0 if the sub-leaf index, n, is invalid*; otherwise it is reserved. |

**Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = OFH, ECX = 0)**

- **NOTES:** Leaf OFH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.

| EAX | Reserved. |
| EBX | Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types. |
| ECX | Reserved. |
| EDX | Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31 - 02: Reserved. |

**L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = OFH, ECX = 1)**

- **NOTES:** Leaf OFH output depends on the initial value in ECX.

<p>| EAX | Reserved. |</p>
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).</td>
</tr>
<tr>
<td>ECX</td>
<td>Maximum range (zero-based) of RMID of this resource type.</td>
</tr>
</tbody>
</table>
| EDX | Bit 00: Supports L3 occupancy monitoring if 1.  
| | Bit 01: Supports L3 Total Bandwidth monitoring if 1.  
| | Bit 02: Supports L3 Local Bandwidth monitoring if 1.  
| | Bits 31 - 03: Reserved. |

**Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)**

**NOTES:**
- Leaf 10H output depends on the initial value in ECX.
- Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.

**EAX**
Reserved.

**EBX**
Bit 00: Reserved.  
Bit 01: Supports L3 Cache Allocation Technology if 1.  
Bit 02: Supports L2 Cache Allocation Technology if 1.  
Bit 03: Supports Memory Bandwidth Allocation if 1.  
Bits 31 - 04: Reserved.

**ECX**
Reserved.

**EDX**
Reserved.

**L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)**

**NOTES:**
- Leaf 10H output depends on the initial value in ECX.

**EAX**
Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation.  
Bits 31 - 05: Reserved.

**EBX**
Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.

**ECX**
Bits 01 - 00: Reserved.  
Bit 02: Code and Data Prioritization Technology supported if 1.  
Bits 31 - 03: Reserved.

**EDX**
Bits 15 - 00: Highest COS number supported for this ResID.  
Bits 31 - 16: Reserved.

**L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =2)**

**NOTES:**
- Leaf 10H output depends on the initial value in ECX.

**EAX**
Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation.  
Bits 31 - 05: Reserved.

**EBX**
Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.

**ECX**
Bits 31 - 00: Reserved.

**EDX**
Bits 15 - 00: Highest COS number supported for this ResID.  
Bits 31 - 16: Reserved.

**Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)**

**NOTES:**
- Leaf 10H output depends on the initial value in ECX.
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| EAX               | Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID using minus-one notation.  
                        Bits 31 - 12: Reserved. |
| EBX               | Bits 31 - 00: Reserved. |
| ECX               | Bits 01 - 00: Reserved.  
                        Bit 02: Reports whether the response of the delay values is linear.  
                        Bits 31 - 03: Reserved. |
| EDX               | Bits 15 - 00: Highest COS number supported for this ResID.  
                        Bits 31 - 16: Reserved. |

**Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)**

12H

**NOTES:**
Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.

**EAX**
Bit 00: SGX1.  
If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions.  
Bit 01: SGX2.  
If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions.  
Bits 04 - 02: Reserved.  
Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVIRTCHILD, EDECVIRTCHILD, and ESETCONTEXT.  
Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC.  
Bits 31 - 02: Reserved.

**EBX**
Bits 31 - 00: MISCESELECT. Bit vector of supported extended SGX features.

**ECX**
Bits 31 - 00: Reserved.

**EDX**
Bits 07 - 00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is $2^{(EDX[7:0])}$.  
Bits 15 - 08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is $2^{(EDX[15:8])}$.  
Bits 31 - 16: Reserved.

**Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)**

12H

**NOTES:**
Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.

**EAX**
Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.

**EBX**
Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.

**ECX**
Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.

**EDX**
Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.

**Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)**

12H

**NOTES:**
Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.  
For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EAX[31:4] depends on the sub-leaf type listed below.

**EAX**
Bit 03 - 00: Sub-leaf Type  
0000b: Indicates this sub-leaf is invalid.  
0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section.  
All other type encodings are reserved.

**Type**
0000b. This sub-leaf is invalid.  
EDX:ECX:EBX:EAX return 0.
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows.</td>
<td><strong>NOTES:</strong></td>
</tr>
<tr>
<td>EAX[11:04]: Reserved (enumerate 0).</td>
<td>Leaf 14H main leaf (ECX = 0).</td>
</tr>
<tr>
<td>EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section.</td>
<td>EAX Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.</td>
</tr>
<tr>
<td>EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section.</td>
<td>EBX Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed.</td>
</tr>
<tr>
<td>EBX[31:20]: Reserved.</td>
<td>Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode.</td>
</tr>
<tr>
<td>ECX[03:00]: EPC section property encoding defined as follows:</td>
<td>Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset.</td>
</tr>
<tr>
<td>If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0.</td>
<td>Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets.</td>
</tr>
<tr>
<td>If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encodings are reserved.</td>
<td>Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets.</td>
</tr>
<tr>
<td>ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.</td>
<td>Bit 31 - 06: Reserved.</td>
</tr>
<tr>
<td>EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory.</td>
<td>ECX Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed.</td>
</tr>
<tr>
<td>EDX[31:20]: Reserved.</td>
<td>Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS.</td>
</tr>
</tbody>
</table>

**Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)**

<table>
<thead>
<tr>
<th>EAX</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14H</td>
<td>Notes:</td>
</tr>
<tr>
<td>EAX</td>
<td>Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 31 - 06: Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 02: If 1, indicates support of Single-Range Output scheme.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 03: If 1, indicates support of output to Trace Transport subsystem.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 30 - 04: Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 31 - 00: Reserved.</td>
</tr>
</tbody>
</table>

**Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)**

<table>
<thead>
<tr>
<th>EAX</th>
<th>Bits 02 - 00: Number of configurable Address Ranges for filtering.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Bits 15 - 03: Reserved.</td>
</tr>
<tr>
<td>EAX</td>
<td>Bits 31 - 16: Bitmap of supported MTC period encodings.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 15 - 00: Bitmap of supported Cycle Threshold value encodings.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 31 - 16: Bitmap of supported Configurable PSB frequency encodings.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 31 - 00: Reserved.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>Bits 31 - 00: Reserved.</td>
</tr>
</tbody>
</table>

#### Time Stamp Counter and Nominal Core Crystal Clock Information Leaf

<table>
<thead>
<tr>
<th>15H</th>
<th>NOTES:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If EBX[31:0] is 0, the TSC/&quot;core crystal clock&quot; ratio is not enumerated.</td>
</tr>
<tr>
<td></td>
<td>EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency.</td>
</tr>
<tr>
<td></td>
<td>If ECX is 0, the nominal core clock frequency is not enumerated.</td>
</tr>
<tr>
<td></td>
<td>“TSC frequency” = “core crystal clock frequency” * EBX/EAX.</td>
</tr>
<tr>
<td></td>
<td>The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.</td>
</tr>
<tr>
<td>EAX</td>
<td>Bits 31 - 00: An unsigned integer which is the denominator of the TSC/&quot;core crystal clock&quot; ratio.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 31 - 00: An unsigned integer which is the numerator of the TSC/&quot;core crystal clock&quot; ratio.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 31 - 00: An unsigned integer which is the nominal frequency of the core clock in Hz.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 31 - 00: Reserved = 0.</td>
</tr>
</tbody>
</table>

#### Processor Frequency Information Leaf

<table>
<thead>
<tr>
<th>16H</th>
<th>EAX</th>
<th>Bits 15 - 00: Processor Base Frequency (in MHz).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bits 31 - 16: Reserved = 0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EBX</td>
<td>Bits 15 - 00: Maximum Frequency (in MHz).</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 16: Reserved = 0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECX</td>
<td>Bits 15 - 00: Bus (Reference) Frequency (in MHz).</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 16: Reserved = 0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDX</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

NOTES:
* Data is returned from this interface in accordance with the processor’s specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor.brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces.

While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.

#### System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)

<table>
<thead>
<tr>
<th>17H</th>
<th>NOTES:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Leaf 17H main leaf (ECX = 0).</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H output depends on the initial value in ECX.</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String.</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H is valid if MaxSOCID_Index &gt;= 3.</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H sub-leaves 4 and above are reserved.</td>
</tr>
<tr>
<td>EAX</td>
<td>Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 15 - 00: SOC Vendor ID.</td>
</tr>
<tr>
<td></td>
<td>Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 17: Reserved = 0.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.</td>
</tr>
</tbody>
</table>
### System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>17H</td>
<td>Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.</td>
</tr>
<tr>
<td>EAX</td>
<td>Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.</td>
</tr>
</tbody>
</table>

**NOTES:**
- Leaf 17H output depends on the initial value in ECX.
- SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H.
- The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.

### System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>17H</td>
<td>Leaf 17H output depends on the initial value in ECX.</td>
</tr>
<tr>
<td>EAX</td>
<td>Bits 31 - 00: Reserved = 0.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 31 - 00: Reserved = 0.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 31 - 00: Reserved = 0.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 31 - 00: Reserved = 0.</td>
</tr>
</tbody>
</table>

### Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>18H</td>
<td>Each sub-leaf enumerates a different address translation structure.</td>
</tr>
<tr>
<td>EAX</td>
<td>Bits 31 - 00: Reports the maximum input value of supported sub-leaf in leaf 18H.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 00: 4K page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bit 01: 2MB page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bit 02: 4MB page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bit 03: 1 GB page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bits 07 - 04: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure).</td>
</tr>
<tr>
<td></td>
<td>Bits 15 - 11: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 16: W = Ways of associativity.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 31 - 00: S = Number of Sets.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>Bits 04 - 00: Translation cache type field.</td>
</tr>
<tr>
<td></td>
<td>00000b: Null (indicates this sub-leaf is not valid).</td>
</tr>
<tr>
<td></td>
<td>00001b: Data TLB.</td>
</tr>
<tr>
<td></td>
<td>00010b: Instruction TLB.</td>
</tr>
<tr>
<td></td>
<td>00011b: Unified TLB*.</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 07 - 05: Translation cache level (starts at 1).</td>
</tr>
<tr>
<td></td>
<td>Bit 08: Fully associative structure.</td>
</tr>
<tr>
<td></td>
<td>Bits 13 - 09: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache**</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 26: Reserved.</td>
</tr>
</tbody>
</table>

### Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)

#### 18H

**NOTES:**
- Each sub-leaf enumerates a different address translation structure.
- If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure.
- * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). Please see the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product.
- ** Add one to the return value to get the result.

<table>
<thead>
<tr>
<th>EAX</th>
<th>Bits 31 - 00: Reserved.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>Bit 00: 4K page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bit 01: 2MB page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bit 02: 4MB page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bit 03: 1 GB page size entries supported by this structure.</td>
</tr>
<tr>
<td></td>
<td>Bits 07 - 04: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure).</td>
</tr>
<tr>
<td></td>
<td>Bits 15 - 11: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 16: W = Ways of associativity.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 31 - 00: S = Number of Sets.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 04 - 00: Translation cache type field.</td>
</tr>
<tr>
<td></td>
<td>0000b: Null (indicates this sub-leaf is not valid).</td>
</tr>
<tr>
<td></td>
<td>0001b: Data TLB.</td>
</tr>
<tr>
<td></td>
<td>0010b: Instruction TLB.</td>
</tr>
<tr>
<td></td>
<td>0011b: Unified TLB*.</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 07 - 05: Translation cache level (starts at 1).</td>
</tr>
<tr>
<td></td>
<td>Bit 08: Fully associative structure.</td>
</tr>
<tr>
<td></td>
<td>Bits 13 - 09: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache**</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 26: Reserved.</td>
</tr>
</tbody>
</table>

### Unimplemented CPUID Leaf Functions

<p>| 40000000H - 4FFFFFFFH | Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH. |</p>
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000000H EAX</td>
<td>Maximum Input Value for Extended Function CPUID Information.</td>
</tr>
<tr>
<td></td>
<td>EBX Reserved.</td>
</tr>
<tr>
<td></td>
<td>ECX Reserved.</td>
</tr>
<tr>
<td></td>
<td>EDX Reserved.</td>
</tr>
<tr>
<td>80000001H EAX</td>
<td>Extended Processor Signature and Feature Bits.</td>
</tr>
<tr>
<td></td>
<td>EBX Reserved.</td>
</tr>
<tr>
<td></td>
<td>ECX Bit 00: LAHF/SAHF available in 64-bit mode.</td>
</tr>
<tr>
<td></td>
<td>Bits 04 - 01: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 05: LZCNT.</td>
</tr>
<tr>
<td></td>
<td>Bits 07 - 06: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 08: PREFETCHW.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 09: Reserved.</td>
</tr>
<tr>
<td></td>
<td>EDX Bits 10 - 00: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 11: SYSCALL/SYSRET available in 64-bit mode.</td>
</tr>
<tr>
<td></td>
<td>Bits 19 - 12: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>Bit 20: Execute Disable Bit available.</td>
</tr>
<tr>
<td></td>
<td>Bits 25 - 21: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>Bit 26: 1-GByte pages are available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 27: RDTSCP and IA32_TSC_AUX are available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 28: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>Bit 29: Intel® 64 Architecture available if 1.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 30: Reserved = 0.</td>
</tr>
<tr>
<td>80000002H EAX</td>
<td>Processor Brand String.</td>
</tr>
<tr>
<td></td>
<td>EBX Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>ECX Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>EDX Processor Brand String Continued.</td>
</tr>
<tr>
<td>80000003H EAX</td>
<td>Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>EBX Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>ECX Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>EDX Processor Brand String Continued.</td>
</tr>
<tr>
<td>80000004H EAX</td>
<td>Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>EBX Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>ECX Processor Brand String Continued.</td>
</tr>
<tr>
<td></td>
<td>EDX Processor Brand String Continued.</td>
</tr>
<tr>
<td>80000005H EAX</td>
<td>Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>EBX Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>ECX Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>EDX Reserved = 0.</td>
</tr>
<tr>
<td>80000006H EAX</td>
<td>Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>EBX Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>ECX Bits 07 - 00: Cache Line size in bytes.</td>
</tr>
<tr>
<td></td>
<td>Bits 11 - 08: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 15 - 12: L2 Associativity field *.</td>
</tr>
<tr>
<td></td>
<td>Bits 31 - 16: Cache size in 1K units.</td>
</tr>
<tr>
<td></td>
<td>EDX Reserved = 0.</td>
</tr>
</tbody>
</table>
INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

\[\begin{align*}
\text{EBX} & \leftarrow 756e6547h \quad (\text{"Genu", with } G \text{ in the low eight bits of BL}) \\
\text{EDX} & \leftarrow 49656e69h \quad (\text{"ineI", with } i \text{ in the low eight bits of DL}) \\
\text{ECX} & \leftarrow 6c65746eh \quad (\text{"ntel", with } n \text{ in the low eight bits of CL})
\end{align*}\]

INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-9 for available processor type values. Stepping IDs are provided as needed.

![Figure 3-6. Version Information Returned by CPUID in EAX](image)

**Table 3-9. Processor Type Field**

<table>
<thead>
<tr>
<th>Type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original OEM Processor</td>
<td>00B</td>
</tr>
<tr>
<td>Intel OverDrive Processor</td>
<td>01B</td>
</tr>
<tr>
<td>Dual processor (not applicable to Intel486 processors)</td>
<td>10B</td>
</tr>
<tr>
<td>Intel reserved</td>
<td>11B</td>
</tr>
</tbody>
</table>

**NOTE**

See Chapter 19 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for information on identifying earlier IA-32 processors.
The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

\[
\text{IF Family_ID} \neq 0FH \\
\quad \text{THEN DisplayFamily} = \text{Family_ID}; \\
\quad \text{ELSE DisplayFamily} = \text{Extended_Family_ID} + \text{Family_ID}; \\
\quad \text{(* Right justify and zero-extend 4-bit field. *)} \\
\text{FI}; \\
\text{(* Show DisplayFamily as HEX field. *)}
\]

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

\[
\text{IF (Family_ID} = 06H \text{ or Family_ID} = 0FH) \\
\quad \text{THEN DisplayModel} = (\text{Extended_Model_ID} \ll 4) + \text{Model_ID}; \\
\quad \text{(* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)} \\
\text{ELSE DisplayModel} = \text{Model_ID}; \\
\text{FI}; \\
\text{(* Show DisplayModel as HEX field. *)}
\]

**INPUT EAX = 01H: Returns Additional Information in EBX**

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

**INPUT EAX = 01H: Returns Feature Information in ECX and EDX**

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

**NOTE**

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.
Figure 3-7. Feature Information Returned in the ECX Register

Table 3-10. Feature Information Returned in the ECX Register

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SSE3</td>
<td>Streaming SIMD Extensions 3 (SSE3). A value of 1 indicates the processor supports this technology.</td>
</tr>
<tr>
<td>1</td>
<td>PCLMULQDQ</td>
<td>PCLMULQDQ. A value of 1 indicates the processor supports the PCLMULQDQ instruction.</td>
</tr>
<tr>
<td>2</td>
<td>DTES64</td>
<td>64-bit DS Area. A value of 1 indicates the processor supports DS area using 64-bit layout.</td>
</tr>
<tr>
<td>3</td>
<td>MONITOR</td>
<td>MONITOR/MWAIT. A value of 1 indicates the processor supports this feature.</td>
</tr>
<tr>
<td>4</td>
<td>DS-CPL</td>
<td>CPL Qualified Debug Store. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.</td>
</tr>
<tr>
<td>5</td>
<td>VMX</td>
<td>Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.</td>
</tr>
<tr>
<td>6</td>
<td>SMX</td>
<td>Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 6, “Safer Mode Extensions Reference”.</td>
</tr>
<tr>
<td>7</td>
<td>EIST</td>
<td>Enhanced Intel SpeedStep® technology. A value of 1 indicates that the processor supports this technology.</td>
</tr>
<tr>
<td>8</td>
<td>TM2</td>
<td>Thermal Monitor 2. A value of 1 indicates whether the processor supports this technology.</td>
</tr>
<tr>
<td>9</td>
<td>SSSE3</td>
<td>A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.</td>
</tr>
<tr>
<td>Bit #</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>10</td>
<td>CNXT-ID</td>
<td><strong>L1 Context ID.</strong> A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.</td>
</tr>
<tr>
<td>11</td>
<td>SDBG</td>
<td>A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.</td>
</tr>
<tr>
<td>12</td>
<td>FMA</td>
<td>A value of 1 indicates the processor supports FMA extensions using YMM state.</td>
</tr>
<tr>
<td>13</td>
<td>CMPXCHG16B</td>
<td><strong>CMPXCHG16B Available.</strong> A value of 1 indicates that the feature is available. See the &quot;CMPXCHG16B—Compare and Exchange Bytes&quot; section in this chapter for a description.</td>
</tr>
<tr>
<td>14</td>
<td>xTPR Update Control</td>
<td><strong>xTPR Update Control.</strong> A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].</td>
</tr>
<tr>
<td>15</td>
<td>PDCM</td>
<td><strong>Perfmon and Debug Capability:</strong> A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>17</td>
<td>PCID</td>
<td><strong>Process-context identifiers.</strong> A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.</td>
</tr>
<tr>
<td>18</td>
<td>DCA</td>
<td>A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.</td>
</tr>
<tr>
<td>19</td>
<td>SSE4.1</td>
<td>A value of 1 indicates that the processor supports SSE4.1.</td>
</tr>
<tr>
<td>20</td>
<td>SSE4.2</td>
<td>A value of 1 indicates that the processor supports SSE4.2.</td>
</tr>
<tr>
<td>21</td>
<td>x2APIC</td>
<td>A value of 1 indicates that the processor supports x2APIC feature.</td>
</tr>
<tr>
<td>22</td>
<td>MOVBE</td>
<td>A value of 1 indicates that the processor supports MOVBE instruction.</td>
</tr>
<tr>
<td>23</td>
<td>POPCNT</td>
<td>A value of 1 indicates that the processor supports the POPCNT instruction.</td>
</tr>
<tr>
<td>24</td>
<td>TSC-Deadline</td>
<td>A value of 1 indicates that the processor’s local APIC timer supports one-shot operation using a TSC deadline value.</td>
</tr>
<tr>
<td>25</td>
<td>AESNI</td>
<td>A value of 1 indicates that the processor supports the AESNI instruction extensions.</td>
</tr>
<tr>
<td>26</td>
<td>XSAVE</td>
<td>A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.</td>
</tr>
<tr>
<td>27</td>
<td>OSXSAVE</td>
<td>A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.</td>
</tr>
<tr>
<td>28</td>
<td>AVX</td>
<td>A value of 1 indicates that the processor supports the AVX instruction extensions.</td>
</tr>
<tr>
<td>29</td>
<td>F16C</td>
<td>A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.</td>
</tr>
<tr>
<td>30</td>
<td>RDRAND</td>
<td>A value of 1 indicates that processor supports RDRAND instruction.</td>
</tr>
<tr>
<td>31</td>
<td>Not Used</td>
<td>Always returns 0.</td>
</tr>
</tbody>
</table>
Figure 3-8. Feature Information Returned in the EDX Register
<table>
<thead>
<tr>
<th>Bit #</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FPU</td>
<td>Floating Point Unit On-Chip. The processor contains an x87 FPU.</td>
</tr>
<tr>
<td>1</td>
<td>VME</td>
<td>Virtual 8086 Mode Enhancements. Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirect, expansion of the TSS with the software indirect bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.</td>
</tr>
<tr>
<td>2</td>
<td>DE</td>
<td>Debugging Extensions. Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.</td>
</tr>
<tr>
<td>3</td>
<td>PSE</td>
<td>Page Size Extension. Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.</td>
</tr>
<tr>
<td>4</td>
<td>TSC</td>
<td>Time Stamp Counter. The RDTSC instruction is supported, including CR4.TSD for controlling privilege.</td>
</tr>
<tr>
<td>5</td>
<td>MSR</td>
<td>Model Specific Registers RDMSR and WRMSR Instructions. The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.</td>
</tr>
<tr>
<td>6</td>
<td>PAE</td>
<td>Physical Address Extension. Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.</td>
</tr>
<tr>
<td>7</td>
<td>MCE</td>
<td>Machine Check Exception. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.</td>
</tr>
<tr>
<td>8</td>
<td>CX8</td>
<td>CMPXCHG8B Instruction. The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).</td>
</tr>
<tr>
<td>9</td>
<td>APIC</td>
<td>APIC On-Chip. The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>SEP</td>
<td>SYSENTER and SYSEXIT Instructions. The SYSENTER and SYSEXIT and associated MSRs are supported.</td>
</tr>
<tr>
<td>12</td>
<td>MTRR</td>
<td>Memory Type Range Registers. MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.</td>
</tr>
<tr>
<td>13</td>
<td>PGE</td>
<td>Page Global Bit. The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.</td>
</tr>
<tr>
<td>14</td>
<td>MCA</td>
<td>Machine Check Architecture. A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.</td>
</tr>
<tr>
<td>15</td>
<td>CMOV</td>
<td>Conditional Move Instructions. The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported.</td>
</tr>
<tr>
<td>16</td>
<td>PAT</td>
<td>Page Attribute Table. Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.</td>
</tr>
<tr>
<td>17</td>
<td>PSE-36</td>
<td>36-Bit Page Size Extension. 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.</td>
</tr>
<tr>
<td>18</td>
<td>PSN</td>
<td>Processor Serial Number. The processor supports the 96-bit processor identification number feature and the feature is enabled.</td>
</tr>
<tr>
<td>19</td>
<td>CLFSH</td>
<td>CLFLUSH Instruction. CLFLUSH Instruction is supported.</td>
</tr>
<tr>
<td>20</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 3-11. More on Feature Information Returned in the EDX Register
When CPUID executes with EAX set to 02H, the processor returns information about the processor’s internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of “cache type” via CPUID leaf 2.

### Table 3-11. More on Feature Information Returned in the EDX Register (Contd.)

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>DS</td>
<td>Debug Store. The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and processor event-based sampling (PEBS) facilities (see Chapter 23, “Introduction to Virtual-Machine Extensions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C).</td>
</tr>
<tr>
<td>22</td>
<td>ACPI</td>
<td>Thermal Monitor and Software Controlled Clock Facilities. The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.</td>
</tr>
<tr>
<td>23</td>
<td>MMX</td>
<td>Intel MMX Technology. The processor supports the Intel MMX technology.</td>
</tr>
<tr>
<td>24</td>
<td>FXSR</td>
<td>FXSAVE and FXRSTOR Instructions. The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.</td>
</tr>
<tr>
<td>25</td>
<td>SSE</td>
<td>SSE. The processor supports the SSE extensions.</td>
</tr>
<tr>
<td>26</td>
<td>SSE2</td>
<td>SSE2. The processor supports the SSE2 extensions.</td>
</tr>
<tr>
<td>27</td>
<td>SS</td>
<td>Self Snoop. The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.</td>
</tr>
<tr>
<td>28</td>
<td>HTT</td>
<td>Max APIC IDs reserved field is Valid. A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.</td>
</tr>
<tr>
<td>29</td>
<td>TM</td>
<td>Thermal Monitor. The processor implements the thermal monitor automatic thermal control circuitry (TCC).</td>
</tr>
<tr>
<td>30</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>31</td>
<td>PBE</td>
<td>Pending Break Enable. The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.</td>
</tr>
</tbody>
</table>
### Table 3-12. Encoding of CPUID Leaf 2 Descriptors

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>General</td>
<td>Null descriptor, this byte contains no information</td>
</tr>
<tr>
<td>01H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries</td>
</tr>
<tr>
<td>02H</td>
<td>TLB</td>
<td>Instruction TLB: 4 MByte pages, fully associative, 2 entries</td>
</tr>
<tr>
<td>03H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte pages, 4-way set associative, 64 entries</td>
</tr>
<tr>
<td>04H</td>
<td>TLB</td>
<td>Data TLB: 4 MByte pages, 4-way set associative, 8 entries</td>
</tr>
<tr>
<td>05H</td>
<td>TLB</td>
<td>Data TLB1: 4 MByte pages, 4-way set associative, 32 entries</td>
</tr>
<tr>
<td>06H</td>
<td>Cache</td>
<td>1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>08H</td>
<td>Cache</td>
<td>1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>09H</td>
<td>Cache</td>
<td>1st-level instruction cache: 32 KBytes, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>0AH</td>
<td>Cache</td>
<td>1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>0BH</td>
<td>Cache</td>
<td>1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>0CH</td>
<td>Cache</td>
<td>1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>0EH</td>
<td>Cache</td>
<td>1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>1DH</td>
<td>Cache</td>
<td>2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>21H</td>
<td>Cache</td>
<td>2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>22H</td>
<td>Cache</td>
<td>3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>23H</td>
<td>Cache</td>
<td>3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>24H</td>
<td>Cache</td>
<td>3rd-level cache: 1 MBytes, 16-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>25H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>29H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>2CH</td>
<td>Cache</td>
<td>1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>30H</td>
<td>Cache</td>
<td>1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>40H</td>
<td>Cache</td>
<td>No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache</td>
</tr>
<tr>
<td>41H</td>
<td>Cache</td>
<td>2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>42H</td>
<td>Cache</td>
<td>2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>43H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>44H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>45H</td>
<td>Cache</td>
<td>2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>46H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>47H</td>
<td>Cache</td>
<td>3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>48H</td>
<td>Cache</td>
<td>2nd-level cache: 3 MByte, 12-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>49H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>4AH</td>
<td>Cache</td>
<td>3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>4BH</td>
<td>Cache</td>
<td>3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>4CH</td>
<td>Cache</td>
<td>3rd-level cache: 12 MByte, 12-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>4DH</td>
<td>Cache</td>
<td>3rd-level cache: 16 MByte, 16-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>4EH</td>
<td>Cache</td>
<td>2nd-level cache: 6 MByte, 24-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>4FH</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, 32 entries</td>
</tr>
</tbody>
</table>
Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>50H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries</td>
</tr>
<tr>
<td>51H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries</td>
</tr>
<tr>
<td>52H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries</td>
</tr>
<tr>
<td>55H</td>
<td>TLB</td>
<td>Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries</td>
</tr>
<tr>
<td>56H</td>
<td>TLB</td>
<td>Data TLB0: 4 MByte pages, 4-way set associative, 16 entries</td>
</tr>
<tr>
<td>57H</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte pages, 4-way associative, 16 entries</td>
</tr>
<tr>
<td>59H</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte pages, fully associative, 16 entries</td>
</tr>
<tr>
<td>5AH</td>
<td>TLB</td>
<td>Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries</td>
</tr>
<tr>
<td>5BH</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte and 4 MByte pages, 64 entries</td>
</tr>
<tr>
<td>5DH</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte and 4 MByte pages, 128 entries</td>
</tr>
<tr>
<td>5FH</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte and 4 MByte pages, 256 entries</td>
</tr>
<tr>
<td>60H</td>
<td>Cache</td>
<td>1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>61H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, fully associative, 48 entries</td>
</tr>
<tr>
<td>63H</td>
<td>TLB</td>
<td>Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries</td>
</tr>
<tr>
<td>64H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte pages, 4-way set associative, 512 entries</td>
</tr>
<tr>
<td>66H</td>
<td>Cache</td>
<td>1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>67H</td>
<td>Cache</td>
<td>1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>68H</td>
<td>Cache</td>
<td>1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>6AH</td>
<td>Cache</td>
<td>uTLB: 4 KByte pages, 8-way set associative, 64 entries</td>
</tr>
<tr>
<td>6BH</td>
<td>Cache</td>
<td>DTLB: 4 KByte pages, 8-way set associative, 256 entries</td>
</tr>
<tr>
<td>6CH</td>
<td>Cache</td>
<td>DTLB: 2M/4M pages, 8-way set associative, 128 entries</td>
</tr>
<tr>
<td>6DH</td>
<td>Cache</td>
<td>DTLB: 1 GByte pages, fully associative, 16 entries</td>
</tr>
<tr>
<td>70H</td>
<td>Cache</td>
<td>Trace cache: 12 K-μop, 8-way set associative</td>
</tr>
<tr>
<td>71H</td>
<td>Cache</td>
<td>Trace cache: 16 K-μop, 8-way set associative</td>
</tr>
<tr>
<td>72H</td>
<td>Cache</td>
<td>Trace cache: 32 K-μop, 8-way set associative</td>
</tr>
<tr>
<td>76H</td>
<td>TLB</td>
<td>Instruction TLB: 2M/4M pages, fully associative, 8 entries</td>
</tr>
<tr>
<td>78H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 4-way set associative, 64byte line size</td>
</tr>
<tr>
<td>79H</td>
<td>Cache</td>
<td>2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>7AH</td>
<td>Cache</td>
<td>2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>7BH</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>7CH</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector</td>
</tr>
<tr>
<td>7DH</td>
<td>Cache</td>
<td>2nd-level cache: 2 MByte, 8-way set associative, 64byte line size</td>
</tr>
<tr>
<td>7FH</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size</td>
</tr>
<tr>
<td>80H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size</td>
</tr>
<tr>
<td>82H</td>
<td>Cache</td>
<td>2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>83H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>84H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>85H</td>
<td>Cache</td>
<td>2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size</td>
</tr>
<tr>
<td>86H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>87H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size</td>
</tr>
</tbody>
</table>
Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0H</td>
<td>DTLB</td>
<td>DTLB: 4k pages, fully associative, 32 entries</td>
</tr>
<tr>
<td>B0H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries</td>
</tr>
<tr>
<td>B1H</td>
<td>TLB</td>
<td>Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries</td>
</tr>
<tr>
<td>B2H</td>
<td>TLB</td>
<td>Instruction TLB: 4KByte pages, 4-way set associative, 64 entries</td>
</tr>
<tr>
<td>B3H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte pages, 4-way set associative, 128 entries</td>
</tr>
<tr>
<td>B4H</td>
<td>TLB</td>
<td>Data TLB1: 4 KByte pages, 4-way associative, 256 entries</td>
</tr>
<tr>
<td>B5H</td>
<td>TLB</td>
<td>Instruction TLB: 4KByte pages, 8-way set associative, 64 entries</td>
</tr>
<tr>
<td>B6H</td>
<td>TLB</td>
<td>Instruction TLB: 4KByte pages, 8-way set associative, 128 entries</td>
</tr>
<tr>
<td>BAH</td>
<td>TLB</td>
<td>Data TLB1: 4 KByte pages, 4-way associative, 64 entries</td>
</tr>
<tr>
<td>C0H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries</td>
</tr>
<tr>
<td>C1H</td>
<td>STLB</td>
<td>Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries</td>
</tr>
<tr>
<td>C2H</td>
<td>DTLB</td>
<td>DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries</td>
</tr>
<tr>
<td>C3H</td>
<td>STLB</td>
<td>Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GBbyte pages, 4-way, 16 entries.</td>
</tr>
<tr>
<td>C4H</td>
<td>DTLB</td>
<td>DTLB: 2M/4M Byte pages, 4-way associative, 32 entries</td>
</tr>
<tr>
<td>CAH</td>
<td>STLB</td>
<td>Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries</td>
</tr>
<tr>
<td>D0H</td>
<td>Cache</td>
<td>3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>D1H</td>
<td>Cache</td>
<td>3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>D2H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>D6H</td>
<td>Cache</td>
<td>3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>D7H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>D8H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>DCH</td>
<td>Cache</td>
<td>3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>DDH</td>
<td>Cache</td>
<td>3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>DEH</td>
<td>Cache</td>
<td>3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>E2H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>E3H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>E4H</td>
<td>Cache</td>
<td>3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>EAH</td>
<td>Cache</td>
<td>3rd-level cache: 12MByte, 24-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>EBH</td>
<td>Cache</td>
<td>3rd-level cache: 18MByte, 24-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>ECH</td>
<td>Cache</td>
<td>3rd-level cache: 24MByte, 24-way set associative, 64 byte line size</td>
</tr>
<tr>
<td>F0H</td>
<td>Prefetch</td>
<td>64-Byte prefetching</td>
</tr>
<tr>
<td>F1H</td>
<td>Prefetch</td>
<td>128-Byte prefetching</td>
</tr>
<tr>
<td>FEH</td>
<td>General</td>
<td>CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.</td>
</tr>
<tr>
<td>FFH</td>
<td>General</td>
<td>CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters</td>
</tr>
</tbody>
</table>
Example 3-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

EAX 66 5B 50 01H
EBX 0H
ECX 0H
EDX 00 7A 70 00H

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This value should be ignored.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K-μop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

\[
= (\text{Ways} + 1) \times (\text{Partitions} + 1) \times (\text{Line}_\text{Size} + 1) \times (\text{Sets} + 1)
\]

\[
= (\text{EBX}[31:22] + 1) \times (\text{EBX}[21:12] + 1) \times (\text{EBX}[11:0] + 1) \times (\text{ECX} + 1)
\]

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, “Multiple-Processor Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.
INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, "Introduction to Virtual-Machine Extensions," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

For i = 2 to 62 // sub-leaf 1 is reserved
  IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1) // VECTOR is the 64-bit value of EDX:EAX
    Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
  FI;

INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 0FH and ECX = n (n >= 1, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit
1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

**INPUT EAX = 12H: Returns Intel SGX Enumeration Information**

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

**INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information**

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

**INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information**

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

**INPUT EAX = 16H: Returns Processor Frequency Information**

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

**INPUT EAX = 17H: Returns System-On-Chip Information**

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

**INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information**

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-8.

**METHODS FOR RETURNING BRANDING INFORMATION**

Use the following techniques to access branding information:

1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: “Identification of Earlier IA-32 Processors” in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

**The Processor Brand String Method**

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.
How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 8000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.
Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

**Table 3-13. Processor Brand String Returned with Pentium 4 Processor**

<table>
<thead>
<tr>
<th>EAX Input Value</th>
<th>Return Values</th>
<th>ASCII Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000002H</td>
<td>EAX = 20202020H</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td></td>
<td>EBX = 20202020H</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td></td>
<td>ECX = 20202020H</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td></td>
<td>EDX = 6E492020H</td>
<td>&quot;ni &quot;</td>
</tr>
<tr>
<td>80000003H</td>
<td>EAX = 28666574H</td>
<td>&quot;(let&quot;</td>
</tr>
<tr>
<td></td>
<td>EBX = 50202952H</td>
<td>&quot;P )R&quot;</td>
</tr>
<tr>
<td></td>
<td>ECX = 69746E65H</td>
<td>&quot;itne&quot;</td>
</tr>
<tr>
<td></td>
<td>EDX = 52286D75H</td>
<td>&quot;R(mu&quot;</td>
</tr>
<tr>
<td>80000004H</td>
<td>EAX = 20342029H</td>
<td>&quot; 4 )&quot;</td>
</tr>
<tr>
<td></td>
<td>EBX = 20555043H</td>
<td>&quot; UPC&quot;</td>
</tr>
<tr>
<td></td>
<td>ECX = 30303531H</td>
<td>&quot;0051&quot;</td>
</tr>
<tr>
<td></td>
<td>EDX = 007A484DH</td>
<td>&quot;0zHM&quot;</td>
</tr>
</tbody>
</table>

**Extracting the Processor Frequency from Brand Strings**

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.

![Algorithm for Extracting Processor Frequency](OM15195)

**Figure 3-10. Algorithm for Extracting Processor Frequency**
The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associated with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-14 shows brand indices that have identification strings associated with them.

### Table 3-14. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

<table>
<thead>
<tr>
<th>Brand Index</th>
<th>Brand String</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>This processor does not support the brand identification feature</td>
</tr>
<tr>
<td>01H</td>
<td>Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>02H</td>
<td>Intel(R) Pentium(R) III processor¹</td>
</tr>
<tr>
<td>03H</td>
<td>Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor</td>
</tr>
<tr>
<td>04H</td>
<td>Intel(R) Pentium(R) III processor</td>
</tr>
<tr>
<td>06H</td>
<td>Mobile Intel(R) Pentium(R) III processor-M</td>
</tr>
<tr>
<td>07H</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>08H</td>
<td>Intel(R) Pentium(R) 4 processor</td>
</tr>
<tr>
<td>09H</td>
<td>Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>0AH</td>
<td>Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>0BH</td>
<td>Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP</td>
</tr>
<tr>
<td>0CH</td>
<td>Intel(R) Xeon(R) processor</td>
</tr>
<tr>
<td>0EH</td>
<td>Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor</td>
</tr>
<tr>
<td>0FH</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>11H</td>
<td>Mobile Genuine Intel(R) processor</td>
</tr>
<tr>
<td>12H</td>
<td>Intel(R) Celeron(R) M processor</td>
</tr>
<tr>
<td>13H</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>14H</td>
<td>Intel(R) Celeron(R) processor</td>
</tr>
<tr>
<td>15H</td>
<td>Mobile Genuine Intel(R) processor</td>
</tr>
<tr>
<td>16H</td>
<td>Intel(R) Pentium(R) M processor</td>
</tr>
<tr>
<td>17H</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>18H – OFFH</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Indicates versions of these processors that were introduced after the Pentium III

IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.
Operation

IA32_BIOS_SIGN_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF
  EAX = 0:
    EAX ← Highest basic function input value understood by CPUID;
    EBX ← Vendor identification string;
    EDX ← Vendor identification string;
    ECX ← Vendor identification string;
  BREAK;
  EAX = 1H:
    EAX[3:0] ← Stepping ID;
    EAX[7:4] ← Model;
    EAX[11:8] ← Family;
    EAX[13:12] ← Processor type;
    EAX[15:14] ← Reserved;
    EAX[19:16] ← Extended Model;
    EAX[27:20] ← Extended Family;
    EAX[31:28] ← Reserved;
    EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
    EBX[15:8] ← CLFLUSH Line Size;
    EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
    EBX[24:31] ← Initial APIC ID;
    ECX ← Feature flags; (* See Figure 3-7. *)
    EDX ← Feature flags; (* See Figure 3-8. *)
  BREAK;
  EAX = 2H:
    EAX ← Cache and TLB information;
    EBX ← Cache and TLB information;
    ECX ← Cache and TLB information;
    EDX ← Cache and TLB information;
  BREAK;
  EAX = 3H:
    EAX ← Reserved;
    EBX ← Reserved;
    ECX ← ProcessorSerialNumber[31:0];
      (* Pentium III processors only, otherwise reserved. *)
    EDX ← ProcessorSerialNumber[63:32];
      (* Pentium III processors only, otherwise reserved. *)
  BREAK
  EAX = 4H:
    EAX ← Deterministic Cache Parameters Leaf; (* See Table 3-8. *)
    EBX ← Deterministic Cache Parameters Leaf;
    ECX ← Deterministic Cache Parameters Leaf;
    EDX ← Deterministic Cache Parameters Leaf;
  BREAK;
  EAX = 5H:
    EAX ← MONITOR/MWAIT Leaf; (* See Table 3-8. *)
    EBX ← MONITOR/MWAIT Leaf;
    ECX ← MONITOR/MWAIT Leaf;
    EDX ← MONITOR/MWAIT Leaf;
  BREAK;
EAX = 6H:
EAX ← Thermal and Power Management Leaf; (* See Table 3-8. *)
EBX ← Thermal and Power Management Leaf;
ECX ← Thermal and Power Management Leaf;
EDX ← Thermal and Power Management Leaf;
BREAK;
EAX = 7H:
EAX ← Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-8. *)
EBX ← Structured Extended Feature Flags Enumeration Leaf;
ECX ← Structured Extended Feature Flags Enumeration Leaf;
EDX ← Structured Extended Feature Flags Enumeration Leaf;
BREAK;
EAX = 8H:
EAX ← Reserved = 0;
EBX ← Reserved = 0;
ECX ← Reserved = 0;
EDX ← Reserved = 0;
BREAK;
EAX = 9H:
EAX ← Direct Cache Access Information Leaf; (* See Table 3-8. *)
EBX ← Direct Cache Access Information Leaf;
ECX ← Direct Cache Access Information Leaf;
EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
EAX ← Architectural Performance Monitoring Leaf; (* See Table 3-8. *)
EBX ← Architectural Performance Monitoring Leaf;
ECX ← Architectural Performance Monitoring Leaf;
EDX ← Architectural Performance Monitoring Leaf;
BREAK
EAX = BH:
EAX ← Extended Topology Enumeration Leaf; (* See Table 3-8. *)
EBX ← Extended Topology Enumeration Leaf;
ECX ← Extended Topology Enumeration Leaf;
EDX ← Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
EAX ← Reserved = 0;
EBX ← Reserved = 0;
ECX ← Reserved = 0;
EDX ← Reserved = 0;
BREAK;
EAX = DH:
EAX ← Processor Extended State Enumeration Leaf; (* See Table 3-8. *)
EBX ← Processor Extended State Enumeration Leaf;
ECX ← Processor Extended State Enumeration Leaf;
EDX ← Processor Extended State Enumeration Leaf;
BREAK;
EAX = EH:
EAX ← Reserved = 0;
EBX ← Reserved = 0;
ECX ← Reserved = 0;
EDX ← Reserved = 0;
BREAK;
EAX = FH:
   EAX ← Intel Resource Director Technology Monitoring Enumeration Leaf; (* See Table 3-8. *)
   EBX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
   ECX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
   EDX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
BREAK;
EAX = 10H:
   EAX ← Intel Resource Director Technology Allocation Enumeration Leaf; (* See Table 3-8. *)
   EBX ← Intel Resource Director Technology Allocation Enumeration Leaf;
   ECX ← Intel Resource Director Technology Allocation Enumeration Leaf;
   EDX ← Intel Resource Director Technology Allocation Enumeration Leaf;
BREAK;
EAX = 12H:
   EAX ← Intel SGX Enumeration Leaf; (* See Table 3-8. *)
   EBX ← Intel SGX Enumeration Leaf;
   ECX ← Intel SGX Enumeration Leaf;
   EDX ← Intel SGX Enumeration Leaf;
BREAK;
EAX = 14H:
   EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 3-8. *)
   EBX ← Intel Processor Trace Enumeration Leaf;
   ECX ← Intel Processor Trace Enumeration Leaf;
   EDX ← Intel Processor Trace Enumeration Leaf;
BREAK;
EAX = 15H:
   EAX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (* See Table 3-8. *)
   EBX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
   ECX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
   EDX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
BREAK;
EAX = 16H:
   EAX ← Processor Frequency Information Enumeration Leaf; (* See Table 3-8. *)
   EBX ← Processor Frequency Information Enumeration Leaf;
   ECX ← Processor Frequency Information Enumeration Leaf;
   EDX ← Processor Frequency Information Enumeration Leaf;
BREAK;
EAX = 17H:
   EAX ← System-On-Chip Vendor Attribute Enumeration Leaf; (* See Table 3-8. *)
   EBX ← System-On-Chip Vendor Attribute Enumeration Leaf;
   ECX ← System-On-Chip Vendor Attribute Enumeration Leaf;
   EDX ← System-On-Chip Vendor Attribute Enumeration Leaf;
BREAK;
EAX = 18H:
   EAX ← Deterministic Address Translation Parameters Enumeration Leaf; (* See Table 3-8. *)
   EBX ← Deterministic Address Translation Parameters Enumeration Leaf;
   ECX ← Deterministic Address Translation Parameters Enumeration Leaf;
   EDX ← Deterministic Address Translation Parameters Enumeration Leaf;
BREAK;
EAX = 80000000H:
   EAX ← Highest extended function input value understood by CPUID;
   EBX ← Reserved;
   ECX ← Reserved;
   EDX ← Reserved;
BREAK;
EAX = 80000001H:
  EAX ← Reserved;
  EBX ← Reserved;
  ECX ← Extended Feature Bits (* See Table 3-8.*);
  EDX ← Extended Feature Bits (* See Table 3-8.*);
BREAK;
EAX = 80000002H:
  EAX ← Processor Brand String;
  EBX ← Processor Brand String, continued;
  ECX ← Processor Brand String, continued;
  EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000003H:
  EAX ← Processor Brand String, continued;
  EBX ← Processor Brand String, continued;
  ECX ← Processor Brand String, continued;
  EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000004H:
  EAX ← Processor Brand String, continued;
  EBX ← Processor Brand String, continued;
  ECX ← Processor Brand String, continued;
  EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000005H:
  EAX ← Reserved = 0;
  EBX ← Reserved = 0;
  ECX ← Reserved = 0;
  EDX ← Reserved = 0;
BREAK;
EAX = 80000006H:
  EAX ← Reserved = 0;
  EBX ← Reserved = 0;
  ECX ← Cache information;
  EDX ← Reserved = 0;
BREAK;
EAX = 80000007H:
  EAX ← Reserved = 0;
  EBX ← Reserved = 0;
  ECX ← Reserved = 0;
  EDX ← Reserved = Misc Feature Flags;
BREAK;
EAX = 80000008H:
  EAX ← Reserved = Physical Address Size Information;
  EBX ← Reserved = Virtual Address Size Information;
  ECX ← Reserved = 0;
  EDX ← Reserved = 0;
BREAK;
EAX >= 40000000H and EAX <= 4FFFFFFFH:
DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
  (* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
  EAX ← Reserved; (* Information returned for highest basic information leaf. *)
  EBX ← Reserved; (* Information returned for highest basic information leaf. *)
  ECX ← Reserved; (* Information returned for highest basic information leaf. *)
EDX ← Reserved; (* Information returned for highest basic information leaf. *)

BREAK;

ESAC;

**Flags Affected**

None.

**Exceptions (All Operating Modes)**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#UD</td>
<td>If the LOCK prefix is used.</td>
</tr>
</tbody>
</table>

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.
**CRC32 — Accumulate CRC32 Value**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 38 F0  /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Accumulate CRC32 on r/m8.</td>
</tr>
<tr>
<td>CRC32 r32, r/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX 0F 38 F0  /r</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Accumulate CRC32 on r/m8.</td>
</tr>
<tr>
<td>CRC32 r32, r/m8*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 0F 38 F1  /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Accumulate CRC32 on r/m16.</td>
</tr>
<tr>
<td>CRC32 r32, r/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 0F 38 F1  /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Accumulate CRC32 on r/m32.</td>
</tr>
<tr>
<td>CRC32 r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX.W 0F 38 F0  /r</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Accumulate CRC32 on r/m8.</td>
</tr>
<tr>
<td>CRC32 r64, r/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX.W 0F 38 F1  /r</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Accumulate CRC32 on r/m64.</td>
</tr>
<tr>
<td>CRC32 r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

**Operation**

**Notes:**

BIT_REFLECT64: DST[63-0] = SRC[0-63]
BIT_REFLECT32: DST[31-0] = SRC[0-31]
BIT_REFLECT16: DST[15-0] = SRC[0-15]
BIT_REFLECT8: DST[7-0] = SRC[0-7]
MOD2: Remainder from Polynomial division modulus 2
CRC32 instruction for 64-bit source operand and 64-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[63-0] & \leftarrow \text{BIT\_REFLECT64} (\text{SRC}[63-0]) \\
\text{TEMP2}[31-0] & \leftarrow \text{BIT\_REFLECT32} (\text{DEST}[31-0]) \\
\text{TEMP3}[95-0] & \leftarrow \text{TEMP1}[63-0] \ll 32 \\
\text{TEMP4}[95-0] & \leftarrow \text{TEMP2}[31-0] \ll 64 \\
\text{TEMP5}[95-0] & \leftarrow \text{TEMP3}[95-0] \oplus \text{TEMP4}[95-0] \\
\text{TEMP6}[31-0] & \leftarrow \text{TEMP5}[95-0] \mod 11ECD6F41H \\
\text{DEST}[31-0] & \leftarrow \text{BIT\_REFLECT} (\text{TEMP6}[31-0]) \\
\text{DEST}[63-32] & \leftarrow \text{00000000H}
\end{align*}
\]

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[31-0] & \leftarrow \text{BIT\_REFLECT32} (\text{SRC}[31-0]) \\
\text{TEMP2}[31-0] & \leftarrow \text{BIT\_REFLECT32} (\text{DEST}[31-0]) \\
\text{TEMP3}[63-0] & \leftarrow \text{TEMP1}[31-0] \ll 32 \\
\text{TEMP4}[63-0] & \leftarrow \text{TEMP2}[31-0] \ll 32 \\
\text{TEMP5}[63-0] & \leftarrow \text{TEMP3}[63-0] \oplus \text{TEMP4}[63-0] \\
\text{TEMP6}[31-0] & \leftarrow \text{TEMP5}[63-0] \mod 11ECD6F41H \\
\text{DEST}[31-0] & \leftarrow \text{BIT\_REFLECT} (\text{TEMP6}[31-0])
\end{align*}
\]

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[15-0] & \leftarrow \text{BIT\_REFLECT16} (\text{SRC}[15-0]) \\
\text{TEMP2}[31-0] & \leftarrow \text{BIT\_REFLECT32} (\text{DEST}[31-0]) \\
\text{TEMP3}[47-0] & \leftarrow \text{TEMP1}[15-0] \ll 16 \\
\text{TEMP4}[47-0] & \leftarrow \text{TEMP2}[31-0] \ll 16 \\
\text{TEMP5}[47-0] & \leftarrow \text{TEMP3}[47-0] \oplus \text{TEMP4}[47-0] \\
\text{TEMP6}[31-0] & \leftarrow \text{TEMP5}[47-0] \mod 11ECD6F41H \\
\text{DEST}[31-0] & \leftarrow \text{BIT\_REFLECT} (\text{TEMP6}[31-0])
\end{align*}
\]

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[7-0] & \leftarrow \text{BIT\_REFLECT8} (\text{SRC}[7-0]) \\
\text{TEMP2}[31-0] & \leftarrow \text{BIT\_REFLECT32} (\text{DEST}[31-0]) \\
\text{TEMP3}[39-0] & \leftarrow \text{TEMP1}[7-0] \ll 32 \\
\text{TEMP4}[39-0] & \leftarrow \text{TEMP2}[31-0] \ll 8 \\
\text{TEMP5}[39-0] & \leftarrow \text{TEMP3}[39-0] \oplus \text{TEMP4}[39-0] \\
\text{TEMP6}[31-0] & \leftarrow \text{TEMP5}[39-0] \mod 11ECD6F41H \\
\text{DEST}[31-0] & \leftarrow \text{BIT\_REFLECT} (\text{TEMP6}[31-0]) \\
\text{DEST}[63-32] & \leftarrow \text{00000000H}
\end{align*}
\]

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[7-0] & \leftarrow \text{BIT\_REFLECT8} (\text{SRC}[7-0]) \\
\text{TEMP2}[31-0] & \leftarrow \text{BIT\_REFLECT32} (\text{DEST}[31-0]) \\
\text{TEMP3}[39-0] & \leftarrow \text{TEMP1}[7-0] \ll 32 \\
\text{TEMP4}[39-0] & \leftarrow \text{TEMP2}[31-0] \ll 8 \\
\text{TEMP5}[39-0] & \leftarrow \text{TEMP3}[39-0] \oplus \text{TEMP4}[39-0] \\
\text{TEMP6}[31-0] & \leftarrow \text{TEMP5}[39-0] \mod 11ECD6F41H \\
\text{DEST}[31-0] & \leftarrow \text{BIT\_REFLECT} (\text{TEMP6}[31-0])
\end{align*}
\]

Flags Affected

None
Intel C/C++ Compiler Intrinsic Equivalent

unsigned int _mm_crc32_u8( unsigned int crc, unsigned char data )
unsigned int _mm_crc32_u16( unsigned int crc, unsigned short data )
unsigned int _mm_crc32_u32( unsigned int crc, unsigned int data )
unsigned __int64 _mm_crc32_u64( unsigned __int64 crc, unsigned __int64 data )

SIMD Floating Point Exceptions

None

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
If LOCK prefix is used.
## CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:Wig E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:Wig E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F:W0 E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert 2 packed signed doubleword integers from xmm2/m128/m32bcst to eight packed double-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F:W0 E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F:W0 E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double-precision floating-point values in the destination operand (the first operand).

**EVEX encoded versions:** The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

**VEX.256 encoded version:** The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register.

**VEX.128 encoded version:** The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
Operation

VCVTDQ2PD (EVEX encoded versions) when src operand is a register

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[\text{FOR } j \leftarrow 0 \text{ TO } KL-1\]
\[\begin{align*}
i & \leftarrow j \ast 64 \\
k & \leftarrow j \ast 32 \\
\text{IF } k1[j] \text{ OR } *\text{no writemask}* & \text{ THEN } \text{DEST}[i+63:i] \leftarrow \\
& \quad \text{Convert_Integer_To_Double_Precision_Floating_Point}(SRC[k+31:k]) \\
\text{ELSE} & \text{ IF } *\text{merging-masking}* \\
& \quad \text{THEN } *\text{DEST}[i+63:i] \text{ remains unchanged}* \\
& \quad \text{ELSE } *\text{zeroing-masking}* \\
& \quad \text{DEST}[i+63:i] \leftarrow 0 \\
& \end{align*}\]

\[\text{FI}\]

\[\text{FI};\]

\[\text{ENDFOR}\]

\[\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\]
VCVTDQ2PD (EVEX encoded versions) when src operand is a memory source

(\(KL, VL\) = (2, 128), (4, 256), (8, 512))

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j \times 64\)
\(k \leftarrow j \times 32\)

IF \(k1[j]\) OR "no writemask"

THEN

IF (EVEX.b = 1)

THEN

\(\text{DEST}[i+63:i] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[31:0])\)
ELSE

\(\text{DEST}[i+63:i] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[k+31:k])\)
FI;
ELSE

IF "merging-masking" ; merging-masking

THEN "DEST[i+63:i] remains unchanged"

ELSE ; zeroing-masking

\(\text{DEST}[i+63:i] \leftarrow 0\)
FI
ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

VCVTDQ2PD (VEX.256 encoded version)

\(\text{DEST}[63:0] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[31:0])\)
\(\text{DEST}[127:64] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[63:32])\)
\(\text{DEST}[191:128] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[95:64])\)
\(\text{DEST}[255:192] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[127:96])\)
\(\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0\)

VCVTDQ2PD (VEX.128 encoded version)

\(\text{DEST}[63:0] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[31:0])\)
\(\text{DEST}[127:64] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[63:32])\)
\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0\)

CVTDQ2PD (128-bit Legacy SSE version)

\(\text{DEST}[63:0] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[31:0])\)
\(\text{DEST}[127:64] \leftarrow \text{Convert Integer To Double-Precision Floating Point}(\text{SRC}[63:32])\)
\(\text{DEST}[\text{MAXVL}-1:128] \) (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

\(\text{VCVTDQ2PD } _\text{m512d } _\text{mm512_cvtepi32_pd}(\text{ } _\text{m256i } a)\);
\(\text{VCVTDQ2PD } _\text{m512d } _\text{mm512_mask_cvtepi32_pd}(\text{ } _\text{m512d } s, _\text{mmask8 } k, _\text{m256i } a)\);
\(\text{VCVTDQ2PD } _\text{m512d } _\text{mm512_maskz_cvtepi32_pd}(\text{ } _\text{mmask8 } k, _\text{m256i } a)\);
\(\text{VCVTDQ2PD } _\text{m256d } _\text{mm256_cvtepi32_pd}(\text{ } _\text{m128i } src)\);
\(\text{VCVTDQ2PD } _\text{m256d } _\text{mm256_mask_cvtepi32_pd}(\text{ } _\text{m256d } s, _\text{mmask8 } k, _\text{m256i } a)\);
\(\text{VCVTDQ2PD } _\text{m256d } _\text{mm256_maskz_cvtepi32_pd}(\text{ } _\text{mmask8 } k, _\text{m256i } a)\);
\(\text{VCVTDQ2PD } _\text{m128d } _\text{mm128_cvtepi32_pd}(\text{ } _\text{m128d } s, _\text{mmask8 } k, _\text{m128i } a)\);
\(\text{VCVTDQ2PD } _\text{m128d } _\text{mm128_mask_cvtepi32_pd}(\text{ } _\text{mmask8 } k, _\text{m128i } a)\);
\(\text{CVTDQ2PD } _\text{m128d } _\text{mm128_cvtepi32_pd}(\text{ } _\text{m128i } src)\)
Other Exceptions
VEX-encoded instructions, see Exceptions Type 5;
EVEX-encoded instructions, see Exceptions Type E5.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
### CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F:wIG 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F:wIG 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.0F:w0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F:w0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F:w0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single-precision floating-point values in the destination operand.

**EVEX encoded versions:** The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding register destination are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

**128-bit Legacy SSE version:** The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.**
**Operation**

**VCVTDQ2PS (EVEX encoded versions) when SRC operand is a register**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

**IF (VL = 512) AND (EVEX.b = 1)\)**

**THEN**

\[SET_{RM}(EVEX.RC); \text{ ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1}\]

**ELSE**

\[SET_{RM}(MXCSR.RM); \text{ ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1}\]

**FI;**

**FOR** \[j \leftarrow 0 \text{ TO KL-1}\]

\[i \leftarrow j \times 32\]

**IF k1[j] OR *no writemask*\**

**THEN** \[\text{DEST}[i+31:i] \leftarrow \text{Convert_Integer_To_Single_Precision_Floating_Point}(\text{SRC}[i+31:i])\]

**ELSE**

**IF *merging-masking* ; merging-masking**

**THEN** \[\text{DEST}[i+31:i] \text{ remains unchanged}\]

**ELSE ; zeroing-masking**

\[\text{DEST}[i+31:i] \leftarrow 0\]

**FI**

**FI;**

**ENDFOR**

\[\text{DEST}[\text{MAXVL}-1:\text{VL}] \leftarrow 0\]

**VCVTDQ2PS (EVEX encoded versions) when SRC operand is a memory source**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

**FOR** \[j \leftarrow 0 \text{ TO KL-1}\]

\[i \leftarrow j \times 32\]

**IF k1[j] OR *no writemask*\**

**THEN**

**IF (EVEX.b = 1)\**

**THEN**

\[\text{DEST}[i+31:i] \leftarrow \text{Convert_Integer_To_Single_Precision_Floating_Point}(\text{SRC}[31:0])\]

**ELSE**

\[\text{DEST}[i+31:i] \leftarrow \text{Convert_Integer_To_Single_Precision_Floating_Point}(\text{SRC}[i+31:i])\]

**FI;**

**ELSE**

**IF *merging-masking* ; merging-masking**

**THEN** \[\text{DEST}[i+31:i] \text{ remains unchanged}\]

**ELSE ; zeroing-masking**

\[\text{DEST}[i+31:i] \leftarrow 0\]

**FI**

**FI;**

**ENDFOR**

\[\text{DEST}[\text{MAXVL}-1:\text{VL}] \leftarrow 0\]
VCVTQ2PS (VEX.256 encoded version)
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[159:128] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224])
DEST[MAXVL-1:256] ← 0

VCVTQ2PS (VEX.128 encoded version)
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:128] ← 0

VCVTQ2PS (128-bit Legacy SSE version)
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTQ2PS __m512 __m512_cvtepi32_ps(__m512i a);
VCVTQ2PS __m512 __m512_mask_cvtepi32_ps(__m512 s, __mmask16 k, __m512i a);
VCVTQ2PS __m512 __m512_maskz_cvtepi32_ps(__mmask16 k, __m512i a);
VCVTQ2PS __m512 __m512_cvt_roundepi32_ps(__m512i a, int r);
VCVTQ2PS __m512 __m512_maskz_cvtepi32_ps(__mmask16 k, __m512i a, int r);
VCVTQ2PS __m256 __m256_mask_cvtepi32_ps(__m256 s, __mmask8 k, __m256i a);
VCVTQ2PS __m256 __m256_maskz_cvtepi32_ps(__mmask8 k, __m256i a);
VCVTQ2PS __m128 __m128_mask_cvtepi32_ps(__m128 s, __mmask8 k, __m128i a);
VCVTQ2PS __m128 __m128_maskz_cvtepi32_ps(__mmask8 k, __m128i a);
CVTDQ2PS __m256 __m256_cvtepi32_ps(__m256i src)
CVTDQ2PS __m128 __m128_cvtepi32_ps(__m128i src)

SIMD Floating-Point Exceptions

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;
EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>VEX.128.F2.0F:WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F2.0F:WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:W1E6 /r:EVEX.256.F2.0F:W1E6 /r:EVEX.512.F2.0F:W1E6 /r:CVVTPD2DQ xmm1 [k1]{z}, xmm2/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W1E6 /r:EVEX.512.F2.0F:W1E6 /r:CVVTPD2DQ xmm1 [k1]{z}, ymm2/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W1E6 /r:CVVTPD2DQ ymm1 [k1]{z}, zmm2/m512/m64bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed double-precision floating-point values in the source operand (second operand) to packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where \(w\) represents the number of bits in the destination format) is returned.

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

**VEX.256 encoded version:** The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits [127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
**Operation**

VCVTPD2DQ (EVEX encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_RM(EVEX.RC);
ELSE

SET_RM(MXCSR.RM);
FI;

FOR j \(\leftarrow 0\) TO KL-1

\[i \leftarrow j \times 32\]

\[k \leftarrow j \times 64\]

IF k1[j] OR *no writemask*

THEN DEST\[i+31:i\] \(\leftarrow\) Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking

DEST\[i+31:i\] \(\leftarrow 0\)
FI

FI;
ENDFOR

DEST[MAXVL-1:VL/2] \(\leftarrow 0\)
VCVTPD2DQ (EVEX encoded versions) when src operand is a memory source

KL, VL = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
        ELSE
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI
  ENDFOR

DEST[MAXVL-1:VL/2] ← 0

VCVTPD2DQ (VEX.256 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192])
DEST[MAXVL-1:128] ← 0

VCVTPD2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[MAXVL-1:64] ← 0

CVTPD2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] ← 0
DEST[MAXVL-1:128] (unmodified)
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPD2DQ __m256i __mm512_cvtpd_epi32(__m512d a);
VCVTPD2DQ __m256i __mm512_mask_cvtpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_maskz_cvtpd_epi32(__mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_cvt_roundpd_epi32(__m512d a, int r);
VCVTPD2DQ __m256i __mm512_mask_cvt_roundpd_epi32(__m512d a, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i __mm512_maskz_cvt_roundpd_epi32(__mmask8 k, __m512d a);
VCVTPD2DQ __m128i __mm256_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i __mm256_maskz_cvtpd_epi32(__m256d a);
VCVTPD2DQ __m128i __mm_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm_maskz_cvtpd_epi32(__mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm256_cvtpd_epi32(__m256d src);
CVTPD2DQ __m128i __mm_cvtpd_epi32(__m128d src)

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
See Exceptions Type 2; additionally
EVEX-encoded instructions, see Exceptions Type E2.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

\[
\text{DEST}[31:0] \leftarrow \text{Convert Double Precision Floating Point To Integer32}(\text{SRC}[63:0]); \\
\text{DEST}[63:32] \leftarrow \text{Convert Double Precision Floating Point To Integer32}(\text{SRC}[127:64]);
\]

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD1PI: __m64 _mm_cvtpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1 S</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F:W1 S</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 V</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 V</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 V</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight single-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed single-precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

VCVTPD2PS (EVEX encoded version) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0

Figure 3-13. VCVTPD2PS (VEX.256 encoded version)
VCVTDP2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI
  ENDFOR
DEST[MAXVL-1:VL/2] ← 0

VCVTDP2PS (VEX.256 encoded version)
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
DEST[MAXVL-1:128] ← 0

VCVTDP2PS (VEX.128 encoded version)
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] ← 0

CVTDP2PS (128-bit Legacy SSE version)
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] ← 0
DEST[MAXVL-1:128] (unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2PS __m256 __mm512_cvtpd_ps(__m512d a);
VCVTPD2PS __m256 __mm512_mask_cvtpd_ps(__m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_maskz_cvtpd_ps(__mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_cvt_roundpd_ps(__m512d a, int r);
VCVTPD2PS __m256 __mm512_mask_cvt_roundpd_ps(__m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 __mm512_maskz_cvt_roundpd_ps(__mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 __mm256_mask_cvtpd_ps(__m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm256_maskz_cvtpd_ps(__mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm_mask_cvtpd_ps(__m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 __mm_maskz_cvtpd_ps(__mmask8 k, __m128d a);
CVTPD2PS __m128 __mm_cvtpd_ps(__m256d a)

SIMD Floating-Point Exceptions
Invalid, Precision, Underflow, Overflow, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2;
EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvv != 1111B or EVEX.vvv != 1111B.
CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Oper 4-Bit Mode</th>
<th>Compat/32-Bit Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2A /r</td>
<td>RM</td>
<td>Valid</td>
<td>Convert two packed signed doubleword integers from mm/mem64 to two packed double-precision floating-point values in xmm.</td>
</tr>
</tbody>
</table>

**NOTES:**

*Operation is different for different operand sets; see the Description section.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg</td>
<td>ModRM:r/m</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands xmm, mm:** the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.

- **For operands xmm, m64:** the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32]);

**Intel C/C++ Compiler Intrinsic Equivalent**

CVTPI2PD: __m128d_mm_cvtpi32_pd(__m64 a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 22-6, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
### CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/LEG Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2A /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two signed doubleword integers from mmm/m64 to two single-precision floating-point values in xmm.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>RM</td>
</tr>
</tbody>
</table>

**Description**

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

\[
\text{DEST}[31:0] \leftarrow \text{Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])};
\]

\[
\text{DEST}[63:32] \leftarrow \text{Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])};
\]

(* High quadword of destination unchanged *)

**Intel C/C++ Compiler Intrinsic Equivalent**

CVTPI2PS: __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

See Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.*
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F:WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where \(w\) represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

VCVTPS2DQ (encoded versions) when src operand is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

VCVTPS2DQ (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
                ELSE
                    DEST[i+31:i] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
                FI
        FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
VCVTPS2DQ (VEX.256 encoded version)
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])

VCVTPS2DQ (VEX.128 encoded version)
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])

CVTPS2DQ (128-bit Legacy SSE version)
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2DQ __m512i _mm512_cvtps_epi32( __m512 a);
VCVTPS2DQ __m512i __mm512_mask_cvtps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2DQ __m512i __mm512_maskz_cvtps_epi32( __mmask16 k, __m512 a);
VCVTPS2DQ __m512i __mm512_cvtpd_roundps_epi32( __m512 a, int r);
VCVTPS2DQ __m512i __mm512_mask_cvtpd_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m512i __mm512_maskz_cvtpd_roundps_epi32( __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m256i __mm256_mask_cvtps_epi32( __mmask8 k, __m256 a);
VCVTPS2DQ __m256i __mm256_maskz_cvtps_epi32( __mmask8 k, __m256 a);
VCVTPS2DQ __m128i __mm_mask_cvtps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2DQ __m128i __mm_maskz_cvtps_epi32( __mmask8 k, __m128 a);
VCVTPS2DQ __m256i __mm256_cvtps_epi32( __m256 a)
CVTPS2DQ __m128i __mm_cvtps_epi32( __m128 a)

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2;
EVEX-encoded instructions, see Exceptions Type E2.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VCVTPS2PD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VCVTPS2PD ymm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed single-precision floating-point values in xmm2/m128 to four packed double-precision floating-point values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.0F.WIG 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert two packed single-precision floating-point values in xmm2/m64/m32bcst to packed double-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.WIG 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert four packed single-precision floating-point values in xmm2/m128/m32bcst to packed double-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.WIG 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert eight packed single-precision floating-point values in ymm2/m256/b32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts two, four or eight packed single-precision floating-point values in the source operand (second operand) to two, four or eight packed double-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zereod.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

VCVTPS2PD (EVEX encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
           Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VCVTPS2PD (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
        FI;
      ELSE
        FI
    ELSE
IF *merging-masking* ; merging-masking
  THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
  DEST[i+63:i] ← 0
FI

ENDFOR
DEST[MAXVL-1:VL] ← 0

VCVTPS2PD (VEX.256 encoded version)
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:256] ← 0

VCVTPS2PD (VEX.128 encoded version)
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] ← 0

CVTPS2PD (128-bit Legacy SSE version)
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2PD __m512d _mm512_cvtps_pd( __m256 a);
VCVTPS2PD __m512d _mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m512d a);
VCVTPS2PD __m512d _mm512_maskz_cvtps_pd( __mmask8 k, __m512d a);
VCVTPS2PD __m512d _mm512_cvt_roundps_pd( __m512d s, __mmask8 k, __m512d a);
VCVTPS2PD __m512d _mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m512d a, int sae);
VCVTPS2PD __m512d _mm512_maskz_cvt_roundps_pd( __mmask8 k, __m512d a, int sae);
VCVTPS2PD __m256d _mm256_cvtps_pd( __m128 a);
VCVTPS2PD __m256d _mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m256d a);
VCVTPS2PD __m256d _mm256_maskz_cvtps_pd( __mmask8 k, __m256d a);
VCVTPS2PD __m256d _mm256_cvt_roundps_pd( __m256d s, __mmask8 k, __m256d a);
VCVTPS2PD __m256d _mm256_mask_cvt_roundps_pd( __m256d s, __mmask8 k, __m256d a, int sae);
VCVTPS2PD __m256d _mm256_maskz_cvt_roundps_pd( __mmask8 k, __m256d a, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3;
EVEX-encoded instructions, see Exceptions Type E3.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
**CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2D /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two packed single-precision floating-point values from xmm/m64 to two packed signed doubleword integers in mm.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);

**Intel C/C++ Compiler Intrinsic Equivalent**

CVTPS2PI: __m64 _mm_cvtps_pi32(__m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 2D /r CVTSD2SI r32, xmm1/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.</td>
</tr>
<tr>
<td>F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W0 2D /r CVTSD2SI r32, xmm1/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W1 2D /r CVTSD2SI r64, xmm1/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W0 2D /r CVTSD2SI r32, xmm1/m644(ier)</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 2D /r CVTSD2SI r64, xmm1/m644(ier)</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.</td>
</tr>
</tbody>
</table>

NOTES:
1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VCVTSD2SI (EVEX encoded version)
IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    FI
(V)CVTSD2SI
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    FI
 Intel C/C++ Compiler Intrinsic Equivalent
VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvtsd_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvtsd_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3;
EVEX-encoded instructions, see Exceptions Type E3NF.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F:W1 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F:W1 5A /r VCVTSD2SS xmm1 {k1}[z], xmm2, xmm3/m64{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts a double-precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single-precision floating-point value in the destination operand. When the “convert-from” operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

**VCVTSD2SS (EVEX encoded version)**

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

IF k1[0] or *no writemask*
    THEN DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
    FI;

DEST[MAXVL-1:128] ← 0

**VCVTSD2SS (VEX.128 encoded version)**

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[MAXVL-1:128] ← 0

**CVTSD2SS (128-bit Legacy SSE version)**

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAXVL-1:32] Unmodified *)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSD2SS __m128 _mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 _mm_maskz_cvtsd_ss(__mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 _mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_mask_cvt_roundsd_ss(__mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a, __m128d b, int r);
CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128d b)

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 2A / r CVTSI2SD xmm1, r32/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one signed doubleword integer from r32/m32 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>F2 REX.w 0F 2A / r CVTSI2SD xmm1, r/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F.W0 2A / r VCVTSI2SD xmm1, xmm2, r/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F.W1 2A / r VCVTSI2SD xmm1, xmm2, r/m64</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W0 2A / r VCVTSI2SD xmm1, xmm2, r/m32</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W1 2A / r VCVTSI2SD xmm1, xmm2, r/m64[er]</td>
<td>C</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

NOTES:
1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
**Operation**

**VCVTSD2SD (EVEX encoded version)**

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
    FI;

DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**VCVTSD2SD (VEX.128 encoded version)**

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
    FI;

DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**VCVTSD2SD**

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
    ELSE
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
    FI;

DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSD2SD __m128d __mm_cvti32_sd(__m128d s, int a);
VCVTSD2SD __m128d __mm_cvti64_sd(__m128d s, __int64 a);
VCVTSD2SD __m128d __mm_cvt_roundu64_sd(__m128d s, __int64 a, int r);
CVTSD2SD __m128d __mm_cvtis64_sd(__m128d s, __int64 a);
CVTSD2SD __m128d __mm_cvtsi32_sd(__m128d a, int b)

**SIMD Floating-Point Exceptions**

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3 if W1, else Type 5.
EVEX-encoded instructions, see Exceptions Type E3NF if W1, else Type E10NF.
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 2A /r CVTSI2SS xmm1, r/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE</td>
<td>Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m64</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}</td>
<td>C</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

NOTES:
1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>TUPLE1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX512F</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a single-precision floating-point value in the destination operand (first operand). The “convert-from” source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VCVTSS (EVEX encoded version)
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] <- Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] <- Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:128] <- 0

VCVTSS (VEX.128 encoded version)
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] <- Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] <- Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:128] <- 0

VCVTSS (128-bit Legacy SSE version)
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] <- Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] <- Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSS __m128_mm_cvtsi32_ss(__m128 s, int a);
VCVTSS __m128_mm_cvtsi32_round32_ss(__m128 s, int a, int r);
VCVTSS __m128_mm_cvtsi64_ss(__m128 s, __int64 a);
VCVTSS __m128_mm_cvtsi64_round64_ss(__m128 s, __int64 a, int r);
CVTSS __m128_mm_cvtss_round64 __m128 s, __int64 a);
CVTSS __m128_mm_cvtss_round32 __m128 s, __int64 a, int b);

SIMD Floating-Point Exceptions
Precision

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3NF.
CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F:W1G 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 5A /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2 under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts a single-precision floating-point value in the "convert-from" source operand to a double-precision floating-point value in the destination operand. When the "convert-from" source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

**VCVTSS2SD (EVEX encoded version)**

IF k1[0] or "no writemask"
    THEN DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0]);
ELSE
    IF "merging-masking"
        THEN *DEST[63:0] remains unchanged*;
    ELSE
        THEN DEST[63:0] = 0;
    FI;
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
**VCVTSS2SD (VEX.128 encoded version)**

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**VCVTSS2SD (128-bit Legacy SSE version)**

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);
VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a, __m128 b);
VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a, __m128 b);
CVTSS2SD __m128d_mm_cvtss_sd(__m128d a, __m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 2D r /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.</td>
</tr>
<tr>
<td>F3 REX.w 0F 2D r /r</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W0 2D r /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W1 2D r /r</td>
<td>A</td>
<td>V/N.E.²</td>
<td>AVX</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 2D r /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W1 2D r /r</td>
<td>B</td>
<td>V/N.E.²</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.</td>
</tr>
</tbody>
</table>

### Notes:
1. Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ($2^{w-1}$, where $w$ represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation
VCVTSS2SI (EVEX encoded version)
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;

(V)CVTSS2SI (Legacy and VEX.128 encoded version)
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSS2SI int _mm_cvtss_i32(__m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32(__m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64(__m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64(__m128 a, int r);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3; additionally
#UD
    If VEX.vvvv != 1111B.
EVEX-encoded instructions, see Exceptions Type E3NF.
CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 [k1]{z}, xmm2/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 [k1]{z}, ymm2/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 E6 /r VCVTTPD2DQ ymm1 [k1]{z}, zmm2/m512/m64bcst[5ae]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

**VEX.256 encoded version:** The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
**Operation**

VCVTTPD2DQ (EVEX encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
   i ← j * 32
   k ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] ← 0
      FI
   FI
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
VCVTTPD2DQ (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
                ELSE
                    DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI
    ENDFOR
    DEST[MAXVL-1:VL/2] ← 0

VCVTTPD2DQ (VEX.256 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[MAXVL-1:128] ← 0

VCVTTPD2DQ (VEX.128 encoded version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAXVL-1:64] ← 0

CVTTPD2DQ (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] ← 0
DEST[MAXVL-1:128] (unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2DQ __m256i _mm256_cvttpd_epi32( __m512d a);
VCVTTPD2DQ __m256i _mm256_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm256_maskz_cvttpd_epi32( __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm256_cvtt_roundpd_epi32( __m512d a, int sae);
VCVTTPD2DQ __m256i _mm256_mask_cvtt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i _mm256_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm256_maskz_cvttpd_epi32( __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm_maskz_cvttpd_epi32( __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm256_cvtt_roundpd_epi32( __m256d src);
VCVTTPD2DQ __m128i _mm_cvttpd_epi32 ( __m128d src);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;
EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2C r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two packer double-precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm using truncation.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM reg (w)</td>
<td>ModRM r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

\[
\text{DEST}[31:0] \leftarrow \text{Convert Double-Precision Floating Point To Integer32_Truncate} \left( \text{SRC}[63:0] \right);
\]

\[
\text{DEST}[63:32] \leftarrow \text{Convert Double-Precision Floating Point To Integer32_Truncate} \left( \text{SRC}[127:64] \right);
\]

### Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI: \_m64 \_mm_cvtpd_pi32(\_m128 d a)

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Mode Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.F3.0F:W1G 5B /r VCVTTPS2DQ xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>AVX</td>
<td>Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.F3.0F:W1G 5B /r VCVTTPS2DQ ymm1, ymm2/m256</td>
<td>A V/V</td>
<td>AVX</td>
<td>Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F:W0 5B /r VCVTTPS2DQ xmm1 (k1)[z], xmm2/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F:W0 5B /r VCVTTPS2DQ ymm1 (k1)[z], ymm2/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F:W0 5B /r VCVTTPS2DQ zmm1 (k1)[z], zmm2/m512/m32bcst (sa)</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation
VCVTTPS2DQ (EVEX encoded versions) when src operand is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)
\( i \leftarrow j * 32 \)
IF \( k1[j] \) OR *no writemask*
THEN \( \text{DEST}[i+31:i] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[i+31:i])} \)
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
\( \text{DEST}[i+31:i] \leftarrow 0 \)
FI
FI;
ENDFOR
\( \text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0 \)

VCVTTPS2DQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR \( j \leftarrow 0 \) TO \( 15 \)
\( i \leftarrow j * 32 \)
IF \( k1[j] \) OR *no writemask*
THEN
IF (EVEX.b = 1)
THEN
\( \text{DEST}[i+31:i] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])} \)
ELSE
\( \text{DEST}[i+31:i] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[i+31:i])} \)
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
\( \text{DEST}[i+31:i] \leftarrow 0 \)
FI;
ENDFOR
\( \text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0 \)

VCVTTPS2DQ (VEX.256 encoded version)
\( \text{DEST}[31:0] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])} \)
\( \text{DEST}[63:32] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:32])} \)
\( \text{DEST}[95:64] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[95:64])} \)
\( \text{DEST}[127:96] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[127:96])} \)
\( \text{DEST}[159:128] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[159:128])} \)
\( \text{DEST}[191:160] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[191:160])} \)
\( \text{DEST}[223:192] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[223:192])} \)
\( \text{DEST}[255:224] \leftarrow \text{Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[255:224])} \)
VCVTTPS2DQ (VEX.128 encoded version)
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] ← 0

VCVTTPS2DQ (128-bit Legacy SSE version)
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPS2DQ __m512i _mm512_cvtps_epi32( __m512 a);
VCVTTPS2DQ __m512i _mm512_mask_cvtps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_maskz_cvtps_epi32( __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_cvtt_roundps_epi32( __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_mask_cvtt_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m256i _mm256_maskz_cvtps_epi32( __mmask8 k, __m256 a);
VCVTTPS2DQ __m256i _mm256_cvtps_epi32( __m256 a);
VCVTTPS2DQ __m128i _mm_maskz_cvtps_epi32( __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm_mask_cvtps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm128_maskz_cvtps_epi32( __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm128_cvtps_epi32( __m128 a)

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2; additionally
EVEX-encoded instructions, see Exceptions Type E2.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2C /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two single-precision floating-point values from xmm/m64 to two signed doubleword signed integers in mm using truncation.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI: __m64 _mm_cvttps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 2C Ir</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>CVTTSD2SI r32, xmm1/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX.W 0F 2C Ir</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>CVTTSD2SI r64, xmm1/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F:W0 2C Ir</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>CVTTSD2SI r32, xmm1/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F:W1 2C Ir</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>CVTTSD2SI r64, xmm1/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F:W0 2C Ir</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>CVTTSD2SI r32, xmm1/m64{sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F:W1 2C Ir</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>CVTTSD2SI r64, xmm1/m64{sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:
1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.
Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

(V)CVTSD2SI (All versions)

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);

ELSE

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSD2SI int _mm_cvttsd_i32( __m128d a);
VCVTSD2SI int _mm_cvtt_roundsd_i32( __m128d a, int sae);
VCVTSD2SI __int64 _mm_cvttsd_i64( __m128d a);
VCVTSD2SI __int64 _mm_cvtt_roundsd_i64( __m128d a, int sae);
CVTTSD2SI int _mm_cvttsd_si32( __m128d a);
CVTTSD2SI __int64 _mm_cvttsd_si64( __m128d a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.
**CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 2C /r CVTTSS2SI r32, xmm1/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W0 2C /r CVTTSS2SI r32, xmm1/m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W1 2C /r CVTTSS2SI r64, xmm1/m32</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 2C /r CVTTSS2SI r32, xmm1/m32{sae}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W1 2C /r CVTTSS2SI r64, xmm1/m32{sae}</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation
(V)CVTTSS2SI (All versions)
IF 64-Bit Mode and OperandSize = 64 THEN
    DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
ELSE
    DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
FI;

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTSS2SI int_mm_cvttss_i32(__m128 a);
VCVTTSS2SI int_mm_cvtt_roundss_i32(__m128 a, int sae);
VCVTTSS2SI __int64_mm_cvttss_i64(__m128 a);
VCVTTSS2SI __int64_mm_cvtt_roundss_i64(__m128 a, int sae);
CVTTSS2SI int_mm_cvttss_si32(__m128 a);
CVTTSS2SI __int64_mm_cvttss_si64(__m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
See Exceptions Type 3; additionally
#UD If VEX.vvvv != 1111B.
EVEX-encoded instructions, see Exceptions Type E3NF.
CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

### Description
Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

IF OperandSize = 16 (* CWD instruction *)
THEN
   DX ← SignExtend(AX);
ELSE IF OperandSize = 32 (* CDQ instruction *)
   EDX ← SignExtend(EAX); Fl;
ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
   RDX ← SignExtend(RAX); Fl;
FI;

### Flags Affected
None

### Exceptions (All Operating Modes)

- #UD If the LOCK prefix is used.
DAA—Decimal Adjust AL after Addition

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

IF 64-Bit Mode

THEN

#UD;

ELSE

old_AL ← AL;
old_CF ← CF;
CF ← 0;
IF (((AL AND 0FH) > 9) or AF = 1)

THEN

AL ← AL + 6;
CF ← old_CF or (Carry from AL ← AL + 6);
AF ← 1;
ELSE

AF ← 0;
FI;

IF ((old_AL > 99H) or (old_CF = 1))

THEN

AL ← AL + 60H;
CF ← 1;
ELSE

CF ← 0;
FI;

FI;

### Example

**ADD AL, BL**  
Before: AL=79H BL=35H EFLAGS(0SZAPC)=XXXXXX  
After: AL=AEH BL=35H EFLAGS(0SZAPC)=110000

**DAA**  
Before: AL=AEH BL=35H EFLAGS(0SZAPC)=110000  
After: AL=14H BL=35H EFLAGS(0SZAPC)=X00111

**DAA**  
Before: AL=2EH BL=35H EFLAGS(0SZAPC)=110000  
After: AL=34H BL=35H EFLAGS(0SZAPC)=X00101
**Flags Affected**
The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

**Protected Mode Exceptions**
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**
#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**
#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**
#UD If the LOCK prefix is used.

**64-Bit Mode Exceptions**
#UD If in 64-bit mode.
**DAS—Decimal Adjust AL after Subtraction**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2F</td>
<td>DAS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Decimal adjust AL after subtraction.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

**Operation**

IF 64-Bit Mode THEN
    #UD;
ELSE
    old_AL ← AL;
    old_CF ← CF;
    CF ← 0;
    IF (((AL AND 0FH) > 9) or AF = 1) THEN
        AL ← AL - 6;
        CF ← old_CF or (Borrow from AL ← AL - 6);
        AF ← 1;
    ELSE
        AF ← 0;
    FI;
    IF ((old_AL > 99H) or (old_CF = 1)) THEN
        AL ← AL - 60H;
        CF ← 1;
    FI;
FI;

**Example**

SUB AL, BL Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = X0000000
After: AL = E8H, BL = 47H, EFLAGS(OSZAPC) = 01011101
DAA Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 01011111
After: AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X101111

**Flags Affected**

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.
Protected Mode Exceptions
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
DEC—Decrement by 1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE /1</td>
<td>DEC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m8 by 1.</td>
</tr>
<tr>
<td>REX + FE /1</td>
<td>DEC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Decrement r/m8 by 1.</td>
</tr>
<tr>
<td>FF /1</td>
<td>DEC r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m16 by 1.</td>
</tr>
<tr>
<td>FF /1</td>
<td>DEC r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m32 by 1.</td>
</tr>
<tr>
<td>REX.W + FF /1</td>
<td>DEC r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Decrement r/m64 by 1.</td>
</tr>
<tr>
<td>48+rw</td>
<td>DEC r16</td>
<td>0</td>
<td>N.E.</td>
<td>Valid</td>
<td>Decrement r16 by 1.</td>
</tr>
<tr>
<td>48+rd</td>
<td>DEC r32</td>
<td>0</td>
<td>N.E.</td>
<td>Valid</td>
<td>Decrement r32 by 1.</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRMr/m (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Operation
DEST ← DEST - 1;

Flags Affected
The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0) If the destination operand is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
DIV—Unsigned Divide

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /6</td>
<td>DIV r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned divide AX by r/m8, with result stored in AL ← Quotient, AH ← Remainder.</td>
</tr>
<tr>
<td>REX + F6 /6</td>
<td>DIV r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned divide AX by r/m8, with result stored in AL ← Quotient, AH ← Remainder.</td>
</tr>
<tr>
<td>F7 /6</td>
<td>DIV r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned divide AX by r/m16, with result stored in AX ← Quotient, DX ← Remainder.</td>
</tr>
<tr>
<td>F7 /6</td>
<td>DIV r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned divide EDX:EAX by r/m32, with result stored in EAX ← Quotient, EDX ← Remainder.</td>
</tr>
<tr>
<td>REX.W + F7 /6</td>
<td>DIV r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned divide RDX:RAX by r/m64, with result stored in RAX ← Quotient, RDX ← Remainder.</td>
</tr>
</tbody>
</table>

**NOTES:**
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-15.

**Table 3-15. DIV Action**

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
<th>Remainder</th>
<th>Maximum Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word/byte</td>
<td>AX</td>
<td>r/m8</td>
<td>AL</td>
<td>AH</td>
<td>255</td>
</tr>
<tr>
<td>Doubleword/word</td>
<td>DX:AX</td>
<td>r/m16</td>
<td>AX</td>
<td>DX</td>
<td>65,535</td>
</tr>
<tr>
<td>Quadword/doubleword</td>
<td>EDX:EAX</td>
<td>r/m32</td>
<td>EAX</td>
<td>EDX</td>
<td>$2^{32} - 1$</td>
</tr>
<tr>
<td>Doublequadword/quadword</td>
<td>RDX:RAX</td>
<td>r/m64</td>
<td>RAX</td>
<td>RDX</td>
<td>$2^{64} - 1$</td>
</tr>
</tbody>
</table>
**Operation**

IF SRC = 0
   THEN #DE; Fi; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
   THEN
      temp ← AX / SRC;
      IF temp > FFH
         THEN #DE; (* Divide error *)
         ELSE
            AL ← temp;
            AH ← AX MOD SRC;
      FI;
ELSE IF OperandSize = 16 (* Doubleword/word operation *)
   THEN
      temp ← DX:AX / SRC;
      IF temp > FFFFH
         THEN #DE; (* Divide error *)
         ELSE
            AX ← temp;
            DX ← DX:AX MOD SRC;
      FI;
ELSE IF Operands = 32 (* Quadword/doubleword operation *)
   THEN
      temp ← EDX:EAX / SRC;
      IF temp > FFFFFFFFH
         THEN #DE; (* Divide error *)
         ELSE
            EAX ← temp;
            EDX ← EDX:EAX MOD SRC;
      FI;
ELSE IF 64-Bit Mode and Operands = 64 (* Doublequadword/quadword operation *)
   THEN
      temp ← RDX:RAX / SRC;
      IF temp > FFFFFFFFFFFFFFH
         THEN #DE; (* Divide error *)
         ELSE
            RAX ← temp;
            RDX ← RDX:RAX MOD SRC;
      FI;
FI;

**Flags Affected**
The CF, OF, SF, ZF, AF, and PF flags are undefined.
**Protected Mode Exceptions**

- **#DE**: If the source operand (divisor) is 0
  - If the quotient is too large for the designated register.
- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#DE**: If the source operand (divisor) is 0.
  - If the quotient is too large for the designated register.
- **#GP**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#UD**: If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- **#DE**: If the source operand (divisor) is 0.
  - If the quotient is too large for the designated register.
- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made.
- **#UD**: If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)**: If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)**: If the memory address is in a non-canonical form.
- **#DE**: If the source operand (divisor) is 0
  - If the quotient is too large for the designated register.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.
DIVPD—Divide Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5E /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Divide packed double-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 5E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 5E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 5E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 5E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 5E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Divide packed double-precision floating-point values in zmm2 by packed double-precision FP values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD divide of the double-precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand (the second operand) is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.128 encoded version: The first source operand (the second operand) is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination are zeroed.

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL-1:128) of the corresponding destination are unmodified.
Operation

VDIVPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*

THEN

SET_RM(EVEX.RC); ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1

ELSE

SET_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[63:0]

ELSE

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[i+63:i]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

ENDFOR

DEST[MAXVL-1:VL] ← 0

VDIVPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]


DEST[MAXVL-1:256] ← 0;

VDIVPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]


DEST[MAXVL-1:128] ← 0;

DIVPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]


DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VDIVPD __m512d _mm512_div_pd( __m512d a, __m512d b);
VDIVPD __m512d _mm512_mask_div_pd( __m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d _mm512_maskz_div_pd( __mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d _mm256_mask_div_pd( __m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm128_div_pd( __m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm128_mask_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm128_div_round_pd( __m512d a, __m512d b, int);
VDIVPD __m128d _mm128_mask_div_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m128d _mm128_maskz_div_round_pd( __mmask8 k, __m512d a, __m512d b, int);
DIVPD __m256d _mm256_div_pd( __m256d a, __m256d b);
DIVPD __m128d _mm128_div_pd( __m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
### DIVPS—Divide Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5E /r DIVPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Divide packed single-precision floating-point values in xmm1 by packed single-precision floating-point values in xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Divide packed single-precision floating-point values in zmm2 by packed single-precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD divide of the four, eight or sixteen packed single-precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single-precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

**EVEX encoded versions:** The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

VDIVPS (EVEX encoded versions)

 KL, VL = (4, 128), (8, 256), (16, 512)
 IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
     THEN
         SET_RM(EVEX.RM);
     ELSE
         SET_RM(MXCSR.RM);
 FI;
 FOR j ← 0 TO KL-1
     i ← j * 32
     IF k1[j] OR *no writemask*
         THEN
             IF (EVEX.b = 1) AND (SRC2 *is memory*)
                 THEN
                     DEST[i+31:i] ← SRC1[i+31:i] / SRC2[31:0]
                 ELSE
                     DEST[i+31:i] ← SRC1[i+31:i] / SRC2[i+31:i]
             FI;
         ELSE
             IF *merging masking* ; merging masking
                 THEN *DEST[i+31:i] remains unchanged*
             ELSE ; zeroing masking
                 DEST[i+31:i] ← 0
             FI
     FI
 ENDFOR
 DEST[MAXVL-1:VL] ← 0

VDIVPS (VEX.256 encoded version)

 DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
 DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
 DEST[MAXVL-1:256] ← 0;

VDIVPS (VEX.128 encoded version)

 DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
 DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
 DEST[MAXVL-1:128] ← 0;
DIVPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPS __m512 _mm512_div_ps( __m512 a, __m512 b);
VDIVPS __m512 __mm512_mask_div_ps( __m512 s, __mmask16 k, __m512 a, __m512 b);
VDIVPS __m512 _mm512_maskz_div_ps( __mmask16 k, __m512 a, __m512 b);
VDIVPD __m256d __mm256_mask_div_pd( __m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d __mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d __mm128_mask_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d __mm128_maskz_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPS __m512 __mm512_div_round_ps( __m512 a, __m512 b, int);
VDIVPS __m512 __mm512_mask_div_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m512 __mm512_maskz_div_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m256 __mm256_div_ps( __m256 a, __m256 b);
VDIVPS __m128 __mm_div_ps( __m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
DIVSD—Divide Scalar Double-Precision Floating-Point Value

**Opcode/Instruction** | **Op / En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description** |
--- | --- | --- | --- | --- |
F2 0F 5E /r | A | V/V | SSE2 | Divide low double-precision floating-point value in xmm1 by low double-precision floating-point value in xmm2/m64. |
VEX.NDS.LIG.F2.0F:WIG 5E /r | B | V/V | AVX | Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64. |
EVEX.NDS.LIG.F2.0F:W1 5E /r | C | V/V | AVX512F | Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64. |

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VDIVSD (EVEX encoded version)

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

IF k1[0] or *no writemask*
    THEN
        DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
    FI;

DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

VDIVSD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

DIVSD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd( __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd( __m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd( __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd( __mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd( __m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
**DIVSS—Divide Scalar Single-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5E /r DIVSS xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>SSE</td>
<td>Divide low single-precision floating-point value in xmm1 by low single-precision floating-point value in xmm2/m32.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32</td>
<td>B V/V</td>
<td>AVX</td>
<td>Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W0 5E /r VDIVSS xmm1 <a href="e">k1</a>, xmm2, xmm3/m32[er]</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (w)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:reg (w)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:reg (w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**
Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an XMM register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an XMM register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VDIVSS (EVEX encoded version)
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
                ELSE ; zeroing-masking
                    THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[MAXVL-1:128] ← 0

VDIVSS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[MAXVL-1:128] ← 0

DIVSS (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] / SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss( __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss( __m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss( __mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
DPPD — Dot Product of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 OF 3A 41 /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Selectively multiply packed DP floating-point values from xmm1 with packed DP floating-point values from xmm2, add and selectively store the packed DP floating-point values to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.WIG 41 /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Selectively multiply packed DP floating-point values from xmm2 with packed DP floating-point values from xmm3, add and selectively store the packed DP floating-point values to xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Conditionally multiplies the packed double-precision floating-point values in the destination operand (first operand) with the packed double-precision floating-point values in the source (second operand) depending on a mask extracted from bits [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

The two resulting double-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [1:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding qword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPD follows the NaN forwarding rules stated in the Software Developer’s Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

If VDPPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.
Operation

DP_primitive (SRC1, SRC2)

IF (imm8[4] = 1)
    THEN Temp1[63:0] ← DEST[63:0] * SRC[63:0]; // update SIMD exception flags
    ELSE Temp1[63:0] ← +0.0; Fl;
IF (imm8[5] = 1)
    THEN Temp1[127:64] ← DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] ← +0.0; Fli;
/* if unmasked exception reported, execute exception handler*/

Temp2[63:0] ← Temp1[63:0] + Temp1[127:64]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

IF (imm8[0] = 1)
    THEN DEST[63:0] ← Temp2[63:0];
    ELSE DEST[63:0] ← +0.0; Fli;
IF (imm8[1] = 1)
    THEN DEST[127:64] ← Temp2[63:0];
    ELSE DEST[127:64] ← +0.0; Fli;

DPPD (128-bit Legacy SSE version)

DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)

VDPPD (VEX.128 encoded version)

DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] ← 0

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

DPPD: __m128d _mm_dp_pd (__m128d a, __m128d b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal
Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched.

Other Exceptions

See Exceptions Type 2; additionally
#UD If VEX.L= 1.
DPPS — Dot Product of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 40 /r ib DPPS xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Selectively multiply packed SP floating-point values from xmm1 with packed SP floating-point values from xmm2, add and selectively store the packed SP floating-point values or zero values to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.WIG 40 /r ib VDPPS xmm1,xmm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm2/mem selectively add and store to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F3A.WIG 40 /r ib VDPPS ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed single-precision floating-point values from ymm2 with packed SP floating point values from ymm3/mem, selectively add pairs of elements and store to ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single-precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in Imm8[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

The four resulting single-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [3:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding dword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPS follows the NaN forwarding rules stated in the Software Developer’s Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
**Operation**

DPPS (128-bit Legacy SSE version)
DEST[127:0] $\leftarrow$ DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)

VDPPS (VEX.128 encoded version)
DEST[127:0] $\leftarrow$ DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] $\leftarrow$ 0

VDPPS (VEX.256 encoded version)
DEST[127:0] $\leftarrow$ DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] $\leftarrow$ DP_Primitive(SRC1[255:128], SRC2[255:128]);

**Flags Affected**

None
Intel C/C++ Compiler Intrinsic Equivalent

(V)DPPS: __m128_mm_dp_ps (__m128 a, __m128 b, const int mask);
VDPPS: __m256_mm256_dp_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal
Exceptions are determined separately for each add and multiply operation, in the order of their execution. Unmasked exceptions will leave the destination operands unchanged.

Other Exceptions
See Exceptions Type 2.
**EMMS—Empty MMX Technology State**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 77</td>
<td>EMMS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set the x87 FPU tag word to empty.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

```plaintext
x87FPUTagWord ← FFFFH;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```c
void _mm_empty()
```

**Flags Affected**

None

**Protected Mode Exceptions**

- #UD: If CR0.EM[bit 2] = 1.
- #NM: If CR0.TS[bit 3] = 1.
- #MF: If there is a pending FPU exception.
- #UD: If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
ENTER—Make Stack Frame for Procedure Parameters

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB iw 00</td>
<td>ENTER imm16, 0</td>
<td>II</td>
<td>Valid</td>
<td>Valid</td>
<td>Create a stack frame for a procedure.</td>
</tr>
<tr>
<td>CB iw 01</td>
<td>ENTER imm16,1</td>
<td>II</td>
<td>Valid</td>
<td>Valid</td>
<td>Create a stack frame with a nested pointer for a procedure.</td>
</tr>
<tr>
<td>CB iw ib</td>
<td>ENTER imm16, imm8</td>
<td>II</td>
<td>Valid</td>
<td>Valid</td>
<td>Create a stack frame with nested pointers for a procedure.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>II</td>
<td>iw</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Creates a stack frame (comprising of space for dynamic storage and 1-32 frame pointer storage) for a procedure. The first operand (imm16) specifies the size of the dynamic storage in the stack frame (that is, the number of bytes of dynamically allocated on the stack for the procedure). The second operand (imm8) gives the lexical nesting level (0 to 31) of the procedure. The nesting level (imm8 mod 32) and the OperandSize attribute determine the size in bytes of the storage space for frame pointers.

The nesting level determines the number of frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. The default size of the frame pointer is the StackAddrSize attribute, but can be overridden using the 66H prefix. Thus, the OperandSize attribute determines the size of each frame pointer that will be copied into the stack frame and the data being transferred from SP/ESP/RSP register into the BP/EBP/RBP register.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded. Use of 66H prefix changes frame pointer operand size to 16 bits.

When the 66H prefix is used and causing the OperandSize attribute to be less than the StackAddrSize, software is responsible for the following:

- The companion LEAVE instruction must also use the 66H prefix,
- The value in the RBP/EBP register prior to executing “66H ENTER” must be within the same 16KByte region of the current stack pointer (RSP/ESP), such that the value of RBP/EBP after “66H ENTER” remains a valid address in the stack. This ensures “66H LEAVE” can restore 16-bits of data from the stack.
Operation

AllocSize ← imm16;
NestingLevel ← imm8 MOD 32;
IF (OperandSize = 64)
    THEN
        Push(RBP); (* RSP decrements by 8 *)
        FrameTemp ← RSP;
    ELSE IF OperandSize = 32
        THEN
            Push(EBP); (* (E)SP decrements by 4 *)
            FrameTemp ← ESP; FI;
    ELSE (* OperandSize = 16 *)
        Push(BP); (* RSP or (E)SP decrements by 2 *)
        FrameTemp ← SP;
    FI;
IF NestingLevel = 0
    THEN GOTO CONTINUE;
FI;
IF (NestingLevel > 1)
    THEN FOR i ← 1 to (NestingLevel - 1)
        DO
            IF (OperandSize = 64)
                THEN
                    RBP ← RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            THEN
                                EBP ← EBP - 4;
                                Push([EBP]); (* Doubleword push *)
                            ELSE (* StackSize = 16 *)
                                BP ← BP - 4;
                                Push([BP]); (* Doubleword push *)
                            FI;
                        FI;
                    ELSE (* OperandSize = 16 *)
                        IF StackSize = 32
                            THEN
                                EBP ← EBP - 2;
                                Push([EBP]); (* Word push *)
                            ELSE (* StackSize = 16 *)
                                BP ← BP - 2;
                                Push([BP]); (* Word push *)
                            FI;
                    FI;
                FI;
            ELSE (* OperandSize = 16 *)
                IF StackSize = 32
                    THEN
                        EBP ← EBP - 2;
                        Push([EBP]); (* Word push *)
                    ELSE (* StackSize = 16 *)
                        BP ← BP - 2;
                        Push([BP]); (* Word push *)
                    FI;
            FI;
        OD;
    FI;
IF (OperandSize = 64) (* nestinglevel 1 *)
    THEN
        Push(FrameTemp); (* Quadword push and RSP decrements by 8 *)
    ELSE IF OperandSize = 32
        THEN
            Push(RBP);
            FrameTemp ← RSP;
        ELSE IF OperandSize = 32
            THEN
                Push(EBP);
                FrameTemp ← ESP;
            ELSE (* OperandSize = 16 *)
                Push(BP);
                FrameTemp ← SP;
            FI;
        ELSE (* OperandSize = 16 *)
            IF StackSize = 32
                THEN
                    EBP ← EBP - 4;
                    Push([EBP]); (* Doubleword push *)
                ELSE (* StackSize = 16 *)
                    BP ← BP - 4;
                    Push([BP]); (* Doubleword push *)
                FI;
            ELSE (* OperandSize = 16 *)
                IF StackSize = 32
                    THEN
                        EBP ← EBP - 2;
                        Push([EBP]); (* Word push *)
                    ELSE (* StackSize = 16 *)
                        BP ← BP - 2;
                        Push([BP]); (* Word push *)
                    FI;
            FI;
        FI;
    FI;
    ELSE IF NestingLevel = 0
        THEN GOTO CONTINUE;
    FI;
    ELSE IF (NestingLevel > 1)
        THEN FOR i ← 1 to (NestingLevel - 1)
            DO
                IF (OperandSize = 64)
                    THEN
                        RBP ← RBP - 8;
                        Push([RBP]); (* Quadword push *)
                    ELSE IF OperandSize = 32
                        THEN
                            IF StackSize = 32
                                THEN
                                    EBP ← EBP - 4;
                                    Push([EBP]); (* Doubleword push *)
                                ELSE (* StackSize = 16 *)
                                    BP ← BP - 4;
                                    Push([BP]); (* Doubleword push *)
                                FI;
                            FI;
                        ELSE (* OperandSize = 16 *)
                            IF StackSize = 32
                                THEN
                                    EBP ← EBP - 2;
                                    Push([EBP]); (* Word push *)
                                ELSE (* StackSize = 16 *)
                                    BP ← BP - 2;
                                    Push([BP]); (* Word push *)
                                FI;
                    FI;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ← EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 2;
                            Push([BP]); (* Word push *)
                        FI;
                FI;
            OD;
        FI;
    FI;
    ELSE IF NestingLevel = 0
        THEN GOTO CONTINUE;
    FI;
    ELSE IF (NestingLevel > 1)
        THEN FOR i ← 1 to (NestingLevel - 1)
            DO
                IF (OperandSize = 64)
                    THEN
                        RBP ← RBP - 8;
                        Push([RBP]); (* Quadword push *)
                    ELSE IF OperandSize = 32
                        THEN
                            IF StackSize = 32
                                THEN
                                    EBP ← EBP - 4;
                                    Push([EBP]); (* Doubleword push *)
                                ELSE (* StackSize = 16 *)
                                    BP ← BP - 4;
                                    Push([BP]); (* Doubleword push *)
                                FI;
                            FI;
                        ELSE (* OperandSize = 16 *)
                            IF StackSize = 32
                                THEN
                                    EBP ← EBP - 2;
                                    Push([EBP]); (* Word push *)
                                ELSE (* StackSize = 16 *)
                                    BP ← BP - 2;
                                    Push([BP]); (* Word push *)
                                FI;
                        FI;
                    FI;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ← EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 2;
                            Push([BP]); (* Word push *)
                        FI;
                FI;
            OD;
        FI;
    FI;
THEN
    Push(FrameTemp);Fl; (* Doubleword push and (E)SP decrements by 4 *)
ELSE (* OperandSize = 16 *)
    Push(FrameTemp); (* Word push and RSP|ESP|SP decrements by 2 *)
FI;
CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
        RBP ← FrameTemp;
        RSP ← RSP − AllocSize;
    ELSE IF OperandSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP − AllocSize; Fl;
        ELSE (* OperandSize = 16 *)
            BP ← FrameTemp[15:1]; (* Bits 16 and above of applicable RBP/EBP are unmodified *)
            SP ← SP − AllocSize;
        FI;
    END;

Flags Affected
None.

Protected Mode Exceptions
#SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#SS If the new value of the SP or ESP register is outside the stack segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD If the LOCK prefix is used.
EXTRACTPS—Extract Packed Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8</td>
<td>A</td>
<td>VV</td>
<td>SSE4_1</td>
<td>Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Extracts a single-precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single-precision floating-point value is stored in the low 32-bits of the destination operand.

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

VEX.vvvv/EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

**Operation**

VEXTRACTPS (EVEX and VEX.128 encoded version)

\[
\text{SRC\_OFFSET} \leftarrow \text{Imm8}[1:0]
\]

IF (64-Bit Mode and DEST is register)

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{SRC}[127:0] \gg (\text{SRC\_OFFSET}*32)) \text{ AND } 0FFFFFFFh \\
\text{DEST}[63:32] & \leftarrow 0
\end{align*}
\]

ELSE

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{SRC}[127:0] \gg (\text{SRC\_OFFSET}*32)) \text{ AND } 0FFFFFFFh \\
\text{FI}
\end{align*}
\]
EXTRACTPS (128-bit Legacy SSE version)
SRC_OFFSET ← IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] ← (SRC[127:0] >> (SRC_OFFSET*32)) AND 0xFFFFFFFFh
    DEST[63:32] ← 0
ELSE
    DEST[31:0] ← (SRC[127:0] >> (SRC_OFFSET*32)) AND 0xFFFFFFFFh
FI

Intel C/C++ Compiler Intrinsic Equivalent
EXTRACTPS int _mm_extract_ps (__m128 a, const int nidx);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instructions, see Exceptions Type 5; Additionally
EVEX-encoded instructions, see Exceptions Type E9NF.
#UD IF VEX.L = 0.
#UD IF VEX.vvvv != 1111B or EVEX.vvv != 1111B.
F2XM1—Compute $2^x - 1$

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
</table>
| D9 F0  | F2XM1       | Valid       | Valid           | Replace ST(0) with $(2^{ST(0)} - 1)$.

**Description**

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range $-1.0$ to $+1.0$. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-1.0$ to $-0$</td>
<td>$-0.5$ to $-0$</td>
</tr>
<tr>
<td>$-0$</td>
<td>$-0$</td>
</tr>
<tr>
<td>$+0$</td>
<td>$+0$</td>
</tr>
<tr>
<td>$+0$ to $+1.0$</td>
<td>$+0$ to $1.0$</td>
</tr>
</tbody>
</table>

Values other than 2 can be exponentiated using the following formula:

$$x^y \leftarrow 2^{(y \times \log_2 x)}$$

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

$$ST(0) \leftarrow (2^{ST(0)} - 1);$$

**FPU Flags Affected**

- $C1$ Set to 0 if stack underflow occurred.
- Set if result was rounded up; cleared otherwise.
- $C0$, $C2$, $C3$ Undefined.

**Floating-Point Exceptions**

- #IS Stack underflow occurred.
- #IA Source operand is an SNaN value or unsupported format.
- #D Source is a denormal value.
- #U Result is too small for destination format.
- #P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FABS—Absolute Value

Description
Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

Table 3-17. Results Obtained from FABS

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞</td>
<td>+ ∞</td>
</tr>
<tr>
<td>− F</td>
<td>+ F</td>
</tr>
<tr>
<td>− 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ F</td>
<td>+ F</td>
</tr>
<tr>
<td>+ ∞</td>
<td>+ ∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:
F Means finite floating-point value.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
ST(0) ← |ST(0)|;

FPU Flags Affected
C1 Set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FADD/FADDP/FIADD—Add

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /0</td>
<td>FADD m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m32fp to ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DC /0</td>
<td>FADD m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m64fp to ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>D8 C0+i</td>
<td>FADD ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(0) to ST(i) and store result in ST(0).</td>
</tr>
<tr>
<td>DC C0+i</td>
<td>FADD ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(i) to ST(0) and store result in ST(i).</td>
</tr>
<tr>
<td>DE C0+i</td>
<td>FADDP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(0) to ST(i), store result in ST(i), and pop the register stack.</td>
</tr>
<tr>
<td>DE C1</td>
<td>FADDP</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(0) to ST(1), store result in ST(1), and pop the register stack.</td>
</tr>
<tr>
<td>DA /0</td>
<td>FIADD m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m32int to ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DE /0</td>
<td>FIADD m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m16int to ST(0) and store result in ST(0).</td>
</tr>
</tbody>
</table>

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward −∞ mode, in which case the result is −0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-18.
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FIADD
    THEN
        DEST ← DEST + ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST ← DEST + SRC;
    FI;

IF Instruction = FADDP
    THEN
        PopRegisterStack;
    FI;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.
C0, C2, C3  Undefined.

**Floating-Point Exceptions**

#IS  Stack underflow occurred.
#IA  Operand is an SNaN value or unsupported format.
    Operands are infinities of unlike sign.
#D   Source operand is a denormal value.
#U   Result is too small for destination format.
#O   Result is too large for destination format.
#P   Value cannot be represented exactly in destination format.

**Table 3-18. FADD/FADDP/FIADD Results**

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>−F or −I</td>
<td>−∞</td>
<td>−F</td>
<td>SRC</td>
<td>SRC</td>
<td>± F or ± 0</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td>−∞</td>
<td>DEST</td>
<td>−0</td>
<td>± 0</td>
<td>DEST</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>−∞</td>
<td>DEST</td>
<td>± 0</td>
<td>+ 0</td>
<td>DEST</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+F or + I</td>
<td>−∞</td>
<td>± F or ± 0</td>
<td>SRC</td>
<td>SRC</td>
<td>+ F</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

F  Means finite floating-point value.
I  Means integer.
*  Indicates floating-point invalid-arithmetic-operand (#IA) exception.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FBLD—Load Binary Coded Decimal

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF /4</td>
<td>FBLD m8bcd</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert BCD value to floating-point and push onto the FPU stack.</td>
</tr>
</tbody>
</table>

**Description**

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of −0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

TOP ← TOP − 1;
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);

**FPU Flags Affected**

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack overflow occurred.

**Protected Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FBSTP—Store BCD Integer and Pop

**Description**

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>$F \leq -1$</td>
<td>$-D$</td>
</tr>
<tr>
<td>$-1 &lt; F &lt; 0$</td>
<td>**</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$+0$</td>
<td>$+0$</td>
</tr>
<tr>
<td>$+0 &lt; F &lt; +1$</td>
<td>**</td>
</tr>
<tr>
<td>$F \geq +1$</td>
<td>$+D$</td>
</tr>
<tr>
<td>$+\infty$ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>*</td>
</tr>
</tbody>
</table>

**NOTES:**

- $F$ Means finite floating-point value.
- $D$ Means packed-BCD number.
- * Indicates floating-point invalid-operation (#IA) exception.
- ** $\pm0$ or $\pm1$, depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an $\infty$, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

```
DEST ← BCD(ST(0));
PopRegisterStack;
```

**FPU Flags Affected**

- **C1**
  - Set to 0 if stack underflow occurred.
  - Set if result was rounded up; cleared otherwise.
- **C0, C2, C3**
  - Undefined.
Floating-Point Exceptions

#IS Stack underflow occurred.
#IA Converted value that exceeds 18 BCD digits in length.
Source operand is an SNaN, QNaN, ±∞, or in an unsupported format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If a segment register is being loaded with a segment selector that points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FCHS—Change Sign

Description
Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

Table 3-20. FCHS Results

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>+∞</td>
</tr>
<tr>
<td>−F</td>
<td>+F</td>
</tr>
<tr>
<td>−0</td>
<td>+0</td>
</tr>
<tr>
<td>+0</td>
<td>−0</td>
</tr>
<tr>
<td>+F</td>
<td>−F</td>
</tr>
<tr>
<td>+∞</td>
<td>−∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:
* F means finite floating-point value.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
SignBit(ST(0)) ← NOT (SignBit(ST(0)));

FPU Flags Affected
C1 Set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
FCLEX/FNCLEX—Clear Exceptions

Description
Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

IA-32 Architecture Compatibility
When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS register.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

\[
\begin{align*}
\text{FPUStatusWord}[0:7] & \leftarrow 0; \\
\text{FPUStatusWord}[15] & \leftarrow 0;
\end{align*}
\]

FPU Flags Affected
The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions
None

Protected Mode Exceptions

- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#UD**: If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

---

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B DB E2</td>
<td>FCLEX</td>
<td>Valid</td>
<td>Valid</td>
<td>Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>DB E2</td>
<td>FNCLEX</td>
<td>Valid</td>
<td>Valid</td>
<td>Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

**NOTES:**

* See IA-32 Architecture Compatibility section below.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FCMOVcc—Floating-Point Conditional Move

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DA C0+i</td>
<td>FCMOVB ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>DA C8+i</td>
<td>FCMOVE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if equal (ZF=1).</td>
</tr>
<tr>
<td>DA D0+i</td>
<td>FCMOVBE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>DA D8+i</td>
<td>FCMOVU ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if unordered (PF=1).</td>
</tr>
<tr>
<td>DB C0+i</td>
<td>FCMOVNB ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below (CF=0).</td>
</tr>
<tr>
<td>DB C8+i</td>
<td>FCMOVNE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not equal (ZF=0).</td>
</tr>
<tr>
<td>DB D0+i</td>
<td>FCMOVNBE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>DB D8+i</td>
<td>FCMOVNU ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not unordered (PF=0).</td>
</tr>
</tbody>
</table>

NOTES:
* See IA-32 Architecture Compatibility section below.

Description
Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic os given in the Description column above and in Chapter 8 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume I. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOVcc instructions. Software can check if the FCMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS” in this chapter). If both the CMOV and FPU feature bits are set, the FCMOVcc instructions are supported.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility
The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation
IF condition TRUE
THEN ST(0) ← ST(i);
FI;

FPU Flags Affected
C1 Set to 0 if stack underflow occurred.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.

Integer Flags Affected
None.
Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FCOM/FCOMP/FCOMPP—Compare Floating Point Values**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /2</td>
<td>FCOM m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m32fp.</td>
</tr>
<tr>
<td>DC /2</td>
<td>FCOM m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m64fp.</td>
</tr>
<tr>
<td>D8 D0+i</td>
<td>FCOM ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i).</td>
</tr>
<tr>
<td>D8 D1</td>
<td>FCOM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(1).</td>
</tr>
<tr>
<td>D8 /3</td>
<td>FCOMP m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m32fp and pop register stack.</td>
</tr>
<tr>
<td>DC /3</td>
<td>FCOMP m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m64fp and pop register stack.</td>
</tr>
<tr>
<td>D8 D8+i</td>
<td>FCOMP ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i) and pop register stack.</td>
</tr>
<tr>
<td>D8 D9</td>
<td>FCOMP</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(1) and pop register stack.</td>
</tr>
<tr>
<td>DE D9</td>
<td>FCOMPP</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(1) and pop register stack twice.</td>
</tr>
</tbody>
</table>

**Description**

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that –0.0 is equal to +0.0.

<table>
<thead>
<tr>
<th>Condition</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; SRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST(0) &lt; SRC</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST(0) = SRC</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered*</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOTES:**

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to “unordered.” If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
Operation

CASE (relation of operands) OF
   ST > SRC: C3, C2, C0 ← 000;
   ST < SRC: C3, C2, C0 ← 001;
   ST = SRC: C3, C2, C0 ← 100;
ESAC;

IF ST(0) or SRC = NaN or unsupported format
   THEN
       #IA
       IF FPUControlWord.IM = 1
           THEN
               C3, C2, C0 ← 111;
       FI;
   FI;

IF Instruction = FCOMP
   THEN
       PopRegisterStack;
   FI;

IF Instruction = FCOMPP
   THEN
       PopRegisterStack;
       PopRegisterStack;
   FI;

FPU Flags Affected
C1 Set to 0.
C0, C2, C3 See table on previous page.

Floating-Point Exceptions
#IS Stack underflow occurred.
#IA One or both operands are NaN values or have unsupported formats.
       Register is marked empty.
#D One or both operands are denormal values.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
       If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
       current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB F0+i</td>
<td>FCOMI ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i) and set status flags accordingly.</td>
</tr>
<tr>
<td>DF F0+i</td>
<td>FCOMIP ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i), set status flags accordingly, and pop register stack.</td>
</tr>
<tr>
<td>DB E8+i</td>
<td>FUCOMI ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly.</td>
</tr>
<tr>
<td>DF E8+i</td>
<td>FUCOMIP ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack.</td>
</tr>
</tbody>
</table>

**Description**

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

**Table 3-22. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results**

<table>
<thead>
<tr>
<th>Comparison Results*</th>
<th>ZF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0 &gt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST0 &lt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST0 = ST(i)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered**</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

** Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions set the OF, SF and AF flags to zero in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.
Operation

CASE (relation of operands) OF
  ST(0) > ST(i): ZF, PF, CF ← 000;
  ST(0) < ST(i): ZF, PF, CF ← 001;
  ST(0) = ST(i): ZF, PF, CF ← 100;
ESAC;

IF Instruction is FCOMI or FCOMIP
  THEN
    IF ST(0) or ST(i) = NaN or unsupported format
      THEN
        #IA
        IF FPUControlWord.IM = 1
          THEN
            ZF, PF, CF ← 111;
          FL;
        FI;
      FI;
    FI;

IF Instruction is FUCOMI or FUCOMIP
  THEN
    IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format
      THEN
        ZF, PF, CF ← 111;
      ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)
        #IA;
        IF FPUControlWord.IM = 1
          THEN
            ZF, PF, CF ← 111;
          FL;
        FI;
      FL;
    FI;

IF Instruction is FCOMIP or FUCOMIP
  THEN
    PopRegisterStack;
  FI;

FPU Flags Affected

C1 Set to 0.
C0, C2, C3 Not affected.

Floating-Point Exceptions

#IS Stack underflow occurred.
#IA (FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats.
  (FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.
Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FCOS—Cosine**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compag/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FF</td>
<td>FCOS</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(0) with its approximate cosine.</td>
</tr>
</tbody>
</table>

**Description**

Computes the approximate cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range $-2^63$ to $+2^63$. The following table shows the results obtained when taking the cosine of various classes of numbers.

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>*</td>
</tr>
<tr>
<td>$-F$</td>
<td>$-1$ to $+1$</td>
</tr>
<tr>
<td>$0$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$0$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+F$</td>
<td>$-1$ to $+1$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range $-2^63$ to $+2^63$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of $2\pi$. However, even within the range $-2^63$ to $+2^63$, inaccurate results can occur because the finite approximation of $\pi$ used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a discussion of the proper value to use for $\pi$ in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF $|ST(0)| < 2^63$

THEN

C2 ← 0;

ST(0) ← FCOS(ST(0)); // approximation of cosine

ELSE (* Source operand is out-of-range *)

C2 ← 1;

Fl;
FPU Flags Affected
C1    Set to 0 if stack underflow occurred.
     Set if result was rounded up; cleared otherwise.
     Undefined if C2 is 1.
C2    Set to 1 if outside range \((-2^{63} < \text{source operand} < +2^{63})\); otherwise, set to 0.
C0, C3 Undefined.

Floating-Point Exceptions
#IS    Stack underflow occurred.
#IA    Source operand is an SNaN value, \(\infty\), or unsupported format.
#D     Source is a denormal value.
#P     Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#NM    CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF    If there is a pending x87 FPU exception.
#UD    If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FDECSTP—Decrement Stack-Top Pointer

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F6</td>
<td>FDECSTP</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement TOP field in FPU status word.</td>
</tr>
</tbody>
</table>

**Description**

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF TOP = 0
  THEN TOP ← 7;
ELSE TOP ← TOP – 1;
FI;

**FPU Flags Affected**

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FDIV/FDIVP/FIDIV—Divide

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /6</td>
<td>FDIV m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m32fp and store result in ST(0).</td>
</tr>
<tr>
<td>DC /6</td>
<td>FDIV m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m64fp and store result in ST(0).</td>
</tr>
<tr>
<td>DB F0+i</td>
<td>FDIV ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by ST(i) and store result in ST(0).</td>
</tr>
<tr>
<td>DC F8+i</td>
<td>FDIVP ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.</td>
</tr>
<tr>
<td>DE F8+i</td>
<td>FDIVP</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.</td>
</tr>
<tr>
<td>DE F9</td>
<td>FDIVP</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(1) by ST(0), store result in ST(1), and pop the register stack.</td>
</tr>
<tr>
<td>DA /6</td>
<td>FIDIV m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m32int and store result in ST(0).</td>
</tr>
<tr>
<td>DE /6</td>
<td>FIDIV m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m16int and store result in ST(0).</td>
</tr>
</tbody>
</table>

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.
FDIV/FDIVP/FIDIV—Divide

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF SRC = 0
    THEN
        #Z;
    ELSE
        IF Instruction is FIDIV
            THEN
                DEST ← DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
            ELSE (* Source operand is floating-point value *)
                DEST ← DEST / SRC;
        FI;
    FI;

IF Instruction = FDIVP
    THEN
        PopRegisterStack;
    FI;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.
C0, C2, C3  Undefined.

---

**Table 3-24. FDIV/FDIVP/FIDIV Results**

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>−∞ −F −0 +0 +F +∞ NaN</td>
</tr>
<tr>
<td>−F</td>
<td>−∞ +0 +F +0 −0 −F −∞ NaN</td>
</tr>
<tr>
<td>−I</td>
<td>+∞ +F +0 −0 −F −∞ NaN</td>
</tr>
<tr>
<td>−0</td>
<td>+∞ ** * * ** −∞ NaN</td>
</tr>
<tr>
<td>+0</td>
<td>−∞ ** * * ** +∞ NaN</td>
</tr>
<tr>
<td>+I</td>
<td>−∞ −F −0 +0 +F +∞ NaN</td>
</tr>
<tr>
<td>+F</td>
<td>−∞ −F −0 +0 +F +∞ NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>* −0 −0 +0 +0 * NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN NaN NaN NaN NaN NaN NaN NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F  Means finite floating-point value.
I  Means integer.
*  Indicates floating-point invalid-arithmetic-operand (#IA) exception.
** Indicates floating-point zero-divide (#Z) exception.
Floating-Point Exceptions

#IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
  ±∞ / ±∞; ±0 / ±0
#D Source is a denormal value.
#Z DEST / ±0, where DEST is not equal to ±0.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FDIVR/FDIVRP/FIDIVR—Reverse Divide

### Description
Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an \( \infty \) of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.
When the source operand is an integer 0, it is treated as a +0. This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF DEST = 0
THEN
  #Z;
ELSE
  IF Instruction = FIDIVR
  THEN
    DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;
  ELSE (* Source operand is floating-point value *)
    DEST ← SRC / DEST;
  FI;
FI;

IF Instruction = FDIVRP
THEN
  PopRegisterStack;
FI;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.
  Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

---

**Table 3-25. FDIVR/FDIVRP/FIDIVR Results**

<table>
<thead>
<tr>
<th>SRC</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>−∞</td>
<td>−∞</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>+0</td>
<td>+F</td>
<td>**</td>
<td>**</td>
<td>−F</td>
<td>−0</td>
<td>NaN</td>
</tr>
<tr>
<td>−I</td>
<td>+0</td>
<td>+F</td>
<td>**</td>
<td>**</td>
<td>−F</td>
<td>−0</td>
<td>NaN</td>
</tr>
<tr>
<td>−0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>−0</td>
<td>−0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+I</td>
<td>−0</td>
<td>−F</td>
<td>**</td>
<td>**</td>
<td>+F</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>−0</td>
<td>−F</td>
<td>**</td>
<td>**</td>
<td>+F</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>−∞</td>
<td>−∞</td>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-arithmetic-operand (#IA) exception.
** Indicates floating-point zero-divide (#Z) exception.
Floating-Point Exceptions
#IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
±∞ / ±∞; ±0 / ±0
#D Source is a denormal value.
#Z SRC / ±0, where SRC is not equal to ±0.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.
**FFREE—Free Floating-Point Register**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD C0+i</td>
<td>FFREE ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Sets tag for ST(i) to empty.</td>
</tr>
</tbody>
</table>

**Description**
Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[ \text{TAG}(i) \leftarrow 11B; \]

**FPU Flags Affected**

C0, C1, C2, C3 undefined.

**Floating-Point Exceptions**

None

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
**FICOM/FICOMP—Compare Integer**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE /2</td>
<td>FICOM m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m16int.</td>
</tr>
<tr>
<td>DA /2</td>
<td>FICOM m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m32int.</td>
</tr>
<tr>
<td>DE /3</td>
<td>FICOMP m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m16int and pop stack register.</td>
</tr>
<tr>
<td>DA /3</td>
<td>FICOMP m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m32int and pop stack register.</td>
</tr>
</tbody>
</table>

**Description**

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

<table>
<thead>
<tr>
<th>Condition</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; SRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST(0) &lt; SRC</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST(0) = SRC</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

These instructions perform an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to “unordered.”

The sign of zero is ignored, so that –0.0 ← +0.0.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

CASE (relation of operands) OF
  ST(0) > SRC: C3, C2, C0 ← 000;
  ST(0) < SRC: C3, C2, C0 ← 001;
  ST(0) = SRC: C3, C2, C0 ← 100;
  Unordered: C3, C2, C0 ← 111;
ESAC;

IF Instruction = FICOMP
  THEN
    PopRegisterStack;
  FI;

**FPU Flags Affected**

C1          Set to 0.
C0, C2, C3  See table on previous page.

**Floating-Point Exceptions**

#IS        Stack underflow occurred.
#IA        One or both operands are NaN values or have unsupported formats.
#D         One or both operands are denormal values.
Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FILD—Load Integer

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF /0</td>
<td>FILD m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m16int onto the FPU register stack.</td>
</tr>
<tr>
<td>DB /0</td>
<td>FILD m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m32int onto the FPU register stack.</td>
</tr>
<tr>
<td>DF /5</td>
<td>FILD m64int</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m64int onto the FPU register stack.</td>
</tr>
</tbody>
</table>

Description
Converting the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation
TOP ← TOP − 1;
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);

FPU Flags Affected
C1 Set to 1 if stack overflow occurred; set to 0 otherwise.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack overflow occurred.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0_EM[bit 2] or CR0_TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0_EM[bit 2] or CR0_TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FINCSTP—Increment Stack-Top Pointer

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F7</td>
<td>FINCSTP</td>
<td>Valid</td>
<td>Valid</td>
<td>Increment the TOP field in the FPU status register.</td>
</tr>
</tbody>
</table>

**Description**

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
\text{IF TOP} = 7 \\
\quad \text{THEN TOP} \leftarrow 0; \\
\quad \text{ELSE TOP} \leftarrow \text{TOP} + 1; \\
\text{FI;}
\]

**FPU Flags Affected**

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FINIT/FNINIT—Initialize Floating-Point Unit

Description
Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility
When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers. This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

Operation
FPUCtrlWord ← 037FH;
FPUStrWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUIInstPointer ← 0;
FPULastInstOpcode ← 0;

FPU Flags Affected
C0, C1, C2, C3 set to 0.

Floating-Point Exceptions
None
Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FIST/FISTP—Store Integer

### Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF /2</td>
<td>FIST m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m16int.</td>
</tr>
<tr>
<td>DB /2</td>
<td>FIST m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m32int.</td>
</tr>
<tr>
<td>DF /3</td>
<td>FISTP m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m16int and pop register stack.</td>
</tr>
<tr>
<td>DB /3</td>
<td>FISTP m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m32int and pop register stack.</td>
</tr>
<tr>
<td>DF /7</td>
<td>FISTP m64int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m64int and pop register stack.</td>
</tr>
</tbody>
</table>

#### Table 3-27. FIST/FISTP Results

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>F ≤ −1</td>
<td>−1</td>
</tr>
<tr>
<td>−1 &lt; F &lt; −0</td>
<td>**</td>
</tr>
<tr>
<td>−0</td>
<td>0</td>
</tr>
<tr>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>+0 &lt; F &lt; +1</td>
<td>**</td>
</tr>
<tr>
<td>F ≥ +1</td>
<td>+1</td>
</tr>
<tr>
<td>+ ∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>*</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-operation (#IA) exception.
** 0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ±∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
Operation

DEST ← Integer(ST(0));

IF Instruction = FISTP
    THEN
        PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Indicates rounding direction of if the inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
Set to 0 otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Converted value is too large for the destination format.
Source operand is an SNaN, QNaN, ±∞, or unsupported format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
FISTTP—Store Integer with Truncation

**Description**

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>( F \leq -1 )</td>
<td>−1</td>
</tr>
<tr>
<td>( -1 &lt; F &lt; +1 )</td>
<td>0</td>
</tr>
<tr>
<td>( F \leq +1 )</td>
<td>+1</td>
</tr>
<tr>
<td>+∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>*</td>
</tr>
</tbody>
</table>

**Table 3-28. FISTTP Results**

**NOTES:**

F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-operation (#IA) exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

DEST ← ST;
pop ST;

**Flags Affected**

C1 is cleared; C0, C2, C3 undefined.

**Numeric Exceptions**

Invalid, Stack Invalid (stack underflow), Precision.

**Protected Mode Exceptions**

#GP(0) If the destination is in a nonwritable segment.
For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#NM If CR0.EF[bit 2] = 1.
If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
If the LOCK prefix is used.
Real Address Mode Exceptions
GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM If CR0.EM[bit 2] = 1.
    If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
    If the LOCK prefix is used.

Virtual 8086 Mode Exceptions
GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM If CR0.EM[bit 2] = 1.
    If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
    If the LOCK prefix is used.
#PF(fault-code) For a page fault.
#AC(0) For unaligned memory reference if the current privilege is 3.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
    current privilege level is 3.
    If the LOCK prefix is used.
FLD—Load Floating Point Value

Description
Pushes the source operand onto the FPU register stack. The source operand can be in single-precision, double-precision, or double extended-precision floating-point format. If the source operand is in single-precision or double-precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

IF SRC is ST(i)
    THEN
        temp ← ST(i);

Fl;

TOP ← TOP − 1;

IF SRC is memory-operand
    THEN
        ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* SRC is ST(i) *)
        ST(0) ← temp;

Fl;

FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow or overflow occurred.

#IA Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)).

#D Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /0</td>
<td>FLD m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m32fp onto the FPU register stack.</td>
</tr>
<tr>
<td>DD /0</td>
<td>FLD m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m64fp onto the FPU register stack.</td>
</tr>
<tr>
<td>DB /5</td>
<td>FLD m80fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m80fp onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 C0+i</td>
<td>FLD ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Push ST(i) onto the FPU register stack.</td>
</tr>
</tbody>
</table>
Protected Mode Exceptions

#GP(0) If destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
**FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 EB</td>
<td>FLD1</td>
<td>Valid</td>
<td>Valid</td>
<td>Push +1.0 onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 E9</td>
<td>FLDL2T</td>
<td>Valid</td>
<td>Valid</td>
<td>Push log₂10 onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 EA</td>
<td>FLDL2E</td>
<td>Valid</td>
<td>Valid</td>
<td>Push log₂e onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 EB</td>
<td>FLDPI</td>
<td>Valid</td>
<td>Valid</td>
<td>Push π onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 EC</td>
<td>FLDLG2</td>
<td>Valid</td>
<td>Valid</td>
<td>Push log₁₀2 onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 ED</td>
<td>FLDLN2</td>
<td>Valid</td>
<td>Valid</td>
<td>Push logₑ2 onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 EE</td>
<td>FLDZ</td>
<td>Valid</td>
<td>Valid</td>
<td>Push +0.0 onto the FPU register stack.</td>
</tr>
</tbody>
</table>

**NOTES:**

* See IA-32 Architecture Compatibility section below.

**Description**

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, log₂10, log₂e, π, log₁₀2, and logₑ2. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled “Approximation of Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the π constant.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

**Operation**

\[
\text{TOP} \leftarrow \text{TOP} - 1; \\
\text{ST}(0) \leftarrow \text{CONSTANT};
\]

**FPU Flags Affected**

- C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

- #IS Stack overflow occurred.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FLDCW—Load x87 FPU Control Word**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /5</td>
<td>FLDCW m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Load FPU control word from m2byte.</td>
</tr>
</tbody>
</table>

**Description**

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU’s mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

FPUControlWord ← SRC;

**FPU Flags Affected**

C0, C1, C2, C3 undefined.

**Floating-Point Exceptions**

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next “waiting” floating-point instruction.

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#NM** CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#NM** CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#UD** If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**FLDENV—Load x87 FPU Environment**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /4</td>
<td>FLDENV m14/28byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Load FPU environment from m14byte or m28byte.</td>
</tr>
</tbody>
</table>

**Description**

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];

**FPU Flags Affected**

The C0, C1, C2, C3 flags are loaded.

**Floating-Point Exceptions**

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next “waiting” floating-point instruction.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.

#UD If the LOCK prefix is used.
FMUL/FMULP/FIMUL—Multiply

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /1</td>
<td>FMUL m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m32fp and store result in ST(0).</td>
</tr>
<tr>
<td>DC /1</td>
<td>FMUL m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m64fp and store result in ST(0).</td>
</tr>
<tr>
<td>DC C8+i</td>
<td>FMUL ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by ST(i) and store result in ST(0).</td>
</tr>
<tr>
<td>DE C8+i</td>
<td>FMULP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack.</td>
</tr>
<tr>
<td>DE C9</td>
<td>FMULP</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack.</td>
</tr>
<tr>
<td>DA /1</td>
<td>FIMUL m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m32int and store result in ST(0).</td>
</tr>
<tr>
<td>DE /1</td>
<td>FIMUL m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m16int and store result in ST(0).</td>
</tr>
</tbody>
</table>

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞. When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FIMUL  
THEN  
DEST ← DEST * ConvertToDoubleExtendedPrecisionFP(SRC);  
ELSE (* Source operand is floating-point value *)  
DEST ← DEST * SRC;  
FI;  

IF Instruction = FMULP  
THEN  
PopRegisterStack;  
FI;  

**FPU Flags Affected**

C1  
Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.

C0, C2, C3  
Undefined.

**Floating-Point Exceptions**

#IS  
Stack underflow occurred.

#IA  
Operand is an SNaN value or unsupported format.  
One operand is ±0 and the other is ±∞.

#D  
Source operand is a denormal value.

#U  
Result is too small for destination format.

#O  
Result is too large for destination format.

#P  
Value cannot be represented exactly in destination format.

**Table 3-29. FMUL/FMULP/FIMUL Results**

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>−∞</td>
<td>+∞</td>
<td>*</td>
<td>+0</td>
<td>−0</td>
<td>−F</td>
<td>−∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>+∞</td>
<td>+F</td>
<td>+0</td>
<td>−0</td>
<td>−F</td>
<td>−∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>−1</td>
<td>+∞</td>
<td>+F</td>
<td>+0</td>
<td>−0</td>
<td>−F</td>
<td>−∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>−0</td>
<td>+0</td>
<td>*</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>*</td>
<td>−0</td>
<td>−0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>−∞</td>
<td>−F</td>
<td>−0</td>
<td>+0</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+F</td>
<td>−∞</td>
<td>−F</td>
<td>−0</td>
<td>+0</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>−∞</td>
<td>−∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>
Protected Mode Exceptions
#GP(0)    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
         If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)    If a memory operand effective address is outside the SS segment limit.
#NM       CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0)    If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD       If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP       If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS       If a memory operand effective address is outside the SS segment limit.
#NM       CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD       If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0)    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)    If a memory operand effective address is outside the SS segment limit.
#NM       CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0)    If alignment checking is enabled and an unaligned memory reference is made.
#UD       If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0)    If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)    If the memory address is in a non-canonical form.
#NM       CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF       If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0)    If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD       If the LOCK prefix is used.
FNOP—No Operation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 D0</td>
<td>FNOP</td>
<td>Valid</td>
<td>Valid</td>
<td>No operation is performed.</td>
</tr>
</tbody>
</table>

**Description**

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register and the FPU Instruction Pointer.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**FPU Flags Affected**

C0, C1, C2, C3 undefined.

**Floating-Point Exceptions**

None

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FPATAN—Partial Arctangent

Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than +π.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point (−X,Y) is in the second quadrant, resulting in an angle between π/2 and π, while a point (X,−Y) is in the fourth quadrant, resulting in an angle between 0 and −π/2. A point (−X,−Y) is in the third quadrant, giving an angle between −π/2 and −π.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

Table 3-30. FPATAN Results

| ST(0) | −∞ | −F | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | NaN
|-------|----|----|------|------|------|------|------|------|------|------|------|
| ST(1) | −∞ | −3π/4* | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | −π/2 | NaN
| −F | -p | −π to −π/2 | −π/2 | −π/2 | −π/2 | −0 | NaN
| −0 | -p | −p* | −0* | 0 | NaN
| +0 | +p | +p* | +0* | +0 | NaN
| +F | +p | +π to +π/2 | +π/2 | +π/2 | +0 | NaN
| +∞ | +3π/4* | +π/2 | +π/2 | +π/2 | +π/2 | NaN
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN

NOTES:

F Means finite floating-point value.

Table 8-10 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, specifies that the ratios 0/0 and ±∞ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the 0/0 or ±∞ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, arctangent(0,0) etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

0 ≤ |ST(1)| < |ST(0)| < +∞
Operation

\[ ST(1) \leftarrow \arctan(ST(1) / ST(0)) ; \]
\[ \text{PopRegisterStack}; \]

FPU Flags Affected

C1 \hspace{1em} \text{Set to 0 if stack underflow occurred.}
\hspace{1em} \text{Set if result was rounded up; cleared otherwise.}

C0, C2, C3 \hspace{1em} \text{Undefined.}

Floating-Point Exceptions

#IS \hspace{1em} \text{Stack underflow occurred.}
#IA \hspace{1em} \text{Source operand is an SNaN value or unsupported format.}
#D \hspace{1em} \text{Source operand is a denormal value.}
#U \hspace{1em} \text{Result is too small for destination format.}
#P \hspace{1em} \text{Value cannot be represented exactly in destination format.}

Protected Mode Exceptions

#NM \hspace{1em} \text{CR0.EM[bit 2] or CR0.TS[bit 3] = 1.}
#MF \hspace{1em} \text{If there is a pending x87 FPU exception.}
#UD \hspace{1em} \text{If the LOCK prefix is used.}

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
**FPREM—Partial Remainder**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F8</td>
<td>FPREM</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1).</td>
</tr>
</tbody>
</table>

**Description**

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or modulus), and stores the result in ST(0). The remainder represents the following value:

\[
\text{Remainder} \leftarrow ST(0) - (Q \times ST(1))
\]

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of \([ST(0) / ST(1)]\) toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>Table 3-31. FPREM Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ST(1)</strong></td>
</tr>
<tr>
<td><strong>ST(0)</strong></td>
</tr>
<tr>
<td>-∞</td>
</tr>
<tr>
<td>-∞</td>
</tr>
<tr>
<td>-F</td>
</tr>
<tr>
<td>-0</td>
</tr>
<tr>
<td>+0</td>
</tr>
<tr>
<td>+F</td>
</tr>
<tr>
<td>+∞</td>
</tr>
<tr>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
* Indicates floating-point invalid-arithmetic-operand (#IA) exception.
** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is \(-∞\), the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name "partial remainder" because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the partial remainder. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU.
status word. This information is important in argument reduction for the tangent function (using a modulus of \(\pi/4\)), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
D \leftarrow \text{exponent}(ST(0)) - \text{exponent}(ST(1));
\]

**IF** \(D < 64\)

**THEN**

\[
Q \leftarrow \text{Integer}(\text{TruncateTowardZero}(ST(0) / ST(1)));
ST(0) \leftarrow ST(0) - (ST(1) \times Q);
C2 \leftarrow 0;
C0, C3, C1 \leftarrow \text{LeastSignificantBits}(Q); (* Q2, Q1, Q0 *)
\]

**ELSE**

\[
C2 \leftarrow 1;
N \leftarrow \text{An implementation-dependent number between 32 and 63};
QQ \leftarrow \text{Integer}(\text{TruncateTowardZero}((ST(0) / ST(1)) / 2^{(D - N)}));
ST(0) \leftarrow ST(0) - (ST(1) \times QQ \times 2^{(D - N)});
\]

**FI**;

**FPU Flags Affected**

- **C0** Set to bit 2 (Q2) of the quotient.
- **C1** Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
- **C2** Set to 0 if reduction complete; set to 1 if incomplete.
- **C3** Set to bit 1 (Q1) of the quotient.

**Floating-Point Exceptions**

- **#IS** Stack underflow occurred.
- **#IA** Source operand is an SNaN value, modulus is 0, dividend is \(\infty\), or unsupported format.
- **#D** Source operand is a denormal value.
- **#U** Result is too small for destination format.

**Protected Mode Exceptions**

- **#NM** \(\text{CR0.EM}[\text{bit 2}]\) or \(\text{CR0.TS}[\text{bit 3}] = 1\).
- **#MF** If there is a pending x87 FPU exception.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FPREM1—Partial Remainder

### Description
Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

\[
\text{Remainder} \leftarrow ST(0) - (Q \times ST(1))
\]

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of \([ST(0) / ST(1)]\) toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

#### Table 3-32. FPREM1 Results

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>ST(0)</td>
<td>±F or −0</td>
<td>**</td>
<td>**</td>
<td>±F or −0</td>
<td>ST(0)</td>
<td>NaN</td>
</tr>
<tr>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>*</td>
<td>*</td>
<td>−0</td>
<td>-0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>ST(0)</td>
<td>±F or +0</td>
<td>**</td>
<td>**</td>
<td>±F or +0</td>
<td>ST(0)</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
- F Means finite floating-point value.
- * Indicates floating-point invalid-arithmetic-operand (#IA) exception.
- ** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is −∞, the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the “Operation” section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the partial remainder. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU.
status word. This information is important in argument reduction for the tangent function (using a modulus of \( \pi/4 \)), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
D \leftarrow \text{exponent(ST(0))} - \text{exponent(ST(1))};
\]

IF \( D < 64 \)

THEN

\[
Q \leftarrow \text{Integer} \left( \text{RoundTowardNearestInteger} \left( \frac{\text{ST(0)}}{\text{ST(1)}} \right) \right);
\]

\[
\text{ST(0)} \leftarrow \text{ST(0)} - (\text{ST(1)} \times Q);
\]

\[
C2 \leftarrow 0;
\]

\[
C0, C3, C1 \leftarrow \text{LeastSignificantBits}(Q); (* \text{ Q2, Q1, Q0 }*)
\]

ELSE

\[
C2 \leftarrow 1;
\]

\[
N \leftarrow \text{An implementation-dependent number between 32 and 63};
\]

\[
QQ \leftarrow \text{Integer} \left( \text{TruncateTowardZero} \left( \frac{\text{ST(0)}}{\text{ST(1)}} \right) / 2^{(D - N)} \right);
\]

\[
\text{ST(0)} \leftarrow \text{ST(0)} - (\text{ST(1)} \times QQ \times 2^{(D - N)});
\]

FI;

**FPU Flags Affected**

- \( C0 \) Set to bit 2 (Q2) of the quotient.
- \( C1 \) Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
- \( C2 \) Set to 0 if reduction complete; set to 1 if incomplete.
- \( C3 \) Set to bit 1 (Q1) of the quotient.

**Floating-Point Exceptions**

- \#IS Stack underflow occurred.
- \#IA Source operand is an SNaN value, modulus (divisor) is 0, dividend is \( \infty \), or unsupported format.
- \#D Source operand is a denormal value.
- \#U Result is too small for destination format.

**Protected Mode Exceptions**

- \#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- \#MF If there is a pending x87 FPU exception.
- \#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FPTAN—Partial Tangent

**Description**

Computes the approximate tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than ±2^{63}. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range −2^{63} to +2^{63} can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to +2^{63}, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FPTAN only to arguments reduced accurately in software, to a value smaller in absolute value than 3π/8. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a discussion of the proper value to use for π in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

---

### Table 3-33. FPTAN Results

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞</td>
<td>*</td>
</tr>
<tr>
<td>− F</td>
<td>− F to + F</td>
</tr>
<tr>
<td>− 0</td>
<td>− 0</td>
</tr>
<tr>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ F</td>
<td>− F to + F</td>
</tr>
<tr>
<td>+ ∞</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.
Operation

IF ST(0) < 2^{63} THEN
    C2 ← 0;
    ST(0) ← fptan(ST(0)); // approximation of tan
    TOP ← TOP − 1;
    ST(0) ← 1.0;
ELSE (* Source operand is out-of-range *)
    C2 ← 1;
FI;

FPU Flags Affected
C1 Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
    Set if result was rounded up; cleared otherwise.
C2 Set to 1 if outside range (−2^{63} < source operand < +2^{63}); otherwise, set to 0.
C0, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow or overflow occurred.
#IA Source operand is an SNaN value, ∞, or unsupported format.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
### FRNDINT—Round to Integer

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FC</td>
<td>FRNDINT</td>
<td>Valid</td>
<td>Valid</td>
<td>Round ST(0) to an integer.</td>
</tr>
</tbody>
</table>

#### Description
Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is $\infty$, the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

#### Operation

```
ST(0) ← RoundToIntegralValue(ST(0));
```

#### FPU Flags Affected
- **C1** Set to 0 if stack underflow occurred.
- Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

#### Floating-Point Exceptions
- **#IS** Stack underflow occurred.
- **#IA** Source operand is an SNaN value or unsupported format.
- **#D** Source operand is a denormal value.
- **#P** Source operand is not an integral value.

#### Protected Mode Exceptions
- **#NM** CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#MF** If there is a pending x87 FPU exception.
- **#UD** If the LOCK prefix is used.

#### Real-Address Mode Exceptions
Same exceptions as in protected mode.

#### Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

#### Compatibility Mode Exceptions
Same exceptions as in protected mode.

#### 64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FRSTOR—Restore x87 FPU State**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD /4</td>
<td>FRSTOR m94/108byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Load FPU state from m94byte or m108byte.</td>
</tr>
</tbody>
</table>

**Description**

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
\begin{align*}
\text{FPUControlWord} & \leftarrow \text{SRC}[\text{FPUControlWord}]; \\
\text{FPUStatusWord} & \leftarrow \text{SRC}[\text{FPUStatusWord}]; \\
\text{FPUTagWord} & \leftarrow \text{SRC}[\text{FPUTagWord}]; \\
\text{FPUDataPointer} & \leftarrow \text{SRC}[\text{FPUDataPointer}]; \\
\text{FPUInstructionPointer} & \leftarrow \text{SRC}[\text{FPUInstructionPointer}]; \\
\text{FPULastInstructionOpcode} & \leftarrow \text{SRC}[\text{FPULastInstructionOpcode}]; \\
\text{ST}(0) & \leftarrow \text{SRC}[\text{ST}(0)]; \\
\text{ST}(1) & \leftarrow \text{SRC}[\text{ST}(1)]; \\
\text{ST}(2) & \leftarrow \text{SRC}[\text{ST}(2)]; \\
\text{ST}(3) & \leftarrow \text{SRC}[\text{ST}(3)]; \\
\text{ST}(4) & \leftarrow \text{SRC}[\text{ST}(4)]; \\
\text{ST}(5) & \leftarrow \text{SRC}[\text{ST}(5)]; \\
\text{ST}(6) & \leftarrow \text{SRC}[\text{ST}(6)]; \\
\text{ST}(7) & \leftarrow \text{SRC}[\text{ST}(7)]; \\
\end{align*}
\]

**FPU Flags Affected**

The C0, C1, C2, C3 flags are loaded.

**Floating-Point Exceptions**

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.
Protected Mode Exceptions

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.

Real-Address Mode Exceptions

- **#GP**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#UD**: If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made.
- **#UD**: If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- **#SS(0)**: If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)**: If the memory address is in a non-canonical form.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.
**FSAVE/FNSAVE—Store x87 FPU State**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B DD /6</td>
<td>FSAVE m94/108byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU state to m94byte or m108byte after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.</td>
</tr>
<tr>
<td>DD /6</td>
<td>FNSAVE m94/108byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU environment to m94byte or m108byte without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.</td>
</tr>
</tbody>
</table>

**Description**

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed. After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see "FINIT/FNINIT—Initialize Floating-Point Unit" in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a "clean" FPU to a procedure. The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

For Intel math coprocessors and FPsUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.
**Operation**

(* Save FPU State and Registers *)

```
DEST[FPUCtrl] ← FPUCtrl;
DEST[FPUSt] ← FPUSt;
DEST[FPUTag] ← FPUTag;
DEST[FPUDataPoint] ← FPUDataPoint;
DEST[FPUInstrPoint] ← FPUInstrPoint;
DEST[FPULastInstrOpcode] ← FPULastInstrOpcode;

DEST[ST(0)] ← ST(0);
DEST[ST(1)] ← ST(1);
DEST[ST(2)] ← ST(2);
DEST[ST(3)] ← ST(3);
DEST[ST(4)] ← ST(4);
DEST[ST(5)] ← ST(5);
DEST[ST(6)] ← ST(6);
DEST[ST(7)] ← ST(7);
```

(* Initialize FPU *)

```
FPUCtrl ← 037FH;
FPUSt ← 0;
FPUTag ← FFFFH;
FPUDataPoint ← FFFFH;
FPUInstrPoint ← 0;
FPULastInstrOpcode ← 0;
```

**FPU Flags Affected**
The C0, C1, C2, and C3 flags are saved and then cleared.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

- **#GP(0)** If destination is located in a non-writable segment.
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#NM** CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#NM** CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#UD** If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
FSCALE—Scale

Description

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-34. FSCALE Results

<table>
<thead>
<tr>
<th>ST(1)</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>NaN</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>−0</td>
<td>−F</td>
<td>−F</td>
<td>−F</td>
<td>−F</td>
<td>−∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>+0</td>
<td>+F</td>
<td>+F</td>
<td>+F</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>NaN</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:
F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source’s mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

FXTRACT;
FSCALE;
FSTP ST(1);

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

\[ \text{ST}(0) \leftarrow \text{ST}(0) \cdot 2^{\text{RoundTowardZero}(\text{ST}(1))} \]

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.
Floating-Point Exceptions

#IS Stack underflow occurred.
#IA Source operand is an SNaN value or unsupported format.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
FSIN—Sine

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FE</td>
<td>FSIN</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(0) with the approximate of its sine.</td>
</tr>
</tbody>
</table>

**Description**

Computes an approximation of the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range $-2^{63}$ to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>SRC (ST(0))</th>
<th>DEST (ST(0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>*</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$1 + 1$</td>
</tr>
<tr>
<td>$-0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$+0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$1 + 1$</td>
</tr>
</tbody>
</table>

**NOTES:**
- F Means finite floating-point value.
- * Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range $-2^{63}$ to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of $2\pi$. However, even within the range $-2^{63}$ to $+2^{63}$, inaccurate results can occur because the finite approximation of $\pi$ used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSIN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/4$. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for $\pi$ in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

```
IF -2^{63} < ST(0) < 2^{63} 
THEN 
    C2 ← 0; 
    ST(0) ← fsin(ST(0)); // approximation of the mathematical sin function
ELSE (* Source operand out of range *)
    C2 ← 1;
FI;
```

**FPU Flags Affected**

- **C1** Set to 0 if stack underflow occurred.
  Set if result was rounded up; cleared otherwise.
- **C2** Set to 1 if outside range ($-2^{63}$ < source operand < $+2^{63}$); otherwise, set to 0.
- **C0, C3** Undefined.
Floating-Point Exceptions
#IS Stack underflow occurred.
#IA Source operand is an SNaN value, ∞, or unsupported format.
#D Source operand is a denormal value.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FSINCOS—Sine and Cosine**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FB</td>
<td>FSINCOS</td>
<td>Valid</td>
<td>Valid</td>
<td>Compute the sine and cosine of ST(0); replace ST(0) with the approximate sine, and push the approximate cosine onto the register stack.</td>
</tr>
</tbody>
</table>

**Description**

Computes both the approximate sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range \(-2^{63}\) to \(+2^{63}\). The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0)</td>
<td>ST(1) Cosine</td>
</tr>
<tr>
<td>(-\infty)</td>
<td>*</td>
</tr>
<tr>
<td>(-F)</td>
<td>(-1 + 1)</td>
</tr>
<tr>
<td>(-0)</td>
<td>(+1)</td>
</tr>
<tr>
<td>(+0)</td>
<td>(+1)</td>
</tr>
<tr>
<td>(+F)</td>
<td>(-1 + 1)</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range \(-2^{63}\) to \(+2^{63}\) can be reduced to the range of the instruction by subtracting an appropriate integer multiple of \(2\pi\). However, even within the range \(-2^{63}\) to \(+2^{63}\), inaccurate results can occur because the finite approximation of \(\pi\) used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSINCOS only to arguments reduced accurately in software, to a value smaller in absolute value than \(3\pi/8\). See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for \(\pi\) in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
**Operation**

IF ST(0) < $2^{63}$
THEN
   
   C2 ← 0;
   TEMP ← fcos(ST(0)); // approximation of cosine
   ST(0) ← fsin(ST(0)); // approximation of sine
   TOP ← TOP – 1;
   ST(0) ← TEMP;

ELSE (* Source operand out of range *)
   
   C2 ← 1;

Fl;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs.
Set if result was rounded up; cleared otherwise.

C2 Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.

C0, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow or overflow occurred.

#IA Source operand is an SNaN value, $\infty$, or unsupported format.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FSQRT—Square Root

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FA</td>
<td>FSQRT</td>
<td>Valid</td>
<td>Valid</td>
<td>Computes square root of ST(0) and stores the result in ST(0).</td>
</tr>
</tbody>
</table>

**Description**
Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-37. FSQRT Results**

<table>
<thead>
<tr>
<th>SRC (ST(0))</th>
<th>DEST (ST(0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞</td>
<td>*</td>
</tr>
<tr>
<td>− F</td>
<td>*</td>
</tr>
<tr>
<td>− 0</td>
<td>− 0</td>
</tr>
<tr>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ F</td>
<td>+ F</td>
</tr>
<tr>
<td>+ ∞</td>
<td>+ ∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**
- F Means finite floating-point value.
- * Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
\text{ST}(0) \leftarrow \text{SquareRoot}(\text{ST}(0));
\]

**FPU Flags Affected**

- C1
  - Set to 0 if stack underflow occurred.
  - Set if result was rounded up; cleared otherwise.
- C0, C2, C3
  - Undefined.

**Floating-Point Exceptions**

- #IS Stack underflow occurred.
- #IA Source operand is an SNaN value or unsupported format.
- #D Source operand is a negative value (except for -0).
- #P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FST/FSTP—Store Floating Point Value

Description
The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single-precision or double-precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single-precision or double-precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ±0, ±∞, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value’s identity as a 0, ∞, or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
DEST ← ST(0);
IF Instruction = FSTP
    THEN
        PopRegisterStack;
FI;

FPU Flags Affected
C1 Set to 0 if stack underflow occurred.
Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 ← not round; 1 ← roundup.
C0, C2, C3 Undefined.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /2</td>
<td>FST m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m32fp.</td>
</tr>
<tr>
<td>DD /2</td>
<td>FST m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m64fp.</td>
</tr>
<tr>
<td>DD D0+i</td>
<td>FST ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to ST(i).</td>
</tr>
<tr>
<td>D9 /3</td>
<td>FSTP m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m32fp and pop register stack.</td>
</tr>
<tr>
<td>DD /3</td>
<td>FSTP m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m64fp and pop register stack.</td>
</tr>
<tr>
<td>DB /7</td>
<td>FSTP m80fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m80fp and pop register stack.</td>
</tr>
<tr>
<td>DD D8+i</td>
<td>FSTP ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to ST(i) and pop register stack.</td>
</tr>
</tbody>
</table>
Floating-Point Exceptions
#IS Stack underflow occurred.
#IA If destination result is an SNaN value or unsupported format, except when the destination format is in double extended-precision floating-point format.
#U Result is too small for the destination format.
#O Result is too large for the destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSTCW/FNSTCW—Store x87 FPU Control Word

Description
Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility
When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation
DEST ← FPUControlWord;

FPU Flags Affected
The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions
None.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM \( \text{CR0.EM}[\text{bit 2}] \) or \( \text{CR0.TS}[\text{bit 3}] = 1 \).
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM \( \text{CR0.EM}[\text{bit 2}] \) or \( \text{CR0.TS}[\text{bit 3}] = 1 \).
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM \( \text{CR0.EM}[\text{bit 2}] \) or \( \text{CR0.TS}[\text{bit 3}] = 1 \).
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSTENV/FNSTENV—Store x87 FPU Environment

**Description**

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

**Operation**

\[
\text{DEST}[\text{FPUControlWord}] \leftarrow \text{FPUControlWord};
\text{DEST}[\text{FPUSstatusWord}] \leftarrow \text{FPUSstatusWord};
\text{DEST}[\text{FPUTagWord}] \leftarrow \text{FPUTagWord};
\text{DEST}[\text{FPUDataPointer}] \leftarrow \text{FPUDataPointer};
\text{DEST}[\text{FPUInstructionPointer}] \leftarrow \text{FPUInstructionPointer};
\text{DEST}[\text{FPULastInstructionOpcode}] \leftarrow \text{FPULastInstructionOpcode};
\]

**FPU Flags Affected**

The C0, C1, C2, and C3 are undefined.
Floating-Point Exceptions

None

Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSTSW/FNSTSW—Store x87 FPU Status Word

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B DD 7</td>
<td>FSTSW m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word at m2byte after checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>9B DF E0</td>
<td>FSTSW AX</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>DD 7</td>
<td>FNSTSW m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word at m2byte without checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>DF E0</td>
<td>FNSTSW AX</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

**NOTES:**
* See IA-32 Architecture Compatibility section below.

**Description**
Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled “Branching and Conditional Moves on FPU Condition Codes” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.*) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**
When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

**Operation**
DEST ← FPUStratusWord;

**FPU Flags Affected**
The C0, C1, C2, and C3 are undefined.

**Floating-Point Exceptions**
None
Protected Mode Exceptions

#GP(0)  If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD  If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS  If a memory operand effective address is outside the SS segment limit.

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

Virtual-8086 Mode Exceptions

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment fault occurs.

#UD  If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)  If the memory address is in a non-canonical form.

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF  If there is a pending x87 FPU exception.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD  If the LOCK prefix is used.
FSUB/FSUBP/FISUB—Subtract

**Description**

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-38 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST − SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward −∞ mode, in which case the result is −0. This instruction also guarantees that +0 − (−0) = +0, and that −0 − (+0) = −0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /4</td>
<td>FSUB m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m32fp from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DC /4</td>
<td>FSUB m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m64fp from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>D8 E0+i</td>
<td>FSUB ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(i) from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DC E8+i</td>
<td>FSUB ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from ST(i) and store result in ST(i).</td>
</tr>
<tr>
<td>DE E8+i</td>
<td>FSUBP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from ST(i), store result in ST(i), and pop register stack.</td>
</tr>
<tr>
<td>DE E9</td>
<td>FSUBP</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from ST(1), store result in ST(1), and pop register stack.</td>
</tr>
<tr>
<td>DA /4</td>
<td>FISUB m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m32int from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DE /4</td>
<td>FISUB m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m16int from ST(0) and store result in ST(0).</td>
</tr>
</tbody>
</table>
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FISUB
THEN
  DEST ← DEST − ConvertToDoubleExtendedPrecisionFP(SRC);
ELSE (* Source operand is floating-point value *)
  DEST ← DEST − SRC;
FI;

IF Instruction = FSUBP
THEN
  PopRegisterStack;
FI;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.

C0, C2, C3  Undefined.

**Floating-Point Exceptions**

#IS  Stack underflow occurred.

#IA  Operand is an SNaN value or unsupported format.
    Operands are infinities of like sign.

#D  Source operand is a denormal value.

#U  Result is too small for destination format.

#O  Result is too large for destination format.

#P  Value cannot be represented exactly in destination format.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSUBR/FSUBRP/FISUBR—Reverse Subtract

**Description**

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value ($SRC - DEST = result$).

When the difference between two operands of like sign is 0, the result is $+0$, except for the round toward $-\infty$ mode, in which case the result is $-0$. This instruction also guarantees that $+0 - (-0) = +0$, and that $-0 - (+0) = -0$. When the source operand is an integer 0, it is treated as a $+0$.

When one operand is $\infty$, the result is $\infty$ of the expected sign. If both operands are $\infty$ of the same sign, an invalid-operation exception is generated.
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FISUBR
    THEN
        DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) − DEST;
    ELSE (* Source operand is floating-point value *)
        DEST ← SRC − DEST; FI;
IF Instruction = FSUBRP
    THEN
        PopRegisterStack; FI;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
    Operands are infinities of like sign.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

**Table 3-39. FSUBR/FSUBRP/FISUBR Results**

<table>
<thead>
<tr>
<th>DEST</th>
<th>−∞</th>
<th>src</th>
<th>−F or −I</th>
<th>−0</th>
<th>+0</th>
<th>−F or +I</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>−∞</td>
<td>±F or ±0</td>
<td>−DEST</td>
<td>−DEST</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td>−∞</td>
<td>SRC</td>
<td>±0</td>
<td>+0</td>
<td>SRC</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>−∞</td>
<td>SRC</td>
<td>−0</td>
<td>±0</td>
<td>SRC</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+F</td>
<td>−∞</td>
<td>−F</td>
<td>−DEST</td>
<td>−DEST</td>
<td>±F or ±0</td>
<td>+∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Protected Mode Exceptions

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.

Real-Address Mode Exceptions

- **#GP**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#UD**: If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made.
- **#UD**: If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- **#SS(0)**: If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)**: If the memory address is in a non-canonical form.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#MF**: If there is a pending x87 FPU exception.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.
FTST—TEST

Description
Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

<table>
<thead>
<tr>
<th>Condition</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; 0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST(0) &lt; 0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST(0) = 0.0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This instruction performs an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to “unordered” and the invalid operation exception is generated.

The sign of zero is ignored, so that (– 0.0 ← +0.0).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
CASE (relation of operands) OF
   Not comparable: C3, C2, C0 ← 111;
   ST(0) > 0.0: C3, C2, C0 ← 000;
   ST(0) < 0.0: C3, C2, C0 ← 001;
   ST(0) = 0.0: C3, C2, C0 ← 100;
ESAC;

FPU Flags Affected
C1 Set to 0.
C0, C2, C3 See Table 3-40.

Floating-Point Exceptions
#IS Stack underflow occurred.
#IA The source operand is a NaN value or is in an unsupported format.
#D The source operand is a denormal value.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD E0+i</td>
<td>FUCOM</td>
<td>ST(i)</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i).</td>
</tr>
<tr>
<td>DD E1</td>
<td>FUCOM</td>
<td></td>
<td>Valid</td>
<td>Compare ST(0) with ST(1).</td>
</tr>
<tr>
<td>DD E8+i</td>
<td>FUCOMP</td>
<td>ST(i)</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i) and pop register stack.</td>
</tr>
<tr>
<td>DD E9</td>
<td>FUCOMP</td>
<td></td>
<td>Valid</td>
<td>Compare ST(0) with ST(1) and pop register stack.</td>
</tr>
<tr>
<td>DA E9</td>
<td>FUCOMPP</td>
<td></td>
<td>Valid</td>
<td>Compare ST(0) with ST(1) and pop register stack twice.</td>
</tr>
</tbody>
</table>

Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that –0.0 is equal to +0.0.

### Table 3-41. FUCOM/FUCOMP/FUCOMPP Results

<table>
<thead>
<tr>
<th>Comparison Results*</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0 &gt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST0 &lt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST0 = ST(i)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NOTES:
* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
Operation

CASE (relation of operands) OF
   ST > SRC: C3, C2, C0 ← 000;
   ST < SRC: C3, C2, C0 ← 001;
   ST = SRC: C3, C2, C0 ← 100;
ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
   THEN
      C3, C2, C0 ← 111;
   ELSE (* ST(0) or SRC is SNaN or unsupported format *)
      #IA;
      IF FPUControlWord.IM = 1
         THEN
            C3, C2, C0 ← 111;
      FI;
   FI;

IF Instruction = FUCOMP
   THEN
      PopRegisterStack;
   FI;

IF Instruction = FUCOMPP
   THEN
      PopRegisterStack;
   FI;

FPU Flags Affected
C1 Set to 0 if stack underflow occurred.
C0, C2, C3 See Table 3-41.

Floating-Point Exceptions
#IS Stack underflow occurred.
#IA One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
#D One or both operands are denormal values.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
FXAM—Examine Floating-Point

### Description
Examine the contents of the ST(0) register and set the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

#### Table 3-42. FXAM Results

<table>
<thead>
<tr>
<th>Class</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsupported</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Normal finite number</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Infinity</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Empty</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Denormal number</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

#### Operation

\[ C1 \leftarrow \text{sign bit of ST}; \quad (\ast 0 \text{ for positive, } 1 \text{ for negative} \ast) \]

CASE (class of value or number in ST(0)) OF

- Unsupported: C3, C2, C0 \leftarrow 000;
- NaN: C3, C2, C0 \leftarrow 001;
- Normal: C3, C2, C0 \leftarrow 010;
- Infinity: C3, C2, C0 \leftarrow 011;
- Zero: C3, C2, C0 \leftarrow 100;
- Empty: C3, C2, C0 \leftarrow 101;
- Denormal: C3, C2, C0 \leftarrow 110;

ESAC;

#### FPU Flags Affected

- C1: Sign of value in ST(0).
- C0, C2, C3: See Table 3-42.

#### Floating-Point Exceptions

None

#### Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

#### Real-Address Mode Exceptions

Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FXCH—Exchange Register Contents

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 C8+i</td>
<td>FXCH ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange the contents of ST(0) and ST(i).</td>
</tr>
<tr>
<td>D9 C9</td>
<td>FXCH</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange the contents of ST(0) and ST(1).</td>
</tr>
</tbody>
</table>

**Description**

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF (Number-of-operands) is 1
  THEN
    temp ← ST(0);
    ST(0) ← SRC;
    SRC ← temp;
  ELSE
    temp ← ST(0);
    ST(0) ← ST(1);
    ST(1) ← temp;
  FI;

**FPU Flags Affected**

C1 Set to 0.
C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow occurred.

**Protected Mode Exceptions**

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FXRSTOR—Restore x87 FPU, MMX, XMM, and MXCSR State

**Opcode/Instruction** | **Op/En** | **64-Bit Mode** | **Compat/Leg Mode** | **Description**
---|---|---|---|---
NP 0F AE /1 FXRSTOR m512byte | M | Valid | Valid | Restore the x87 FPU, MMX, XMM, and MXCSR register state from m512byte.
NP REX.W + 0F AE /1 FXRSTOR64 m512byte | M | Valid | N.E. | Restore the x87 FPU, MMX, XMM, and MXCSR register state from m512byte.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-43 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-46 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-47 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-43, Table 3-46, or Table 3-47. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

**Operation**

IF 64-Bit Mode

THEN

(x87 FPU, MMX, XMM15-XMM0, MXCSR) ← Load(SRC);

ELSE

(x87 FPU, MMX, XMM7-XMM0, MXCSR) ← Load(SRC);

FI;

**x87 FPU and SIMD Floating-Point Exceptions**

None.
Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
   If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See align-
   ment check exception [#AC] below.)
   For an attempt to set reserved bits in MXCSR.

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.
   If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
   If instruction is preceded by a LOCK prefix.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
   operand is not aligned on a 16-byte boundary, as described above. If the alignment check
   exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
   vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
   If any part of the operand lies outside the effective address space from 0 to FFFFH.
   For an attempt to set reserved bits in MXCSR.

#NM If CR0.TS[bit 3] = 1.
   If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
   If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code) For a page fault.

#AC For unaligned memory reference.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
If memory operand is not aligned on a 16-byte boundary, regardless of segment.
For an attempt to set reserved bits in MXCSR.

#PF(fault-code)  For a page fault.
#NM  If CR0.TS[bit 3] = 1.
If CR0.EM[bit 2] = 1.
#UD  If CPUID.01H:EDX.FXSR[bit 24] = 0.
If instruction is preceded by a LOCK prefix.

#AC  If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
**FXSAVE—Save x87 FPU, MMX Technology, and SSE State**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /0</td>
<td></td>
<td>M</td>
<td>Valid</td>
<td>Save the x87 FPU, MMX, XMM, and MXCSR register state to m512byte.</td>
</tr>
<tr>
<td>FXSAVE m512byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP REX.W + 0F AE /0</td>
<td></td>
<td>M</td>
<td>Valid</td>
<td>Save the x87 FPU, MMX, XMM, and MXCSR register state to m512byte.</td>
</tr>
<tr>
<td>FXSAVE64 m512byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512-byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit submode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area. The operation of FXSAVE in non-64-bit modes is described first.

**Non-64-Bit Mode Operation**

Table 3-43 shows the layout of the state information in memory when the processor is operating in legacy modes.

**Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rsvd</td>
<td>FCS</td>
<td>FIP[31:0]</td>
<td>FOP</td>
<td>Rsvd</td>
<td>FTW</td>
<td>FSW</td>
<td>FCW</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXCSR_MASK</td>
<td>MXCSR</td>
<td>Rsvd</td>
<td>FDS</td>
<td>FDP[31:0]</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST0/MM0</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST1/MM1</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST2/MM2</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST3/MM3</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST4/MM4</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST5/MM5</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST6/MM6</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST7/MM7</td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM0</td>
<td></td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM1</td>
<td></td>
<td>176</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM2</td>
<td></td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM3</td>
<td></td>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM4</td>
<td></td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM5</td>
<td></td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM6</td>
<td></td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM7</td>
<td></td>
<td>272</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-43 are defined in Table 3-44.

### Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region (Contd.)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The fields in Table 3-43 are defined in Table 3-44.

### Table 3-44. Field Definitions

<table>
<thead>
<tr>
<th>Field</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCW</td>
<td>x87 FPU Control Word (16 bits). See Figure 8-6 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the layout of the x87 FPU control word.</td>
</tr>
<tr>
<td>FSW</td>
<td>x87 FPU Status Word (16 bits). See Figure 8-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the layout of the x87 FPU status word.</td>
</tr>
<tr>
<td>Abridged FTW</td>
<td>x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.</td>
</tr>
<tr>
<td>FOP</td>
<td>x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the layout of the x87 FPU opcode field.</td>
</tr>
<tr>
<td>FIP</td>
<td>x87 FPU Instruction Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit IP offset. 16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit IP offset. 64-bit mode without REX.W — 32-bit IP offset. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of the x87 FPU instruction pointer.</td>
</tr>
</tbody>
</table>
The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field ( unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

\[
\begin{array}{cccccccc}
R7 & R6 & R5 & R4 & R3 & R2 & R1 & R0 \\
11 & xx & xx & 11 & 11 & 11 & 11 & 11 \\
\end{array}
\]

Here, 11B indicates empty stack elements and “xx” indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

\[
\begin{array}{cccccccc}
R7 & R6 & R5 & R4 & R3 & R2 & R1 & R0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be
used by an application program to pass a “clean” x87 FPU state to a procedure, since it retains the current
state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE
instruction to reinitialize the x87 FPU state.
• The format of the memory image saved with the FXSAVE instruction is the same regardless of the current
addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management).
This behavior differs from the FSAVE instructions, where the memory image format is different depending on
the addressing mode and operating mode. Because of the different image formats, the memory image saved
with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state
saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.
The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the
stored data was not the contents of MMX technology registers) using Table 3-45.

**Table 3-45. Recreating FSAVE Format**

<table>
<thead>
<tr>
<th>Exponent all 1’s</th>
<th>Exponent all 0’s</th>
<th>Fraction all 0’s</th>
<th>J and M bits</th>
<th>FTW valid bit</th>
<th>x87 FTW</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0x</td>
<td>1</td>
<td>Special 10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1x</td>
<td>1</td>
<td>Valid 00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Special 10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1x</td>
<td>1</td>
<td>Special 10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Special 10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>Zero 01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1x</td>
<td>1</td>
<td>Special 10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1x</td>
<td>1</td>
<td>Special 10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Special 10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>Special 10</td>
</tr>
</tbody>
</table>

For all legal combinations above.

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is
defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right
of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all
0’s.

**IA-32e Mode Operation**

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the
legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there
are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires
REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-46), the FPU IP and FPU DP pointers are 64-bit
wide. In the FXSAVE map for 64-bit mode (Table 3-47), the FPU IP and FPU DP pointers are 32-bits.
Table 3-46. Layout of the 64-bit-mode FXSAVE64 Map
(requires REX.W = 1)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIP</td>
<td>FOP</td>
<td>Reserved</td>
<td>FTW</td>
<td>FSW</td>
<td>FCW</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXCSR_MASK</td>
<td>MXCSR</td>
<td>FDP</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST0/MM0</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST1/MM1</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST2/MM2</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST3/MM3</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST4/MM4</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST5/MM5</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST6/MM6</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST7/MM7</td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM0</td>
<td></td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM1</td>
<td></td>
<td>176</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM2</td>
<td></td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM3</td>
<td></td>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM4</td>
<td></td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM5</td>
<td></td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM6</td>
<td></td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM7</td>
<td></td>
<td>272</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM8</td>
<td></td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM9</td>
<td></td>
<td>304</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM10</td>
<td></td>
<td>320</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM11</td>
<td></td>
<td>336</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM12</td>
<td></td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM13</td>
<td></td>
<td>368</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM14</td>
<td></td>
<td>384</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM15</td>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>416</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>432</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>448</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td>464</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td>480</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td>496</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 3-47. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>FCS</td>
<td>FIP[31:0]</td>
<td>FOP</td>
<td>Reserved</td>
<td>FTW</td>
<td>FSW</td>
<td>FCW</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXCSR_MASK</td>
<td>MXCSR</td>
<td>Reserved</td>
<td>FDS</td>
<td>FDP[31:0]</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST0/MM0</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST1/MM1</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST2/MM2</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST3/MM3</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST4/MM4</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST5/MM5</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST6/MM6</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST7/MM7</td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM0</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM1</td>
<td>176</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM2</td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM3</td>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM4</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM5</td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM6</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM7</td>
<td>272</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM8</td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM9</td>
<td>304</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM10</td>
<td>320</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM11</td>
<td>336</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM12</td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM13</td>
<td>368</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM14</td>
<td>384</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM15</td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>416</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>432</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>448</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>464</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>480</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>496</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operation

IF 64-Bit Mode
  THEN
    IF REX.W = 1
      THEN
        DEST ← Save64BitPromotedFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);
      ELSE
        DEST ← Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);
    FI;
  ELSE
    FI;
    DEST ← SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);

Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See the
description of the alignment check exception [#AC] below.)

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.
If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If CR0.TS[bit 3] = 1.
If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code) For a page fault.

#AC For unaligned memory reference.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.
If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.
If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Implementation Note
The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B. This order vary for FXSAVE for different processor implementations.
FXTRACT—Extract Exponent and Significand

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F4</td>
<td>Valid</td>
<td>Valid</td>
<td>Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.</td>
</tr>
</tbody>
</table>

**Description**

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand’s true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended \( \log_b(x) \) function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of \(-\infty\) is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
\begin{align*}
\text{TEMP} & \leftarrow \text{Significand(ST(0))}; \\
\text{ST(0)} & \leftarrow \text{Exponent(ST(0))}; \\
\text{TOP} & \leftarrow \text{TOP} - 1; \\
\text{ST(0)} & \leftarrow \text{TEMP};
\end{align*}
\]

**FPU Flags Affected**

- \( C1 \) Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
- \( C0, C2, C3 \) Undefined.

**Floating-Point Exceptions**

- #IS Stack underflow or overflow occurred.
- #IA Source operand is an SNaN value or unsupported format.
- #Z ST(0) operand is ±0.
- #D Source operand is a denormal value.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FYL2X—Compute $y \times \log_2 x$

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F1</td>
<td>FYL2X</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(1) with $(ST(1) \times \log_2 ST(0))$ and pop the register stack.</td>
</tr>
</tbody>
</table>

**Description**

Computes $(ST(1) \times \log_2 (ST(0)))$, stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-48. FYL2X Results**

<table>
<thead>
<tr>
<th>ST(1)</th>
<th>$-\infty$</th>
<th>$-F$</th>
<th>$\pm 0$</th>
<th>$+0&lt;+F&lt;+1$</th>
<th>$+1$</th>
<th>$+F&gt;+1$</th>
<th>$+\infty$</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>*</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>NaN</td>
</tr>
<tr>
<td>$-F$</td>
<td>*</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>NaN</td>
</tr>
<tr>
<td>$-0$</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>+F</td>
<td>$-0$</td>
<td>$-F$</td>
<td>$-\infty$</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>$-0$</td>
<td>$-0$</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>$-F$</td>
<td>+0</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>*</td>
<td>*</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
* Indicates floating-point invalid-operation (#IA) exception.
** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains $0$, the instruction returns $\mp$ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base $(b)$:

$$\log_b x \leftarrow (\log_2 b)^{-1} \times \log_2 x$$

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

ST(1) $\leftarrow$ ST(1) $\times$ log2ST(0);
PopRegisterStack;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.
Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Either operand is a SNaN or unsupported format.
   Source operand in register ST(0) is a negative finite value
   (not -0).

#Z Source operand in register ST(0) is ±0.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
FYL2XP1—Compute y * \log_2(x + 1)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F9</td>
<td>FYL2XP1</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(1) with ST(1) * \log_2(ST(0) + 1.0) and pop the register stack.</td>
</tr>
</tbody>
</table>

**Description**

Computes \((ST(1) \times \log_2(ST(0) + 1.0))\), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

\[-(1 - \sqrt{2}/2) \text{ to } (1 - \sqrt{2}/2)\]

The source operand in ST(1) can range from \(-\infty\) to \(+\infty\). If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>ST(1)</th>
<th>ST(0)</th>
<th>(-\infty)</th>
<th>(-0)</th>
<th>(+0)</th>
<th>(+0\text{ to } +(1 - \sqrt{2}/2))</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>--\infty</td>
<td>--\infty</td>
<td>*</td>
<td>*</td>
<td>--\infty</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>--F</td>
<td>+\infty</td>
<td>+0</td>
<td>-0</td>
<td>--F</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>--0</td>
<td>+0</td>
<td>+0</td>
<td>-0</td>
<td>--0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>--0</td>
<td>--0</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+F</td>
<td>--\infty</td>
<td>*</td>
<td>*</td>
<td>+\infty</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon \((\varepsilon)\) values, more significant digits can be retained by using the FYL2XP1 instruction than by using \((\varepsilon + 1)\) as an argument to the FYL2X instruction. The \((\varepsilon + 1)\) expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where \(n\) is the logarithm base desired for the result of the FYL2XP1 instruction:

\[
\text{scale factor} \leftarrow \log_n 2
\]

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
ST(1) \leftarrow ST(1) \times \log_2(ST(0) + 1.0);
\]

PopRegisterStack;
FPU Flags Affected
C1 Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.
#IA Either operand is an SNaN value or unsupported format.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**HADDPD—Packed Double-FP Horizontal Add**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 7C /r HADDPD xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Horizontal add packed double-precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 7C /r VHADDPD xmm1,xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal add packed double-precision floating-point values from xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 7C /r VHADDPD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal add packed double-precision floating-point values from ymm2 and ymm3/mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Adds the double-precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double-precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-16 for HADDPD; see Figure 3-17 for VHADDPD.

![Figure 3-16. HADDPD—Packed Double-FP Horizontal Add](OM15993)
HADDPD—Packed Double-FP Horizontal Add

Figure 3-17. VHADDPD operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zerroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HADDPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)

VHADDPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[MAXVL-1:128] ← 0

VHADDPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD: __m256d _mm256_hadd_pd (__m256d a, __m256d b);
HADDPD: __m128d _mm_hadd_pd (__m128d a, __m128d b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.
**Numeric Exceptions**
Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**
See Exceptions Type 2.
HADDPS—Packed Single-FP Horizontal Add

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 7C /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Horizontal add packed single-precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.F2.0F:W 7C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal add packed single-precision floating-point values from xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.F2.0F:W 7C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal add packed single-precision floating-point values from ymm2 and ymm3/mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Adds the single-precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single-precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).
See Figure 3-18 for HADDPS; see Figure 3-19 for VHADDPS.

**Figure 3-18. HADDPS—Packed Single-FP Horizontal Add**

**Figure 3-19. VHADDPS operation**

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
**Operation**

**HADDPS (128-bit Legacy SSE version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64] \\
\text{DEST}[95:64] & \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0] \\
\text{DEST}[127:96] & \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64] \\
\text{DEST}[MAXVL-1:128] & \text{(Unmodified)}
\end{align*}
\]

**VHADDPS (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64] \\
\text{DEST}[95:64] & \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0] \\
\text{DEST}[127:96] & \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64] \\
\text{DEST}[MAXVL-1:128] & \leftarrow 0
\end{align*}
\]

**VHADDPS (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[63:32] + \text{SRC1}[31:0] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[127:96] + \text{SRC1}[95:64] \\
\text{DEST}[95:64] & \leftarrow \text{SRC2}[63:32] + \text{SRC2}[31:0] \\
\text{DEST}[127:96] & \leftarrow \text{SRC2}[127:96] + \text{SRC2}[95:64] \\
\text{DEST}[159:128] & \leftarrow \text{SRC1}[191:160] + \text{SRC1}[159:128] \\
\text{DEST}[191:160] & \leftarrow \text{SRC1}[255:224] + \text{SRC1}[223:192] \\
\text{DEST}[223:192] & \leftarrow \text{SRC2}[191:160] + \text{SRC2}[159:128] \\
\text{DEST}[255:224] & \leftarrow \text{SRC2}[255:224] + \text{SRC2}[223:192]
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

HADDPS: __m128 _mm_hadd_ps (__m128 a, __m128 b);
VHADDPS: __m256 _mm256_hadd_ps (__m256 a, __m256 b);

**Exceptions**

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**Numeric Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2.
HLT—Halt

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4</td>
<td>HLT</td>
<td>Z0</td>
<td>Valid</td>
<td>Valid</td>
<td>Halt</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z0</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Enter Halt state;

### Flags Affected

None

### Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.
HSUBPD—Packed Double-FP Horizontal Subtract

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 7D /r HSUBPD xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Horizontal subtract packed double-precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 7D /r VHSUBPD xmm1,xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed double-precision floating-point values from xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 7D /r VHSUBPD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed double-precision floating-point values from ymm2 and ymm3/mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

Subtracts the double-precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double-precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-20 for HSUBPD; see Figure 3-21 for VHSUBPD.

![Figure 3-20. HSUBPD—Packed Double-FP Horizontal Subtract](om15995.png)
**Figure 3-21. VHSUBPD operation**

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Operation**

**HSUBPD (128-bit Legacy SSE version)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{SRC1}[63:0] - \text{SRC1}[127:64] \\
\text{DEST}[127:64] & \leftarrow \text{SRC2}[63:0] - \text{SRC2}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow \text{(Unmodified)}
\end{align*}
\]

**VHSUBPD (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{SRC1}[63:0] - \text{SRC1}[127:64] \\
\text{DEST}[127:64] & \leftarrow \text{SRC2}[63:0] - \text{SRC2}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0
\end{align*}
\]

**VHSUBPD (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{SRC1}[63:0] - \text{SRC1}[127:64] \\
\text{DEST}[127:64] & \leftarrow \text{SRC2}[63:0] - \text{SRC2}[127:64] \\
\text{DEST}[191:128] & \leftarrow \text{SRC1}[191:128] - \text{SRC1}[255:192] \\
\text{DEST}[255:192] & \leftarrow \text{SRC2}[191:128] - \text{SRC2}[255:192]
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

HSUBPD:  
\[
\text{__m128d} \text{ _mm_hsub_pd(__m128d a, __m128d b)}
\]

VHSUBPD:  
\[
\text{__m256d} \text{ _mm256_hsub_pd (__m256d a, __m256d b)}
\]

**Exceptions**

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**Numeric Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal
Other Exceptions

See Exceptions Type 2.
HSUBPS—Packed Single-FP Horizontal Subtract

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 7D /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Horizontal subtract packed single-precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>HSUBPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.F2.0F:WIG 7D /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed single-precision floating-point values from xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VHSUBPS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.F2.0F:WIG 7D /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed single-precision floating-point values from ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>VHSUBPS ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>RM</td>
</tr>
<tr>
<td>RVM</td>
</tr>
</tbody>
</table>

**Description**

Subtracts the single-precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single-precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-22 for HSUBPS; see Figure 3-23 for VHSUBPS.
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Figure 3-22. HSUBPS—Packed Single-FP Horizontal Subtract**

**Figure 3-23. VHSUBPS operation**
**Operation**

**HSUBPS (128-bit Legacy SSE version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96] \\
\text{DEST}[95:64] & \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32] \\
\text{DEST}[127:96] & \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0
\end{align*}
\]

**VHSUBPS (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96] \\
\text{DEST}[95:64] & \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32] \\
\text{DEST}[127:96] & \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0
\end{align*}
\]

**VHSUBPS (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32] \\
\text{DEST}[63:32] & \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96] \\
\text{DEST}[95:64] & \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32] \\
\text{DEST}[127:96] & \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96] \\
\text{DEST}[159:128] & \leftarrow \text{SRC1}[159:128] - \text{SRC1}[191:160] \\
\text{DEST}[191:160] & \leftarrow \text{SRC1}[223:192] - \text{SRC1}[255:224] \\
\text{DEST}[223:192] & \leftarrow \text{SRC2}[159:128] - \text{SRC2}[191:160] \\
\text{DEST}[255:224] & \leftarrow \text{SRC2}[223:192] - \text{SRC2}[255:224]
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

- **HSUBPS**: \_m128 \_mm_hsub_ps(\_m128 a, \_m128 b);
- **VHSUBPS**: \_m256 \_mm256_hsub_ps (\_m256 a, \_m256 b);

**Exceptions**

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**Numeric Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2.
IDIV—Signed Divide

**Description**
Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-50.

<table>
<thead>
<tr>
<th>Table 3-50. IDIV Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operand Size</strong></td>
</tr>
<tr>
<td>Word/byte</td>
</tr>
<tr>
<td>Doubleword/word</td>
</tr>
<tr>
<td>Quadword/doubleword</td>
</tr>
<tr>
<td>Doublequadword/quadword</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 cannot be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
Operation

IF SRC = 0
    THEN #DE; (* Divide error *)
FI;

IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp ← AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
            (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL ← temp;
                AH ← AX SignedModulus SRC;
            FI;
        ELSE IF OperandSize = 16 (* Doubleword/word operation *)
            THEN
                temp ← DX:AX / SRC; (* Signed division *)
                IF (temp > 7FFFFH) or (temp < 80000000H)
                    (* If a positive result is greater than 7FFFFH or a negative result is less than 80000000H *)
                    THEN #DE; (* Divide error *)
                    ELSE
                        AX ← temp;
                        DX ← DX:AX SignedModulus SRC;
                    FI;
            ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
                THEN
                    temp ← EDX:EAX / SRC; (* Signed division *)
                    IF (temp > 7FFFFFFFFH) or (temp < 8000000000000000H)
                        (* If a positive result is greater than 7FFFFFFFFH or a negative result is less than 8000000000000000H *)
                        THEN #DE; (* Divide error *)
                        ELSE
                            EAX ← temp;
                            EDX ← EDX:EAX SignedModulus SRC;
                        FI;
                ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
                    THEN temp ← RDX:RAX / SRC; (* Signed division *)
                        IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000000000000000000H)
                            (* If a positive result is greater than 7FFFFFFFFFFFFFFFH or a negative result is less than 8000000000000000000000000000000H *)
                            THEN #DE; (* Divide error *)
                            ELSE
                                RAX ← temp;
                                RDX ← RDX:RAX SignedModulus SRC;
                            FI;
                    FI;
            FI;
        FI;
    FI;
Flags Affected
The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions
#DE If the source operand (divisor) is 0.
The signed result (quotient) is too large for the destination.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#DE If the source operand (divisor) is 0.
The signed result (quotient) is too large for the destination.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#DE If the source operand (divisor) is 0.
The signed result (quotient) is too large for the destination.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#DE If the source operand (divisor) is 0
If the quotient is too large for the designated register.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
IMUL—Signed Multiply

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6/5</td>
<td>IMUL r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>AX ← AL * r/m byte.</td>
</tr>
<tr>
<td>F7/5</td>
<td>IMUL r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>DX:AX ← AX * r/m word.</td>
</tr>
<tr>
<td>F7/5</td>
<td>IMUL r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>EDX:EAX ← EAX * r/m32.</td>
</tr>
<tr>
<td>REX.W + F7/5</td>
<td>IMUL r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>RDX:RAX ← RAX * r/m64.</td>
</tr>
<tr>
<td>0F AF Ir</td>
<td>IMUL r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>word register ← word register * r/m16.</td>
</tr>
<tr>
<td>0F AF Ir</td>
<td>IMUL r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>doubleword register ← doubleword register * r/m32.</td>
</tr>
<tr>
<td>REX.W + 0F AF Ir</td>
<td>IMUL r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Quadword register ← Quadword register * r/m64.</td>
</tr>
<tr>
<td>6B Ir ib</td>
<td>IMUL r16, r/m16, imm8</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>word register ← r/m16 * sign-extended immediate byte.</td>
</tr>
<tr>
<td>6B Ir ib</td>
<td>IMUL r32, r/m32, imm8</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>doubleword register ← r/m32 * sign-extended immediate byte.</td>
</tr>
<tr>
<td>REX.W + 6B Ir ib</td>
<td>IMUL r64, r/m64, imm8</td>
<td>RMI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Quadword register ← r/m64 * sign-extended immediate byte.</td>
</tr>
<tr>
<td>69 Ir iw</td>
<td>IMUL r16, r/m16, imm16</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>word register ← r/m16 * immediate word.</td>
</tr>
<tr>
<td>69 Ir id</td>
<td>IMUL r32, r/m32, imm32</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>doubleword register ← r/m32 * immediate doubleword.</td>
</tr>
<tr>
<td>REX.W + 69 Ir id</td>
<td>IMUL r64, r/m64, imm32</td>
<td>RMI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Quadword register ← r/m64 * immediate doubleword.</td>
</tr>
</tbody>
</table>

Notes:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>imm8/16/32</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.

- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.

- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).
When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three-operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** —The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** —The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

**Operation**

IF (NumberOfOperands = 1)
 THEN IF (OperandSize = 8)
 THEN
   TMP_XP ← AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
   AX ← TMP_XP[15:0];
   IF SignExtend(TMP_XP[7:0]) = TMP_XP
      THEN CF ← 0; OF ← 0;
      ELSE CF ← 1; OF ← 1; Fi;
 ELSE IF OperandSize = 16
 THEN
   TMP_XP ← AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
   DX:AX ← TMP_XP[31:0];
   IF SignExtend(TMP_XP[15:0]) = TMP_XP
      THEN CF ← 0; OF ← 0;
      ELSE CF ← 1; OF ← 1; Fi;
 ELSE IF OperandSize = 32
 THEN
   TMP_XP ← EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
   EDX:EAX ← TMP_XP[63:0];
   IF SignExtend(TMP_XP[31:0]) = TMP_XP
      THEN CF ← 0; OF ← 0;
      ELSE CF ← 1; OF ← 1; Fi;
 ELSE (* OperandSize = 64 *)
   TMP_XP ← RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
   EDX:EAX ← TMP_XP[127:0];
   IF SignExtend(TMP_XP[63:0]) = TMP_XP
      THEN CF ← 0; OF ← 0;
      ELSE CF ← 1; OF ← 1; Fi;
 Fi;
Flags Affected
For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used.
**IN—Input from Port**

**Opcode** | **Instruction** | **Op/En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description**
---|---|---|---|---|---
E4 ib | IN AL, imm8 | I | Valid | Valid | Input byte from imm8 I/O port address into AL.
E5 ib | IN AX, imm8 | I | Valid | Valid | Input word from imm8 I/O port address into AX.
E5 ib | IN EAX, imm8 | I | Valid | Valid | Input dword from imm8 I/O port address into EAX.
EC | IN AL,DX | ZO | Valid | Valid | Input byte from I/O port in DX into AL.
ED | IN AX,DX | ZO | Valid | Valid | Input word from I/O port in DX into AX.
ED | IN EAX,DX | ZO | Valid | Valid | Input doubleword from I/O port in DX into EAX.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
---|---|---|---|---|
I | imm8 | NA | NA | NA
ZO | NA | NA | NA | NA

**Description**

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 18, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
\text{IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))}
\]
\[
\text{THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)}
\]
\[
\text{IF (Any I/O Permission Bit for I/O port being accessed = 1)}
\]
\[
\text{THEN (* I/O operation is not allowed *)}
\]
\[
\text{#GP(0)};
\]
\[
\text{ELSE (* I/O operation is allowed *)}
\]
\[
\text{DEST \leftarrow SRC; (* Read from selected I/O port *)}
\]
\[
\text{FI;}
\]
\[
\text{ELSE (Real Mode or Protected Mode with CPL \leq IOPL *)}
\]
\[
\text{DEST \leftarrow SRC; (* Read from selected I/O port *)}
\]
\[
\text{FI;}
\]

**Flags Affected**

None
**Protected Mode Exceptions**

- **#GP(0)**: If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
- **#UD**: If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#UD**: If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- **#GP(0)**: If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#UD**: If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#GP(0)**: If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
- **#UD**: If the LOCK prefix is used.
INC—Increment by 1

Description
Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC r16 and INC r32 are not encodable (because opcodes 40H through 47H are REX prefixes in 64-bit mode).

Operation
DEST ← DEST + 1;

AFlags Affected
The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions
#GP(0) If the destination operand is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>O</td>
<td>opcode + rd (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
** 40H through 47H are REX prefixes in 64-bit mode.
Real-Address Mode Exceptions

#GP       If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS       If a memory operand effective address is outside the SS segment limit.
#UD       If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)    If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0)    If alignment checking is enabled and an unaligned memory reference is made.
#UD       If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)    If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)    If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0)    If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD       If the LOCK prefix is used but the destination is not a memory operand.
INS/INSB/INSW/INSD—Input from Port to String

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6C</td>
<td>INS m8, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.*</td>
</tr>
<tr>
<td>6D</td>
<td>INS m16, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6D</td>
<td>INS m32, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6C</td>
<td>INSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6D</td>
<td>INSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6D</td>
<td>INSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be “DX,” and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the DI/EDI/RDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.
The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 18, "Input/Output," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

These instructions may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

**Operation**

IF ((PE = 1) and (((CPL > IOPL) or (VM = 1)))

THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)

IF (Any I/O Permission Bit for I/O port being accessed = 1)

THEN (* I/O operation is not allowed *)

#GP(0);

ELSE (* I/O operation is allowed *)

DEST ← SRC; (* Read from I/O port *)

FI;

ELSE (Real Mode or Protected Mode with CPL IOPL *)

DEST ← SRC; (* Read from I/O port *)

FI;

Non-64-bit Mode:

IF (Byte transfer)

THEN IF DF = 0

THEN (E)DI ← (E)DI + 1;

ELSE (E)DI ← (E)DI - 1; FI;

ELSE IF (Word transfer)

THEN IF DF = 0

THEN (E)DI ← (E)DI + 2;

ELSE (E)DI ← (E)DI - 2; FI;

ELSE (* Doubleword transfer *)

THEN IF DF = 0

THEN (E)DI ← (E)DI + 4;

ELSE (E)DI ← (E)DI - 4; FI;

FI;

FI64-bit Mode:

IF (Byte transfer)

THEN IF DF = 0

THEN (E|R)DI ← (E|R)DI + 1;

ELSE (E|R)DI ← (E|R)DI - 1; FI;

ELSE IF (Word transfer)

THEN IF DF = 0

THEN (E)DI ← (E)DI + 2;

ELSE (E)DI ← (E)DI - 2; FI;

ELSE (* Doubleword transfer *)

THEN IF DF = 0

THEN (E)DI ← (E)DI + 4;

ELSE (E)DI ← (E)DI - 4; FI;
THEN IF DF = 0
    THEN (E|R)DI ← (E|R)DI + 4;
    ELSE (E|R)DI ← (E|R)DI - 4; FI;
FI;

Flags Affected
None

Protected Mode Exceptions
#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
   If the destination is located in a non-writable segment.
   If an illegal memory operand effective address in the ES segments is given.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
   If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**INSERTPS**—Insert Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 21 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Insert a single-precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.WIG 21 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 21 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.</td>
</tr>
</tbody>
</table>

### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

**Register source form**

Copy a single-precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count_S bits are ignored.

**Memory source form**

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.
Operation

\textbf{VINSERTPS (VEX.128 and EVEX encoded version)}

\begin{verbatim}
IF (SRC = REG) THEN COUNT_S \leftarrow \text{imm8}[7:6]
  ELSE COUNT_S \leftarrow 0
COUNT_D \leftarrow \text{imm8}[5:4]
ZMASK \leftarrow \text{imm8}[3:0]
CASE (COUNT_S) OF
  0: TMP \leftarrow SRC2[31:0]
  1: TMP \leftarrow SRC2[63:32]
  2: TMP \leftarrow SRC2[95:64]
  3: TMP \leftarrow SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
  0: TMP2[31:0] \leftarrow TMP
  1: TMP2[63:32] \leftarrow SRC1[63:32]
  2: TMP2[95:64] \leftarrow SRC1[95:64]
  3: TMP2[127:96] \leftarrow SRC1[127:96]
ESAC;
IF (ZMASK[0] = 1) THEN DEST[31:0] \leftarrow 00000000H
  ELSE DEST[31:0] \leftarrow TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] \leftarrow 00000000H
  ELSE DEST[63:32] \leftarrow TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] \leftarrow 00000000H
  ELSE DEST[95:64] \leftarrow TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] \leftarrow 00000000H
  ELSE DEST[127:96] \leftarrow TMP2[127:96]
DEST[MAXVL-1:128] \leftarrow 0
\end{verbatim}

\textbf{INSERTPS (128-bit Legacy SSE version)}

\begin{verbatim}
IF (SRC = REG) THEN COUNT_S \leftarrow \text{imm8}[7:6]
  ELSE COUNT_S \leftarrow 0
COUNT_D \leftarrow \text{imm8}[5:4]
ZMASK \leftarrow \text{imm8}[3:0]
CASE (COUNT_S) OF
  0: TMP \leftarrow SRC[31:0]
  1: TMP \leftarrow SRC[63:32]
  2: TMP \leftarrow SRC[95:64]
  3: TMP \leftarrow SRC[127:96]
ESAC;
CASE (COUNT_D) OF
  0: TMP2[31:0] \leftarrow TMP
  1: TMP2[63:32] \leftarrow DEST[63:32]
  2: TMP2[95:64] \leftarrow DEST[95:64]
  3: TMP2[127:96] \leftarrow DEST[127:96]
ESAC;
\end{verbatim}

\textbf{INSERTPS—Insert Scalar Single-Precision Floating-Point Value}
INSERTPS—Insert Scalar Single-Precision Floating-Point Value

```
TMP2[63:0] ← DEST[63:0]
TMP2[127:96] ← DEST[127:96]
3: TMP2[127:96] ← TMP
   TMP2[95:0] ← DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
  ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
  ELSE DEST[95:64] ← TMP2[95:64]
  ELSE DEST[127:96] ← TMP2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSETPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int idx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int idx);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally
#UD If VEX.L = 0.
EVEX-encoded instruction, see Exceptions Type E9NF.
INT n/INTO/INT3/INT1—Call to Interrupt Procedure

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>INT3</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate breakpoint trap.</td>
</tr>
<tr>
<td>CD ib</td>
<td>INT imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate software interrupt with vector specified by immediate byte.</td>
</tr>
<tr>
<td>CE</td>
<td>INTO</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Generate overflow trap if overflow flag is 1.</td>
</tr>
<tr>
<td>F1</td>
<td>INT1</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate debug trap.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The INT n instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT n instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT n in the following ways:

• The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
• The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

The action of the INT n instruction (including the INTO, INT3, and INT1 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT n instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

Each of the INT n, INTO, and INT3 instructions generates a general-protection exception (#GP) if the CPL is greater than the DPL value in the selected gate descriptor in the IDT. In contrast, the INT1 instruction can deliver a #DB

---

1. The mnemonic ICEBP has also been used for the instruction with opcode F1.
even if the CPL is greater than the DPL of descriptor 1 in the IDT. (This behavior supports the use of INT1 by hard-
ware vendors performing hardware debug.)

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the
IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure.
In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate,
task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and
a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in
real-address mode, the IDT is called the interrupt vector table, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in
the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in
the "Operation" section for this instruction (except #GP).

<table>
<thead>
<tr>
<th>Table 3-51. Decision Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE</td>
</tr>
<tr>
<td>VM</td>
</tr>
<tr>
<td>IOPL</td>
</tr>
<tr>
<td>DPL/CPL RELATIONSHIP</td>
</tr>
<tr>
<td>INTERRUPT TYPE</td>
</tr>
<tr>
<td>GATE TYPE</td>
</tr>
<tr>
<td>REAL-ADDRESS-MODE</td>
</tr>
<tr>
<td>PROTECTED-MODE</td>
</tr>
<tr>
<td>TRAP-OR-_INTERRUPT-GATE</td>
</tr>
<tr>
<td>INTERRUPT-PRIVILEGE-LEVEL-INTERRUPT</td>
</tr>
<tr>
<td>INTERRUPT-PRIVILEGE-LEVEL-INTERRUPT</td>
</tr>
<tr>
<td>TASK- FROM- VIRTUAL-8086-MODE</td>
</tr>
<tr>
<td>#GP</td>
</tr>
</tbody>
</table>

NOTES:
− Don’t Care.
Y Yes, action taken.
Blank Action not taken.
S/W Applies to INT n, INTO, and INTO, but not to INT1.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If
the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes
a protected mode interrupt to privilege level 0. The interrupt gate’s DPL must be set to 3 and the target CPL of the
interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base
address value of the IDTR after the processor is powered up or reset is 0.

Operation
The following operational description applies not only to the INT n, INTO, INT3, or INT1 instructions, but also to
external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an
error code.
The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction error_code(num,idt,ext), where idt and ext are bit values. The pseudofunction produces an error code as follows: (1) if idt is 0, the error code is (num & FCH) | ext; (2) if idt is 1, the error code is (num « 3) | 2 | ext.

In many cases, the pseudofunction error_code is invoked with a pseudovariable EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt (INT n, INT3, or INTO), EXT is 0; otherwise (including INT1), EXT is 1.

IF PE = 0
	THEN
	GOTO REAL-ADDRESS-MODE;
ELSE (* PE = 1 *)
	IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT n)
	THEN
		#GP(0); (* Bit 0 of error code is 0 because INT n *)
	ELSE
		IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT n)
		THEN
			Consult bit n of the software interrupt redirection bit map in the TSS;
			IF bit n is clear
			THEN (* redirect interrupt to 8086 program interrupt handler *)
			Push EFLAGS[15:0]; (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
			Push CS;
			Push IP;
			IF IOPL = 3
			THEN IF ← 0; (* Clear interrupt flag *)
			ELSE VIF ← 0; (* Clear virtual interrupt flag *)
			FI;
			TF ← 0; (* Clear trap flag *)
			load CS and EIP (lower 16 bits only) from entry n in interrupt vector table referenced from TSS;
		ELSE
			IF IOPL = 3
			THEN GOTO PROTECTED-MODE;
			ELSE #GP(0); (* Bit 0 of error code is 0 because INT n *)
		FI;
	ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
	IF (IA32_EFER.LMA = 0)
	THEN (* Protected mode, or virtual-8086 mode interrupt *)
	GOTO PROTECTED-MODE;
	ELSE (* IA-32e mode interrupt *)
	GOTO IA-32e-MODE;
	FI;
	FI;
FI;
FI;
REAL-ADDRESS-MODE:
	IF ((vector_number « 2) + 3) is not within IDT limit
	THEN #GP; FI;
	IF stack not large enough for a 6-byte return information
	THEN #SS; FI;
	Push (EFLAGS[15:0]);
IF ← 0; (* Clear interrupt flag *)
TF ← 0; (* Clear trap flag *)
AC ← 0; (* Clear AC flag *)
Push(CS);
Push(IP);
(* No error codes are pushed in real-address mode*)
CS ← IDT(Descriptor (vector_number « 2), selector);
EIP ← IDT(Descriptor (vector_number « 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

PROTECTED-MODE:
IF ((vector_number « 3) + 7) is not within IDT limits
or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
THEN #GP(error_code(vector_number,1,EXT)); Fi;
(* idt operand to error_code set because vector is used *)
IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
THEN
IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
THEN #GP(error_code(vector_number,1,0)); Fi;
(* ext operand to error_code is 0 because INT n, INT3, or INTO*)
Fi;
IF gate not present
THEN #NP(error_code(vector_number,1,EXT)); Fi;
(* idt operand to error_code set because vector is used *)
IF task gate (* Specified in the selected interrupt table descriptor *)
THEN GOTO TASK-GATE;
ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
Fi;
END;
IA-32e-MODE:
IF INTO and CS.L = 1 (64-bit mode)
THEN #UD;
Fi;
IF ((vector_number « 4) + 15) is not in IDT limits
or selected IDT descriptor is not an interrupt-, or trap-gate type
THEN #GP(error_code(vector_number,1,EXT));
(* idt operand to error_code set because vector is used *)
Fi;
IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
THEN
IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
THEN #GP(error_code(vector_number,1,0));
(* ext operand to error_code is 0 because INT n, INT3, or INTO*)
Fi;
If;
IF gate not present
THEN #NP(error_code(vector_number,1,EXT));
(* idt operand to error_code set because vector is used *)
IF task gate (* Specified in the selected interrupt table descriptor *)
THEN GOTO TASK-GATE;
ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
Fi;
END;
IF local/global bit is set to local or index not within GDT limits
    THEN #GP(error_code(TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
Access TSS descriptor in GDT;
IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
    THEN #GP(error_code(TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF TSS not present
    THEN #NP(error_code(TSS selector,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
SWITCH-TASKS (with nesting) to TSS;
IF interrupt caused by fault with error code
    THEN
        IF stack limit does not allow push of error code
            THEN #SS(EXT); FI;
        Push(error code);
    FI;
    IF EIP not within code segment limit
        THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is non-conforming with DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                        DPL < CPL *)
                ELSE (* VM = 1 *)
                    IF new code-segment DPL ≠ 0
                        THEN #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                        GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                        (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
                FI;
                ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
            IF VM = 1
                THEN #GP(error_code(new code-segment selector,0,EXT));
                (* idt operand to error_code is 0 because selector is used *)
                IF new code segment is conforming or new code-segment DPL = CPL
                    THEN
GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
#GP(error_code(new code-segment selector,0,EXT));
(* idt operand to error_code is 0 because selector is used *)
FI;
FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
THEN
(* Identify stack-segment selector for new privilege level in current TSS *)
IF current TSS is 32-bit
THEN
TSSstackAddress ← (new code-segment DPL « 3) + 4;
IF (TSSstackAddress + 5) > current TSS limit
THEN HTS(error_code(current TSS selector,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
ELSE (* current TSS is 16-bit *)
TSSstackAddress ← (new code-segment DPL « 2) + 2
IF (TSSstackAddress + 3) > current TSS limit
THEN HTS(error_code(current TSS selector,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
FI;
IF NewSS is NULL
THEN HTS(EXT); FI;
IF NewSS index is not within its descriptor-table limits or NewSS RPL ≠ new code-segment DPL
THEN HTS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ new code-segment DPL or new stack-segment Type does not indicate writable data segment
THEN HTS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
IF NewSS is not present
THEN HSS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
ELSE (* IA-32e mode *)
IF IDT-gate IST = 0
THEN TSSstackAddress ← (new code-segment DPL « 3) + 4;
ELSE TSSstackAddress ← (IDT gate IST « 3) + 28;
FI;
IF (TSSstackAddress + 7) > current TSS limit
THEN HTS(error_code(current TSS selector,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
FI;
IF IDT gate is 32-bit
THEN
  IF new stack does not have room for 24 bytes (error code pushed)
  or 20 bytes (no error code pushed)
    THEN #SS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
  FI
ELSE
  IF IDT gate is 16-bit
    THEN
      IF new stack does not have room for 12 bytes (error code pushed)
      or 10 bytes (no error code pushed);
      THEN #SS(error_code(NewSS,0,EXT)); FI;
      (* idt operand to error_code is 0 because selector is used *)
    ELSE (* 64-bit IDT gate*)
      IF StackAddress is non-canonical
        THEN #SS(EXT); FI; (* Error code contains NULL selector *)
    FI;
  ELSE (* IA-32e mode *)
    THEN
      IF instruction pointer from IDT gate is not within new code-segment limits
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
      ESP ← NewESP;
      SS ← NewSS; (* Segment descriptor information also loaded *)
    ELSE (* IA-32e mode *)
      IF instruction pointer from IDT gate contains a non-canonical address
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
      RSP ← NewRSP & FFFFFFFFFFFFFFFFOH;
      SS ← NewSS;
    FI;
  IF IDT gate is 32-bit
    THEN
      CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
  ELSE
    IF IDT gate 16-bit
      THEN
        CS:IP ← Gate(CS:IP);
        (* Segment descriptor information also loaded *)
      ELSE (* 64-bit IDT gate *)
        CS:RIP ← Gate(CS:RIP);
        (* Segment descriptor information also loaded *)
    FI;
  FI;
IF IDT gate is 32-bit
  THEN
    Push(far pointer to old stack);
    (* Old SS and ESP, 3 words padded to 4 *)
    Push(EFLAGS);
    Push(far pointer to return instruction);
    (* Old CS and EIP, 3 words padded to 4 *)
    Push(ErrorCode); (* If needed, 4 bytes *)
  ELSE
    IF IDT gate 16-bit
      THEN
Push(far pointer to old stack);
(* Old SS and SP, 2 words *)
Push(EFLAGS(15:0));
Push(far pointer to return instruction);
(* Old CS and IP, 2 words *)
Push(ErrorCode); (* If needed, 2 bytes *)
ELSE (* 64-bit IDT gate *)
Push(far pointer to old stack);
(* Old SS and SP, each an 8-byte push *)
Push(RFLAGS); (* 8-byte push *)
Push(far pointer to return instruction);
(* Old CS and RIP, each an 8-byte push *)
Push(ErrorCode); (* If needed, 8-bytes *)
FI;
FI;
CPL ← new code-segment DPL;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
TF ← 0;
VM ← 0;
RF ← 0;
NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
(* Identify stack-segment selector for privilege level 0 in current TSS *)
IF current TSS is 32-bit
THEN
IF TSS limit < 9
THEN #TS(error_code(current TSS selector,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewSS ← 2 bytes loaded from (current TSS base + 8);
NewESP ← 4 bytes loaded from (current TSS base + 4);
ELSE (* current TSS is 16-bit *)
IF TSS limit < 5
THEN #TS(error_code(current TSS selector,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewSS ← 2 bytes loaded from (current TSS base + 4);
NewESP ← 2 bytes loaded from (current TSS base + 2);
FI;
IF NewSS is NULL
THEN #TS(EXT); FI; (* Error code contains NULL selector *)
IF NewSS index is not within its descriptor table limits
or NewSS RPL ≠ 0
THEN #TS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
THEN #TS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
THEN #SS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
IF IDT gate is 32-bit
THEN
  IF new stack does not have room for 40 bytes (error code pushed)
  or 36 bytes (no error code pushed)
  THEN #SS(error_code(NewSS,0,EXT)); Fl;
  (* idt operand to error_code is 0 because selector is used *)
ELSE (* IDT gate is 16-bit)
  IF new stack does not have room for 20 bytes (error code pushed)
  or 18 bytes (no error code pushed)
  THEN #SS(error_code(NewSS,0,EXT)); Fl;
  (* idt operand to error_code is 0 because selector is used *)
FI;
IF instruction pointer from IDT gate is not within new code-segment limits
  THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
IF service through interrupt gate
  THEN IF = 0; Fl;
TempSS ← SS;
TempESP ← ESP;
SS ← NewSS;
ESP ← NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;*
  Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
Push(TempEFlags);
Push(CS);
Push(EIP);
GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS ← 0;
DS ← 0;
ES ← 0;
CS ← Gate(CS); (* Segment descriptor information also loaded *)
CS(RPL) ← 0;
CPL ← 0;
IF IDT gate is 32-bit
  THEN
    EIP ← Gate(instruction pointer);  
  ELSE (* IDT gate is 16-bit *)
    EIP ← Gate(instruction pointer) AND 0000FFFFH;
FI;
(* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
  (* PE = 1, DPL = CPL or conforming segment *)
IF IA32_EFER.LMA = 1 (* IA-32e mode *)
  IF IDT-descriptor IST = 0
THEN
  TSSstackAddress ← (IDT-descriptor IST « 3) + 28;
  IF (TSSstackAddress + 7) > TSS limit
    THEN #TS(error_code(current TSS selector,0,EXT)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
    NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
  ELSE NewRSP ← RSP;
    Fl;
  FI;
IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
  THEN
    IF current stack does not have room for 16 bytes (error code pushed)
        or 12 bytes (no error code pushed)
      THEN #SS(EXT); Fl; (* Error code contains NULL selector *)
    ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
      IF current stack does not have room for 8 bytes (error code pushed)
          or 6 bytes (no error code pushed)
        THEN #SS(EXT); Fl; (* Error code contains NULL selector *)
    ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
      IF NewRSP contains a non-canonical address
        THEN #SS(EXT); (* Error code contains NULL selector *)
      FI;
  FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
  THEN
    IF instruction pointer from IDT gate is not within new code-segment limit
      THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
    ELSE
      IF instruction pointer from IDT gate contains a non-canonical address
        THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
      RSP ← NewRSP & FFFFFFFFFFFFFFOH;
    FI;
IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
  THEN
    Push (EFLAGS);
    Push (far pointer to return instruction); (* 3 words padded to 4 *)
    CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
    Push (ErrorCode); (* If any *)
  ELSE
    IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
      THEN
        Push (FLAGS);
        Push (far pointer to return location); (* 2 words *)
        CS:IP ← Gate(CS:IP); (* Segment descriptor information also loaded *)
        Push (ErrorCode); (* If any *)
      ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
        Push(far pointer to old stack);
        (* Old SS and SP, each an 8-byte push *)
        Push(RFLAGS); (* 8-byte push *)
        Push(far pointer to return instruction);
        (* Old CS and RIP, each an 8-byte push *)
        Push(ErrorCode); (* If needed, 8 bytes *)
        CS:RIP ← GATE(CS:RIP);
(* Segment descriptor information also loaded *)

Fl;
Fl;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
       THEN IF ← 0; Fl; (* Interrupt flag set to 0; interrupts disabled *)
TF ← 0;
NT ← 0;
VM ← 0;
RF ← 0;
END;

Flags Affected
The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on
the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the
interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task’s TSS.

Protected Mode Exceptions
#GP(error_code) If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code
    segment limits.
    If the segment selector in the interrupt, trap, or task gate is NULL.
    If an interrupt, trap, or task gate, code segment, or TSS segment selector index is outside its
descriptor table limits.
    If the vector selects a descriptor outside the IDT limits.
    If an IDT descriptor is not an interrupt, trap, or task gate.
    If an interrupt is generated by the INT n, INT3, or INTO instruction and the DPL of an interrupt,
    trap, or task gate is less than the CPL.
    If the segment selector in an interrupt or trap gate does not point to a segment descriptor for
    a code segment.
    If the segment selector for a TSS has its local/global bit set for local.
    If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(error_code) If pushing the return address, flags, or error code onto the stack exceeds the bounds of the
    stack segment and no stack switch occurs.
    If the SS register is being loaded and the segment pointed to is marked not present.
    If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds
    of the new stack segment when a stack switch occurs.

#NP(error_code) If code segment, interrupt gate, trap gate, task gate, or TSS is not present.

#TS(error_code) If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment
    being accessed by the interrupt or trap gate.
    If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is
    not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
    If the stack segment selector in the TSS is NULL.
    If the stack segment for the TSS is not a writable data segment.
    If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.
#AC(EXT) If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.
Real-Address Mode Exceptions

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the interrupt vector number is outside the IDT limits.

#SS  If stack limit violation on push.
If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.

#UD  If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)  (For INT n, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt, trap, or task gate is not equal to 3.
If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.
If the segment selector in the interrupt, trap, or task gate is NULL.
If a interrupt gate, trap gate, task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
If the vector selects a descriptor outside the IDT limits.
If an IDT descriptor is not an interrupt, trap, or task gate.
If an interrupt is generated by INT n, INT3, or INTO and the DPL of an interrupt, trap, or task gate is less than the CPL.
If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.
If the segment selector for a TSS has its local/global bit set for local.

#SS(error_code)  If the SS register is being loaded and the segment pointed to is marked not present.
If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.

#NP(error_code)  If code segment, interrupt gate, trap gate, task gate, or TSS is not present.

#TS(error_code)  If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
If DPL of the stack segment descriptor for the TSS’s stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
If the stack segment selector in the TSS is NULL.
If the stack segment for the TSS is not a writable data segment.
If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code)  If a page fault occurs.

#OF  If the INTO instruction is executed and the OF flag is set.

#UD  If the LOCK prefix is used.

#AC(EXT)  If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
**64-Bit Mode Exceptions**

#GP(error_code) If the instruction pointer in the 64-bit interrupt gate or trap gate is non-canonical.
- If the segment selector in the 64-bit interrupt or trap gate is NULL.
- If the vector selects a descriptor outside the IDT limits.
- If the vector points to a gate which is in non-canonical space.
- If the vector points to a descriptor which is not a 64-bit interrupt gate or a 64-bit trap gate.
- If the descriptor pointed to by the gate selector is outside the descriptor table limit.
- If the descriptor pointed to by the gate selector is in non-canonical space.
- If the descriptor pointed to by the gate selector is not a code segment.
- If the descriptor pointed to by the gate selector doesn’t have the L-bit set, or has both the L-bit and D-bit set.
- If the descriptor pointed to by the gate selector has DPL > CPL.

#SS(error_code) If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.
- If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).

#NP(error_code) If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.

#TS(error_code) If an attempt to load RSP from the TSS causes an access to non-canonical space.
- If the RSP from the TSS is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

#AC(EXT) If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.
INVD—Invalidate Internal Caches

Invalidates (flushes) the processor’s internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVD instruction may be used when the cache is used as temporary memory and the cache contents need to be invalidated rather than written back to memory. When the cache is used as temporary memory, no external device should be actively writing data to main memory.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Note that any data from an external device to main memory (for example, via a PCIWrite) can be temporarily stored in the caches; these data can be lost when an INVD instruction is executed. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, temporary memory, testing, or fault recovery where cache coherency with main memory is not a concern), software should instead use the WBINVD instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)

Flags Affected

None

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions
#GP(0) The INVD instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
INVLPGe—Invalid e TLB Entries

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01/7</td>
<td>INVLPG m</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Invalidate TLB entries for page containing m.</td>
</tr>
</tbody>
</table>

NOTES:
* See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.¹

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidates only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see "MOV—Move to/from Control Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B and Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

This instruction's operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility
The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation
Invalidate(RelevantTLBEntries);
Continue; (* Continue execution *)

Flags Affected
None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UDOperand is a register.
If the LOCK prefix is used.

¹. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A), the instruction invalidates all of them.
Real-Address Mode Exceptions

#UD Operand is a register.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The INVLPG instruction cannot be executed at the virtual-8086 mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UD Operand is a register.
If the LOCK prefix is used.
INVPCID—Invalidate Process-Context Identifier

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 82 /r INVPCID r32, m128</td>
<td>RM</td>
<td>NE/V</td>
<td>INVPCID</td>
<td>Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.</td>
</tr>
<tr>
<td>66 0F 38 82 /r INVPCID r64, m128</td>
<td>RM</td>
<td>V/NE</td>
<td>INVPCID</td>
<td>Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>RM</td>
</tr>
</tbody>
</table>

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 4.10, "Caching Translation Information," in Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.) Invalidation is based on the INVPCID type specified in the register operand and the INVPCID descriptor specified in the memory operand.

Outside 64-bit mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode the register operand has 64 bits.

There are four INVPCID types currently defined:

- Individual-address invalidation: If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—for the linear address and PCID specified in the INVPCID descriptor.¹ In some cases, the instruction may invalidate global translations or mappings for other linear addresses (or other PCIDs) as well.
- Single-context invalidation: If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other PCIDs as well.
- All-context invalidation, including global translations: If the INVPCID type is 2, the logical processor invalidates all mappings—including global translations—associated with any PCID.
- All-context invalidation: If the INVPCID type is 3, the logical processor invalidates all mappings—except global translations—associated with any PCID. In some case, the instruction may invalidate global translations as well.

The INVPCID descriptor comprises 128 bits and consists of a PCID and a linear address as shown in Figure 3-24. For INVPCID type 0, the processor uses the full 64 bits of the linear address even outside 64-bit mode; the linear address is not used for other INVPCID types.

![Figure 3-24. INVPCID Descriptor](image_url)

¹. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A), the instruction invalidates all of them.
If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. In this case, executions with INVPCID types 0 and 1 are allowed only if the PCID specified in the INVPCID descriptor is 000H; executions with INVPCID types 2 and 3 invalidate mappings only for PCID 000H. Note that CR4.PCIDE must be 0 outside 64-bit mode (see Chapter 4.10.1, “Process-Context Identifiers (PCIDs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

Operation

INVPCID_TYPE ← value of register operand;  // must be in the range of 0–3
INVPCID_DESC ← value of memory operand;
CASE INVPCID_TYPE OF
  0: // individual-address invalidation
      PCID ← INVPCID_DESC[11:0];
      L_ADDR ← INVPCID_DESC[127:64];
      Invalidate mappings for L_ADDR associated with PCID except global translations;
      BREAK;
  1: // single PCID invalidation
      PCID ← INVPCID_DESC[11:0];
      Invalidate all mappings associated with PCID except global translations;
      BREAK;
  2: // all PCID invalidation including global translations
      Invalidate all mappings for all PCIDs, including global translations;
      BREAK;
  3: // all PCID invalidation retaining global translations
      Invalidate all mappings for all PCIDs except global translations;
      BREAK;
ESAC;

Intel C/C++ Compiler Intrinsic Equivalent

INVPCID: void _invpcid(unsigned __int32 type, void * descriptor);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains an unusable segment.
If the source operand is located in an execute-only code segment.
If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
If bits 63:12 of INVPCID_DESC are not all zero.
If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.

#PF(fault-code) If a page fault occurs in accessing the memory operand.

#SS(0) If the memory operand effective address is outside the SS segment limit.
If the SS register contains an unusable segment.

#UD If if CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
If bits 63:12 of INVPCID_DESC are not all zero.
If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.

#UD If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The INVPCID instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
If bits 63:12 of INVPCID_DESC are not all zero.
If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.
If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.

#PF(fault-code) If a page fault occurs in accessing the memory operand.

#SS(0) If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
IRET/IRETD—Interrupt Return

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 7 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction performs a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, “Handling Multiple NMIs” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A), execution of the IRET instruction unblocks NMIs.

---

### Opcode Instruction Op/ En 64-Bit Mode Compat/ Leg Mode Description

| CF   | IRET   | ZO  | Valid | Valid | Interrupt return (16-bit operand size). |
| CF   | IRETD  | ZO  | Valid | Valid | Interrupt return (32-bit operand size). |
| REX.W + CF | IRETQ | ZO  | Valid | N.E.  | Interrupt return (64-bit operand size). |

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

IF PE = 0
    THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
    FI;

REAL-ADDRESS-MODE:
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        tempEFLAGS ← Pop();
        EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
    ELSE (* OperandSize = 16 *)
        EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
        CS ← Pop(); (* 16-bit pop *)
        EFLAGS[15:0] ← Pop();
    FI;
END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
    THEN IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            EFLAGS ← Pop();
            (* VM, IOPL, VIP and VIF EFLAG bits not modified by pop *)
            IF EIP not within CS limit
                THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
        FI;
    ELSE
        ELSE
#GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)

FI;
END;

PROTECTED-MODE:
IF NT = 1
THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
FI;
IF OperandSize = 32
THEN
  EIP ← Pop();
  CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
  tempEFLAGS ← Pop();
ELSE (* OperandSize = 16 *)
  EIP ← Pop(); (* 16-bit pop; clear upper bits *)
  CS ← Pop(); (* 16-bit pop *)
  tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
FI;
IF tempEFLAGS(VM) = 1 and CPL = 0
THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
ELSE GOTO PROTECTED-MODE-RETURN;
FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
Mark the task just abandoned as NOT BUSY;
IF EIP is not within CS limit
THEN #GP(0); FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
(* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
IF EIP not within CS limit
THEN #GP(0); FI;
EFLAGS ← tempEFLAGS;
ESP ← Pop();
SS ← Pop(); (* Pop 2 words; throw away high-order word *)
ES ← Pop(); (* Pop 2 words; throw away high-order word *)
DS ← Pop(); (* Pop 2 words; throw away high-order word *)
FS ← Pop(); (* Pop 2 words; throw away high-order word *)
GS ← Pop(); (* Pop 2 words; throw away high-order word *)
CPL ← 3;
(* Resume execution in Virtual-8086 mode *)
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
IF CS(RPL) > CPL
THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF OperandSize = 32
THEN
ESP ← Pop();
SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
ELSE IF OperandSize = 16
THEN
   ESP ← Pop(); (* 16-bit pop; clear upper bits *)
   SS ← Pop(); (* 16-bit pop *)
ELSE (* OperandSize = 64 *)
   RSP ← Pop();
   SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
FI;
IF new mode ≠ 64-Bit Mode
THEN
   IF EIP is not within CS limit
      THEN #GP(0); FI;
   ELSE (* new mode = 64-bit mode *)
      IF RIP is non-canonical
         THEN #GP(0); FI;
   FI;
FI;
EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
IF OperandSize = 32 or OperandSize = 64
THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
IF CPL ≤ IOPL
THEN EFLAGS(IF) ← tempEFLAGS; FI;
IF CPL = 0
THEN
   EFLAGS(IOPL) ← tempEFLAGS;
   IF OperandSize = 32 or OperandSize = 64
      THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
FI;
CPL ← CS(RPL);
FOR each SegReg in (ES, FS, GS, and DS)
   DO
      tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
      IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
         THEN (* Segment register invalid *)
            SegmentSelector ← 0; (*Segment selector becomes null*)
      FI;
   OD;
END;
RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
IF new mode ≠ 64-Bit Mode
THEN
   IF EIP is not within CS limit
      THEN #GP(0); FI;
   ELSE (* new mode = 64-bit mode *)
      IF RIP is non-canonical
         THEN #GP(0); FI;
   FI;
FI;
EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
IF OperandSize = 32 or OperandSize = 64
THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
IF CPL ≤ IOPL
THEN EFLAGS(IF) ← tempEFLAGS; FI;
IF CPL = 0
THEN
  EFLAGS(IPL) ← tempEFLAGS;
  IF OperandSize = 32 or OperandSize = 64
  THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
FI;
END;

IA-32e-MODE:
IF NT = 1
THEN #GP(0);
ELSE IF OperandSize = 32
THEN
  EIP ← Pop();
  CS ← Pop();
  tempEFLAGS ← Pop();
ELSE IF OperandSize = 16
THEN
  EIP ← Pop(); (* 16-bit pop; clear upper bits *)
  CS ← Pop(); (* 16-bit pop *)
  tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
FI;
ELSE (* OperandSize = 64 *)
THEN
  RIP ← Pop();
  CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
  tempRFLAGS ← Pop();
FI;
IF CS.RPL > CPL
THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
ELSE
  IF instruction began in 64-Bit Mode
  THEN
    IF OperandSize = 32
    THEN
      ESP ← Pop();
      SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE IF OperandSize = 16
    THEN
      ESP ← Pop(); (* 16-bit pop; clear upper bits *)
      SS ← Pop(); (* 16-bit pop *)
    ELSE (* OperandSize = 64 *)
    THEN
      RSP ← Pop();
      SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
    FI;
  FI;
GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

**Flags Affected**

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task’s TSS.
Protected Mode Exceptions

#GP(0) If the return code or stack segment selector is NULL.
   If the return instruction pointer is not within the return code segment limit.
#GP(selector) If a segment selector index is outside its descriptor table limits.
   If the return code segment selector RPL is less than the CPL.
   If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
   If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
   If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
   If the stack segment is not a writable data segment.
   If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
   If the segment descriptor for a code segment does not indicate it is a code segment.
   If the segment selector for a TSS has its local/global bit set for local.
   If a TSS segment descriptor specifies that the TSS is not busy.
   If a TSS segment descriptor specifies that the TSS is not available.
#SS(0) If the top bytes of stack are not within stack limits.
   If the return stack segment is not present.
#NP (selector) If the return code segment is not present.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If the return instruction pointer is not within the return code segment limit.
#SS If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0) If the return instruction pointer is not within the return code segment limit.
   If IOPL not equal to 3.
#PF(fault-code) If a page fault occurs.
#SS(0) If the top bytes of stack are not within stack limits.
#AC(0) If an unaligned memory reference occurs and alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.
Other exceptions same as in Protected Mode.
64-Bit Mode Exceptions

#GP(0)  If EFLAGS.NT[bit 14] = 1.
If the return code segment selector is NULL.
If the stack segment selector is NULL going back to compatibility mode.
If the stack segment selector is NULL going back to CPL3 64-bit mode.
If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
If the return instruction pointer is not within the return code segment limit.
If the return instruction pointer is non-canonical.

#GP(Selector)  If a segment selector index is outside its descriptor table limits.
If a segment descriptor memory address is non-canonical.
If the segment descriptor for a code segment does not indicate it is a code segment.
If the proposed new code segment descriptor has both the D-bit and L-bit set.
If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
If CPL is greater than the RPL of the code segment selector.
If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
If the stack segment is not a writable data segment.
If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

#SS(0)  If an attempt to pop a value off the stack violates the SS limit.
If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
If the return stack segment is not present.

#NP (selector)  If the return code segment is not present.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD  If the LOCK prefix is used.
## Jcc—Jump if Condition Is Met

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>77 cb</td>
<td>JA rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>73 cb</td>
<td>JAE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if above or equal (CF=0).</td>
</tr>
<tr>
<td>72 cb</td>
<td>JB rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if below (CF=1).</td>
</tr>
<tr>
<td>76 cb</td>
<td>JBE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>72 cb</td>
<td>JC rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if carry (CF=1).</td>
</tr>
<tr>
<td>E3 cb</td>
<td>JECXZ relB</td>
<td>D</td>
<td>N.E.</td>
<td>Valid</td>
<td>Jump short if CX register is 0.</td>
</tr>
<tr>
<td>E3 cb</td>
<td>JRCXZ relB</td>
<td>D</td>
<td>Valid</td>
<td>N.E.</td>
<td>Jump short if RCX register is 0.</td>
</tr>
<tr>
<td>74 cb</td>
<td>JE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if equal (ZF=1).</td>
</tr>
<tr>
<td>7F cb</td>
<td>JG rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>7D cb</td>
<td>JGE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>7C cb</td>
<td>JL rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if less (SF≠OF).</td>
</tr>
<tr>
<td>7E cb</td>
<td>JLE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if less or equal (ZF=1 or SF≠OF).</td>
</tr>
<tr>
<td>76 cb</td>
<td>JNA rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>72 cb</td>
<td>JNAE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not above or equal (CF=1).</td>
</tr>
<tr>
<td>73 cb</td>
<td>JNB rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not below (CF=0).</td>
</tr>
<tr>
<td>77 cb</td>
<td>JNBE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>73 cb</td>
<td>JNC rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not carry (CF=0).</td>
</tr>
<tr>
<td>75 cb</td>
<td>JNE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not equal (ZF=0).</td>
</tr>
<tr>
<td>7E cb</td>
<td>JNG rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not greater (ZF=1 or SF≠OF).</td>
</tr>
<tr>
<td>7C cb</td>
<td>JNGE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not greater or equal (SF≠OF).</td>
</tr>
<tr>
<td>7D cb</td>
<td>JNL rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not less (SF=OF).</td>
</tr>
<tr>
<td>7F cb</td>
<td>JNLE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>71 cb</td>
<td>JNQ rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not overflow (OF=0).</td>
</tr>
<tr>
<td>7B cb</td>
<td>JNP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not parity (PF=0).</td>
</tr>
<tr>
<td>79 cb</td>
<td>JNS rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not sign (SF=0).</td>
</tr>
<tr>
<td>75 cb</td>
<td>JNZ rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not zero (ZF=0).</td>
</tr>
<tr>
<td>70 cb</td>
<td>JO rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if overflow (OF=1).</td>
</tr>
<tr>
<td>7A cb</td>
<td>JP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if parity (PF=1).</td>
</tr>
<tr>
<td>7A cb</td>
<td>JPE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if parity even (PF=1).</td>
</tr>
<tr>
<td>7B cb</td>
<td>JPO rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if parity odd (PF=0).</td>
</tr>
<tr>
<td>78 cb</td>
<td>JS rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if sign (SF=1).</td>
</tr>
<tr>
<td>74 cb</td>
<td>JZ rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if zero (ZF = 1).</td>
</tr>
<tr>
<td>0F 87 cw</td>
<td>JA rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 87 cd</td>
<td>JA rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 83 cw</td>
<td>JAE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if above or equal (CF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Instruction</td>
<td>Op/En</td>
<td>64-Bit Mode</td>
<td>Comp/Le Mode</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-------</td>
<td>-------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>0F 83 cd</td>
<td>JAE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if above or equal (CF=0).</td>
</tr>
<tr>
<td>0F 82 cw</td>
<td>JB rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if below (CF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 82 cd</td>
<td>JB rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if below (CF=1).</td>
</tr>
<tr>
<td>0F 86 cw</td>
<td>JBE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 86 cd</td>
<td>JBE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 82 cw</td>
<td>JC rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if carry (CF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 82 cd</td>
<td>JC rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if carry (CF=1).</td>
</tr>
<tr>
<td>0F 84 cw</td>
<td>JE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if equal (ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 84 cd</td>
<td>JE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if equal (ZF=1).</td>
</tr>
<tr>
<td>0F 86 cw</td>
<td>JG rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 86 cd</td>
<td>JG rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 8D cw</td>
<td>JGE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8D cd</td>
<td>JGE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 8C cw</td>
<td>JL rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if less (SF≠ OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8C cd</td>
<td>JL rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if less (SF≠ OF).</td>
</tr>
<tr>
<td>0F 8E cw</td>
<td>JLE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8E cd</td>
<td>JLE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if less or equal (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>0F 86 cw</td>
<td>JNA rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 86 cd</td>
<td>JNA rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 8C cw</td>
<td>JNC rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not above or equal (CF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8C cd</td>
<td>JNC rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 83 cw</td>
<td>JNB rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not below (CF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 83 cd</td>
<td>JNB rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not below (CF=0).</td>
</tr>
<tr>
<td>0F 87 cw</td>
<td>JNBE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 87 cd</td>
<td>JNBE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 83 cw</td>
<td>JNE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not carry (CF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 83 cd</td>
<td>JNE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not carry (CF=0).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Instruction</td>
<td>Op/En</td>
<td>64-Bit Mode</td>
<td>Comp/Leg Mode</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>0F 85 cw</td>
<td>JNE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not equal (ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 85 cd</td>
<td>JNE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 8E cw</td>
<td>JNG rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not greater (ZF=1 or SF≠ OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8E cd</td>
<td>JNG rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not greater (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>0F 8C cw</td>
<td>JNGE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not greater or equal (SF≠ OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8C cd</td>
<td>JNGE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not greater or equal (SF≠ OF).</td>
</tr>
<tr>
<td>0F 8D cw</td>
<td>JNL rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not less (SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8D cd</td>
<td>JNL rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 8F cw</td>
<td>JNLE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8F cd</td>
<td>JNLE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 81 cw</td>
<td>JNO rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not overflow (OF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 81 cd</td>
<td>JNO rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not overflow (OF=0).</td>
</tr>
<tr>
<td>0F 8B cw</td>
<td>JNP rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not parity (PF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8B cd</td>
<td>JNP rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not parity (PF=0).</td>
</tr>
<tr>
<td>0F 89 cw</td>
<td>JNS rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not sign (SF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 89 cd</td>
<td>JNS rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not sign (SF=0).</td>
</tr>
<tr>
<td>0F 85 cw</td>
<td>JNZ rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not zero (ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 85 cd</td>
<td>JNZ rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not zero (ZF=0).</td>
</tr>
<tr>
<td>0F 80 cw</td>
<td>JO rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if overflow (OF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 80 cd</td>
<td>JO rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if overflow (OF=1).</td>
</tr>
<tr>
<td>0F 8A cw</td>
<td>JP rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if parity (PF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8A cd</td>
<td>JP rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if parity (PF=1).</td>
</tr>
<tr>
<td>0F 8A cw</td>
<td>JPE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if parity even (PF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8A cd</td>
<td>JPE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if parity even (PF=1).</td>
</tr>
<tr>
<td>0F 8B cw</td>
<td>JPO rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if parity odd (PF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8B cd</td>
<td>JPO rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if parity odd (PF=0).</td>
</tr>
<tr>
<td>0F 88 cw</td>
<td>JS rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if sign (SF=1). Not supported in 64-bit mode.</td>
</tr>
</tbody>
</table>
**Description**

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the Jcc instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of –128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each Jcc mnemonic are given in the “Description” column of the table on the preceding page. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The Jcc instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the Jcc instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```plaintext
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```plaintext
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JRCXZ, JECXZ and JCXZ instructions differ from other Jcc instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as LOOPNE). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute $2^{64}$, $2^{32}$ or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

In 64-bit mode, operand size is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32-bit offset sign extended to 64 bits.

---

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Offset</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 88 cd</td>
<td>JS rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if sign (SF=1).</td>
</tr>
<tr>
<td>0F 84 cw</td>
<td>JZ rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 84 cd</td>
<td>JZ rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1).</td>
</tr>
</tbody>
</table>

---

**Jcc—Jump if Condition Is Met**
Operation

IF condition
 THEN
   tempEIP ← EIP + SignExtend(DEST);
   IF OperandSize = 16
      THEN tempEIP ← tempEIP AND 0000FFFFH;
   FI;
   IF tempEIP is not within code segment limit
      THEN #GP(0);
   ELSE EIP ← tempEIP
   FI;
FI;

Flags Affected
None

Protected Mode Exceptions
#GP(0) If the offset being jumped to is beyond the limits of the CS segment.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
#UD If the LOCK prefix is used.
## JMP—Jump

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EB cb</td>
<td>JMP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits</td>
</tr>
<tr>
<td>E9 cw</td>
<td>JMP rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>E9 cd</td>
<td>JMP rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits</td>
</tr>
<tr>
<td>FF /4</td>
<td>JMP r/m16</td>
<td>M</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near, absolute indirect, address = zero-extended r/m16. Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>FF /4</td>
<td>JMP r/m32</td>
<td>M</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near, absolute indirect, address given in r/m32. Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>FF /4</td>
<td>JMP r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Jump near, absolute indirect, RIP = 64-Bit offset from register or memory</td>
</tr>
<tr>
<td>EA cd</td>
<td>JMP ptr16:16</td>
<td>D</td>
<td>Inv.</td>
<td>Valid</td>
<td>Jump far, absolute, address given in operand</td>
</tr>
<tr>
<td>EA cp</td>
<td>JMP ptr16:32</td>
<td>D</td>
<td>Inv.</td>
<td>Valid</td>
<td>Jump far, absolute, address given in operand</td>
</tr>
<tr>
<td>FF /5</td>
<td>JMP m16:16</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump far, absolute indirect, address given in m16:16</td>
</tr>
<tr>
<td>FF /5</td>
<td>JMP m16:32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump far, absolute indirect, address given in m16:32</td>
</tr>
<tr>
<td>REX,W + FF /5</td>
<td>JMP m16:64</td>
<td>D</td>
<td>Valid</td>
<td>N.E.</td>
<td>Jump far, absolute indirect, address given in m16:64</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Offset</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- **Near jump**—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- **Short jump**—A near jump where the jump range is limited to \(-128\) to \(+127\) from the current EIP value.
- **Far jump**—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- **Task switch**—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, Volume 3A, for information on performing task switches with the JMP instruction).

### Near and Short Jumps

When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current
value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (rel8) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16 or r/m32). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task’s code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7 in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for detailed information on the mechanics of a task switch.
Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS’s previous task link field is not loaded with the old task’s TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

**In 64-Bit Mode** — The instruction’s operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF near jump
IF 64-bit Mode
THEN
IF near relative jump
THEN
tempRIP ← RIP + DEST; (* RIP is instruction following JMP instruction*)
ELSE (* Near absolute jump *)
tempRIP ← DEST;
FI;
ELSE
IF near relative jump
THEN
tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
ELSE (* Near absolute jump *)
tempEIP ← DEST;
FI;
FI;
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
and tempEIP outside code segment limit
THEN #GP(0); FI
IF 64-bit mode and tempRIP is not canonical
THEN #GP(0);
FI;
IF OperandSize = 32
THEN
EIP ← tempEIP;
ELSE
IF OperandSize = 16
THEN (* OperandSize = 16 *)
EIP ← tempEIP AND 0000FFFFH;
ELSE (* OperandSize = 64)
RIP ← tempRIP;
FI;
FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
THEN
tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
IF tempEIP is beyond code segment limit
THEN #GP(0); FI;
CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
IF OperandSize = 32
THEN
  EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
ELSE (* OperandSize = 16 *)
  EIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
FI;
FI;
IF far jump and (PE = 1 and VM = 0)
  (* IA-32e mode or protected mode, not virtual-8086 mode *)
  THEN
    IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
    or segment selector in target operand NULL
      THEN #GP(0); FI;
    IF segment selector index not within descriptor table limits
      THEN #GP(new selector); FI;
    Read type and access rights of segment descriptor;
    IF (EFERLMA = 0)
      THEN
        IF segment type is not a conforming or nonconforming code
          segment, call gate, task gate, or TSS
          THEN #GP(segment selector); FI;
        ELSE
          IF segment type is not a conforming or nonconforming code segment
            call gate
            THEN #GP(segment selector); FI;
          FI;
      FI;
    Depending on type and access rights:
      GO TO CONFORMING-CODE-SEGMENT;
      GO TO NONCONFORMING-CODE-SEGMENT;
      GO TO CALL-GATE;
      GO TO TASK-GATE;
      GO TO TASK-STATE-SEGMENT;
  ELSE
    #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
  IF L-Bit = 1 and D-BIT = 1 and IA32_EFERLMA = 1
    THEN GP(new code segment selector); FI;
  IF DPL > CPL
    THEN #GP(segment selector); FI;
  IF segment not present
    THEN #NP(segment selector); FI;
  tempEIP ← DEST(Offset);
  IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH;
  FI;
  IF (IA32_EFERLMA = 0 or target mode = Compatibility mode) and
  tempEIP outside code segment limit
    THEN #GP(0); FI
  IF tempEIP is non-canonical
    THEN #GP(0); FI;
  CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
  CS(RPL) ← CPL
  EIP ← tempEIP;
END;
NONCONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); Fl;
IF (RPL > CPL) OR (DPL ≠ CPL)
    THEN #GP(code segment selector); Fl;
IF segment not present
    THEN #NP(segment selector); Fl;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
    and tempEIP outside code segment limit
    THEN #GP(0); Fl
IF tempEIP is non-canonical THEN #GP(0); Fl;
CS ← DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
END;

CALL-GATE:
IF call gate DPL < CPL
    or call gate DPL < call gate segment-selector RPL
    THEN #GP(call gate selector); Fl;
IF call gate not present
    THEN #NP(call gate selector); Fl;
IF call gate code-segment selector is NULL
    THEN #GP(0); Fl;
IF call gate code-segment selector index outside descriptor table limits
    THEN #GP(code segment selector); Fl;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor is conforming and DPL > CPL
    or code-segment segment descriptor is non-conforming and DPL ≠ CPL
    THEN #GP(code segment selector); Fl;
IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
    or code-segment segment descriptor has both L-Bit and D-bit set)
    THEN #GP(code segment selector); Fl;
IF code segment is not present
    THEN #NP(code segment selector); Fl;
tempEIP ← DEST(Offset);
IF GateSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
    outside code segment limit
    THEN #GP(0); Fl
CS ← DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
END;

TASK-GATE:
IF task gate DPL < CPL
    or task gate DPL < task gate segment-selector RPL
    THEN #GP(task gate selector); Fl;
IF task gate not present
THEN #NP(gate selector); Fl;
Read the TSS segment selector in the task-gate descriptor;
IF TSS segment selector local/global bit is set to local
or index not within GDT limits
or descriptor is not a TSS segment
or TSS descriptor specifies that the TSS is busy
THEN #GP(TSS selector); Fl;
IF TSS not present
THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
THEN #GP(0); Fl;
END;

TASK-STATE-SEGMENT:
IF TSS DPL < CPL
or TSS DPL < TSS segment-selector RPL
or TSS descriptor indicates TSS not available
THEN #GP(TSS selector); Fl;
IF TSS is not present
THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
THEN #GP(0); Fl;
END;

Flags Affected
All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions
#GP(0)  If offset in target operand, call gate, or TSS is beyond the code segment limits.
If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#GP(selector)  If the segment selector index is outside descriptor table limits.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
If the DPL for a nonconforming-code segment is not equal to the CPL
(When not using a call gate.) If the RPL for the segment’s segment selector is greater than the CPL.
If the DPL for a conforming-code segment is greater than the CPL.
If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS’s segment selector.
If the segment descriptor for selector in a call gate does not indicate it is a code segment.
If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.
If the segment selector for a TSS has its local/global bit set for local.
If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0)  If a memory operand effective address is outside the SS segment limit.
#NP (selector) If the code segment being accessed is not present.
If call gate, task gate, or TSS not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If the target operand is beyond the code segment limits.
If target offset in destination operand is non-canonical.
If target offset in destination operand is beyond the new code segment limit.
If the segment selector in the destination operand is NULL.
If the code segment selector in the 64-bit gate is NULL.

#GP(selector) If the code segment or 64-bit call gate is outside descriptor table limits.
If the code segment or 64-bit call gate overlaps non-canonical space.
If the segment descriptor from a 64-bit call gate is in non-canonical space.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.
If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set.
If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment’s segment selector is greater than the CPL.
If the DPL for a conforming-code segment is greater than the CPL.
If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
If the upper type field of a 64-bit call gate is not 0x0.
If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
If the code segment descriptor pointed to by the selector in the 64-bit gate doesn’t have the L-bit set and the D-bit clear.
If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
If the code segment is non-conforming and CPL \(\neq\) DPL.
If the code segment is confirming and CPL < DPL.

#NP(selector)  If a code segment or 64-bit call gate is not present.

#UD  (64-bit mode only) If a far jump is direct to an absolute address in memory.
     If the LOCK prefix is used.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
KADDW/KADDB/KADDQ/KADDD—ADD Two Masks

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L1.0F.W0 4A /r KADDW k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Add 16 bits masks in k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Add 8 bits masks in k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add 64 bits masks in k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 4A /r KADDD k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add 32 bits masks in k2 and k3 and place result in k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

**KADDW**

\[
\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] + \text{SRC2}[15:0]
\]

\[
\text{DEST}[\text{MAX}_K\text{L}-1:16] \leftarrow 0
\]

**KADDB**

\[
\text{DEST}[7:0] \leftarrow \text{SRC1}[7:0] + \text{SRC2}[7:0]
\]

\[
\text{DEST}[\text{MAX}_K\text{L}-1:8] \leftarrow 0
\]

**KADDQ**

\[
\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]
\]

\[
\text{DEST}[\text{MAX}_K\text{L}-1:64] \leftarrow 0
\]

**KADDD**

\[
\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]
\]

\[
\text{DEST}[\text{MAX}_K\text{L}-1:32] \leftarrow 0
\]

Intel C/C++ Compiler Intrinsic Equivalent

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.
KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.L1.0F.W0 41 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND 16 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDW k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 41 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise AND 8 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDB k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.0F.W1 41 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND 64 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDQ k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 41 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND 32 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDD k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM: [7:6] must be 11b)</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

**Operation**

**KANDW**

DEST[15:0] ← SRC1[15:0] BITWISE AND SRC2[15:0]  
DEST[MAX_KL-1:16] ← 0

**KANDB**

DEST[7:0] ← SRC1[7:0] BITWISE AND SRC2[7:0]  
DEST[MAX_KL-1:8] ← 0

**KANDQ**

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]  
DEST[MAX_KL-1:64] ← 0

**KANDD**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]  
DEST[MAX_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KANDW __mmask16 _mm512_kand(__mmask16 a, __mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.
## KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.L1.0F:w0 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDNW k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F:w0 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.</td>
</tr>
<tr>
<td>KANDNB k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.0F:w1 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDNQ k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F:w1 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDND k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM:[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

### Description
Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

### Operation

**KANDNW**

\[
\text{DEST}[15:0] \leftarrow (\text{BITWISE NOT SRC1}[15:0]) \text{BITWISE AND SRC2}[15:0]
\]
\[
\text{DEST}[\text{MAX}_K1:1] \leftarrow 0
\]

**KANDNB**

\[
\text{DEST}[7:0] \leftarrow (\text{BITWISE NOT SRC1}[7:0]) \text{BITWISE AND SRC2}[7:0]
\]
\[
\text{DEST}[\text{MAX}_K1:8] \leftarrow 0
\]

**KANDNQ**

\[
\text{DEST}[63:0] \leftarrow (\text{BITWISE NOT SRC1}[63:0]) \text{BITWISE AND SRC2}[63:0]
\]
\[
\text{DEST}[\text{MAX}_K1:64] \leftarrow 0
\]

**KANDND**

\[
\text{DEST}[31:0] \leftarrow (\text{BITWISE NOT SRC1}[31:0]) \text{BITWISE AND SRC2}[31:0]
\]
\[
\text{DEST}[\text{MAX}_K1:32] \leftarrow 0
\]

### Intel C/C++ Compiler Intrinsic Equivalent

KANDNW __mmask16 _mm512_kandn(__mmask16 a, __mmask16 b);

### Flags Affected
None

### SIMD Floating-Point Exceptions
None

### Other Exceptions
See Exceptions Type K20.
KMOVW/KMOVB/KMOVQ/KMOVD—Move from and to Mask Registers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.0F.W0 90 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from k2/m16 and store the result in k1.</td>
</tr>
<tr>
<td>KMOVW k1, k2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 90 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from k2/m8 and store the result in k1.</td>
</tr>
<tr>
<td>KMOVB k1, k2/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.0F.W1 90 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from k2/m64 and store the result in k1.</td>
</tr>
<tr>
<td>KMOVQ k1, k2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 90 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from k2/m32 and store the result in k1.</td>
</tr>
<tr>
<td>KMODO k1, k2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.0F.W0 91 /r</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from k1 and store the result in m16.</td>
</tr>
<tr>
<td>KMOVW m16, k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 91 /r</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from k1 and store the result in m8.</td>
</tr>
<tr>
<td>KMOVB m8, k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.0F.W1 91 /r</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from k1 and store the result in m64.</td>
</tr>
<tr>
<td>KMOVQ m64, k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 91 /r</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from k1 and store the result in m32.</td>
</tr>
<tr>
<td>KMODO m32, k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.0F.W0 92 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from r32 to k1.</td>
</tr>
<tr>
<td>KMOVW k1, r32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 92 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from r32 to k1.</td>
</tr>
<tr>
<td>KMOVB k1, r32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W1 92 /r</td>
<td>RR</td>
<td>V/I</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from r64 to k1.</td>
</tr>
<tr>
<td>KMOVQ k1, r64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W0 92 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from r32 to k1.</td>
</tr>
<tr>
<td>KMODO k1, r32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.0F.W0 93 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from k1 to r32.</td>
</tr>
<tr>
<td>KMOVW k1, r32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 93 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from k1 to r32.</td>
</tr>
<tr>
<td>KMOVB r32, k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W1 93 /r</td>
<td>RR</td>
<td>V/I</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from k1 to r64.</td>
</tr>
<tr>
<td>KMODO r64, k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W0 93 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from k1 to r32.</td>
</tr>
<tr>
<td>KMODO r32, k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (w)</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (w, ModRM[7:6] must not be 11b)</td>
<td>ModRM:r/m (w, ModRM[7:6] must not be 11b)</td>
</tr>
<tr>
<td>RR</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (w, ModRM[7:6] must not be 11b)</td>
</tr>
</tbody>
</table>

Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination’s size is 32 bits. In 64-bit mode, the default GPR destination’s size is 64 bits. Note that VEX.W can only be used to modify the size of the GPR operand in 64b mode.
**Operation**

**KMOVW**
If *destination is a memory location*
   \[ \text{DEST}[15:0] \leftarrow \text{SRC}[15:0] \]
If *destination is a mask register or a GPR *
   \[ \text{DEST} \leftarrow \text{ZeroExtension} (\text{SRC}[15:0]) \]

**KMOVB**
If *destination is a memory location*
   \[ \text{DEST}[7:0] \leftarrow \text{SRC}[7:0] \]
If *destination is a mask register or a GPR *
   \[ \text{DEST} \leftarrow \text{ZeroExtension} (\text{SRC}[7:0]) \]

**KMOVQ**
If *destination is a memory location or a GPR*
   \[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0] \]
If *destination is a mask register*
   \[ \text{DEST} \leftarrow \text{ZeroExtension} (\text{SRC}[63:0]) \]

**KMOVD**
If *destination is a memory location*
   \[ \text{DEST}[31:0] \leftarrow \text{SRC}[31:0] \]
If *destination is a mask register or a GPR *
   \[ \text{DEST} \leftarrow \text{ZeroExtension} (\text{SRC}[31:0]) \]

**Intel C/C++ Compiler Intrinsic Equivalent**
KMOVW __mmask16 __mm512_kmov(__mmask16 a);

**Flags Affected**
None

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Instructions with RR operand encoding See Exceptions Type K20.
Instructions with RM or MR operand encoding See Exceptions Type K21.
**KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.0F.W0 44 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise NOT of 16 bits mask k2.</td>
</tr>
<tr>
<td>KNOTW k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 44 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise NOT of 8 bits mask k2.</td>
</tr>
<tr>
<td>KNOTB k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.0F.W1 44 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise NOT of 64 bits mask k2.</td>
</tr>
<tr>
<td>KNOTQ k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 44 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise NOT of 32 bits mask k2.</td>
</tr>
<tr>
<td>KNOTD k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

**Operation**

**KNOTW**

DEST[15:0] ← BITWISE NOT SRC[15:0]
DEST[MAX_KL-1:16] ← 0

**KNOTB**

DEST[7:0] ← BITWISE NOT SRC[7:0]
DEST[MAX_KL-1:8] ← 0

**KNOTQ**

DEST[63:0] ← BITWISE NOT SRC[63:0]
DEST[MAX_KL-1:64] ← 0

**KNOTD**

DEST[31:0] ← BITWISE NOT SRC[31:0]
DEST[MAX_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KNOTW __mmask16 _mm512_knot(__mmask16 a);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.
KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.L1.0F.w0 45 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise OR 16 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KORW k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.w0 45 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise OR 8 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KORB k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.0F.w1 45 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise OR 64 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KORQ k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.w1 45 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise OR 32 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KORD k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

**KORW**

DEST[15:0] ← SRC1[15:0] BITWISE OR SRC2[15:0]
DEST[MAX_KL-1:16] ← 0

**KORB**

DEST[7:0] ← SRC1[7:0] BITWISE OR SRC2[7:0]
DEST[MAX_KL-1:8] ← 0

**KORQ**

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAX_KL-1:64] ← 0

**KORD**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[MAX_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KORW __mmask16 _mm512_kor(__mmask16 a, __mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.
**KORTESTw/KORTESTb/KORTESTq/KORTESTd—OR Masks And Set Flags**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.0F.W0 98 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>KORTESTw k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RR V/V AVX512DQ</td>
<td></td>
<td>Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 98 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>KORTESTQ k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RR V/V AVX512Bw</td>
<td></td>
<td>Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 98 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>KORTESTD k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

**Description**
Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result. ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1’s.

**Operation**

**KORTESTW**

\[
\text{TMP}[15:0] \leftarrow \text{DEST}[15:0] \text{BITWISE OR SRC}[15:0]
\]

\[
\text{IF}(\text{TMP}[15:0]=0)
\]

\[
\text{THEN ZF} \leftarrow 1
\]

\[
\text{ELSE ZF} \leftarrow 0
\]

\[
\text{FI};
\]

\[
\text{IF}(\text{TMP}[15:0]=\text{FFFFh})
\]

\[
\text{THEN CF} \leftarrow 1
\]

\[
\text{ELSE CF} \leftarrow 0
\]

\[
\text{FI};
\]

**KORTESTB**

\[
\text{TMP}[7:0] \leftarrow \text{DEST}[7:0] \text{BITWISE OR SRC}[7:0]
\]

\[
\text{IF}(\text{TMP}[7:0]=0)
\]

\[
\text{THEN ZF} \leftarrow 1
\]

\[
\text{ELSE ZF} \leftarrow 0
\]

\[
\text{FI};
\]

\[
\text{IF}(\text{TMP}[7:0]=\text{FFh})
\]

\[
\text{THEN CF} \leftarrow 1
\]

\[
\text{ELSE CF} \leftarrow 0
\]

\[
\text{FI};
\]
KORTESTQ
TMP[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
IF(TMP[63:0]=0)
    THEN ZF ← 1
    ELSE ZF ← 0
FI;
IF(TMP[63:0]==FFFFFFFF_FFFFFFFFh)
    THEN CF ← 1
    ELSE CF ← 0
FI;

KORTESTD
TMP[31:0] ← DEST[31:0] BITWISE OR SRC[31:0]
IF(TMP[31:0]=0)
    THEN ZF ← 1
    ELSE ZF ← 0
FI;
IF(TMP[31:0]==FFFFFFFFh)
    THEN CF ← 1
    ELSE CF ← 0
FI;

**Intel C/C++ Compiler Intrinsic Equivalent**
KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);

**Flags Affected**
The ZF flag is set if the result of OR-ing both sources is all 0s.
The CF flag is set if the result of OR-ing both sources is all 1s.
The OF, SF, AF, and PF flags are set to 0.

**Other Exceptions**
See Exceptions Type K20.
KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.66.0F3A.W1 32 /r KSHIFTLW k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift left 16 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W0 32 /r KSHIFTLB k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Shift left 8 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W1 33 /r KSHIFTLQ k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift left 64 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W0 33 /r KSHIFTLD k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift left 32 bits in k2 by immediate and write result in k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

**KSHIFTLW**

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=15
   THEN DEST[15:0] ← SRC1[15:0] << COUNT;
FI;

**KSHIFTLB**

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=7
   THEN DEST[7:0] ← SRC1[7:0] << COUNT;
FI;

**KSHIFTLQ**

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=63
   THEN DEST[63:0] ← SRC1[63:0] << COUNT;
FI;
KSHIFTLD

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=31
    THEN DEST[31:0] ← SRC1[31:0] << COUNT;
FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

Compiler auto generates KSHIFTLW when needed.

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.
KSHIFTRw/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.l0.66.0F3A.W1 30 /r KSHIFTRw k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift right 16 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>VEX.l0.66.0F3A.W0 30 /r KSHIFTRB k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Shift right 8 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>VEX.l0.66.0F3A.W1 31 /r KSHIFTRQ k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift right 64 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>VEX.l0.66.0F3A.W0 31 /r KSHIFTRD k1, k2, imm8</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift right 32 bits in k2 by immediate and write result in k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTRw

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=15
    THEN DEST[15:0] ← SRC1[15:0] >> COUNT;
FI;

KSHIFTRB

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=7
    THEN DEST[7:0] ← SRC1[7:0] >> COUNT;
FI;

KSHIFTRQ

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=63
    THEN DEST[63:0] ← SRC1[63:0] >> COUNT;
FI;

KSHIFTRD

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=63
    THEN DEST[63:0] ← SRC1[63:0] >> COUNT;
FI;
**KSHIFTRD**

COUNT $\leftarrow$ imm8[7:0]  
DEST[MAX_KL-1:0] $\leftarrow$ 0  
IF COUNT $\leq$ 31  
THEN DEST[31:0] $\leftarrow$ SRC1[31:0] $>>$ COUNT;  
FI;

**Intel C/C++ Compiler Intrinsic Equivalent**  
Compiler auto generates KSHIFTRW when needed.

**Flags Affected**  
None

**SIMD Floating-Point Exceptions**  
None

**Other Exceptions**  
See Exceptions Type K20.
### KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.0F.W0 99 /r KTESTW k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 99 /r KTESTB k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.</td>
</tr>
<tr>
<td>VEX.L0.0F.W1 99 /r KTESTQ k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 99 /r KTESTD k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>ModRMreg (r)</td>
<td>ModRMr/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**KTESTW**

\[
\text{TEMP}[15:0] \leftarrow \text{SRC2}[15:0] \text{ AND SRC1}[15:0]
\]

\[
\text{IF} (\text{TEMP}[15:0] = = 0)
\]

\[
\text{THEN ZF} \leftarrow 1;
\]

\[
\text{ELSE ZF} \leftarrow 0;
\]

\[
\text{FI};
\]

\[
\text{TEMP}[15:0] \leftarrow \text{SRC2}[15:0] \text{ AND NOT SRC1}[15:0]
\]

\[
\text{IF} (\text{TEMP}[15:0] = = 0)
\]

\[
\text{THEN CF} \leftarrow 1;
\]

\[
\text{ELSE CF} \leftarrow 0;
\]

\[
\text{FI};
\]

\[
\text{AF} \leftarrow \text{OF} \leftarrow \text{PF} \leftarrow \text{SF} \leftarrow 0;
\]

**KTESTB**

\[
\text{TEMP}[7:0] \leftarrow \text{SRC2}[7:0] \text{ AND SRC1}[7:0]
\]

\[
\text{IF} (\text{TEMP}[7:0] = = 0)
\]

\[
\text{THEN ZF} \leftarrow 1;
\]

\[
\text{ELSE ZF} \leftarrow 0;
\]

\[
\text{FI};
\]

\[
\text{TEMP}[7:0] \leftarrow \text{SRC2}[7:0] \text{ AND NOT SRC1}[7:0]
\]

\[
\text{IF} (\text{TEMP}[7:0] = = 0)
\]

\[
\text{THEN CF} \leftarrow 1;
\]

\[
\text{ELSE CF} \leftarrow 0;
\]

\[
\text{FI};
\]

\[
\text{AF} \leftarrow \text{OF} \leftarrow \text{PF} \leftarrow \text{SF} \leftarrow 0;
\]
KTESTQ
TEMP[63:0] ← SRC2[63:0] AND SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI;
TEMP[63:0] ← SRC2[63:0] AND NOT SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
FI;
AF ← OF ← PF ← SF ← 0;

KTESTD
TEMP[31:0] ← SRC2[31:0] AND SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI;
TEMP[31:0] ← SRC2[31:0] AND NOT SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
FI;
AF ← OF ← PF ← SF ← 0;

Intel C/C++ Compiler Intrinsic Equivalent

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type K20.
KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.L1.66.0F.w0 4B /r KUNPCKBW k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpack and interleave 8 bits masks in k2 and k3 and write word result in k1.</td>
</tr>
<tr>
<td>VEX.NDS.L1.66.0F.w0 4B /r KUNPCKKD k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Unpack and interleave 16 bits in k2 and k3 and write double-word result in k1.</td>
</tr>
<tr>
<td>VEX.NDS.L1.66.0F.w1 4B /r KUNPCKDQ k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Unpack and interleave 32 bits masks in k2 and k3 and write quadword result in k1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>RVR</td>
</tr>
</tbody>
</table>

Description
Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

Operation
KUNPCKBW
DEST[7:0] ← SRC2[7:0]
DEST[15:8] ← SRC1[7:0]
DEST[MAX_KL-1:16] ← 0

KUNPCKWD
DEST[15:0] ← SRC2[15:0]
DEST[31:16] ← SRC1[15:0]
DEST[MAX_KL-1:32] ← 0

KUNPCKDQ
DEST[31:0] ← SRC2[31:0]
DEST[63:32] ← SRC1[31:0]
DEST[MAX_KL-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
KUNPCKBW __mmask16 _mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 _mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 _mm512_kunpackw(__mmask32 a, __mmask32 b);

Flags Affected
None

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type K20.
KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks

### Description
Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

### Operation
**KXNORW**
\[
\text{DEST}[15:0] \leftarrow \text{NOT (SRC1}[15:0] \text{ BITWISE XOR SRC2}[15:0])
\]
\[
\text{DEST}[\text{MAX}_K_L-1:16) \leftarrow 0
\]

**KXNORB**
\[
\text{DEST}[7:0] \leftarrow \text{NOT (SRC1}[7:0] \text{ BITWISE XOR SRC2}[7:0])
\]
\[
\text{DEST}[\text{MAX}_K_L-1:8) \leftarrow 0
\]

**KXNORQ**
\[
\text{DEST}[63:0] \leftarrow \text{NOT (SRC1}[63:0] \text{ BITWISE XOR SRC2}[63:0])
\]
\[
\text{DEST}[\text{MAX}_K_L-1:64) \leftarrow 0
\]

**KXNORD**
\[
\text{DEST}[31:0] \leftarrow \text{NOT (SRC1}[31:0] \text{ BITWISE XOR SRC2}[31:0])
\]
\[
\text{DEST}[\text{MAX}_K_L-1:32) \leftarrow 0
\]

**Intel C/C++ Compiler Intrinsic Equivalent**
KXNORw __mmask16 _mm512_kxnor(__mmask16 a, __mmask16 b);

### Flags Affected
None

### SIMD Floating-Point Exceptions
None

### Other Exceptions
See Exceptions Type K20.
KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.L1.0F.W0 47 /r KXORW k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise XOR 16 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 47 /r KXORB k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise XOR 8 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.0F.W1 47 /r KXORQ k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise XOR 64 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 47 /r KXORD k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Bitwise XOR 32 bits masks k2 and k3 and place result in k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

**Operation**

**KXORW**

DEST[15:0] ← SRC1[15:0] BITWISE XOR SRC2[15:0]
DEST[MAX_KL-1:16] ← 0

**KXORB**

DEST[7:0] ← SRC1[7:0] BITWISE XOR SRC2[7:0]
DEST[MAX_KL-1:8] ← 0

**KXORQ**

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAX_KL-1:64] ← 0

**KXORD**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[MAX_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KXORW __mmask16 _mm512_kxor(__mmask16 a, __mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.
LAHF—Load Status Flags into AH Register

**Op
code** | **Instruction** | **Op/
En** | **64-Bit
Mode** | **Compat/
Leg Mode** | **Description**
---|---|---|---|---|---
9F | LAHF | ZO | Invalid* | Valid | Load: \( \text{AH} \leftarrow \text{EFLAGS(SF:ZF:0:AF:0:PF:1:CF)} \).

**NOTES:**

*Valid in specific steppings. See Description section.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

**Operation**

IF 64-Bit Mode
THEN
   IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
   THEN \( \text{AH} \leftarrow \text{RFLAGS(SF:ZF:0:AF:0:PF:1:CF)} \);
   ELSE #UD;
   ELSE
      \( \text{AH} \leftarrow \text{EFLAGS(SF:ZF:0:AF:0:PF:1:CF)} \);
   FI;
FI;

**Flags Affected**

None. The state of the flags in the EFLAGS register is not affected.

**Protected Mode Exceptions**

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#UD If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.
   If the LOCK prefix is used.
LAR—Load Access Rights Byte

**Description**

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
  - Bits 19:16 are undefined.
  - Bit 20 returns the software-available bit in the descriptor.
  - Bit 21 returns the L flag.
  - Bit 22 returns the D/B flag.
  - Bit 23 returns the G flag.
  - Bits 31:24 are returned as 0.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-52.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**NOTES:**

1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

**Opcode Instruction Op/En**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 02 /r</td>
<td>LAR r16, r16/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 ← access rights referenced by r16/m16</td>
</tr>
<tr>
<td>0F 02 /r</td>
<td>LAR reg, r32/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>reg ← access rights referenced by r32/m16</td>
</tr>
</tbody>
</table>
The LAR instruction can only be executed in protected mode and IA-32e mode.

Table 3-52. Segment and Gate Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Protected Mode</th>
<th>IA-32e Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Available 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>LDT</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Busy 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>16-bit call gate</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>16-bit/32-bit task gate</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>16-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>16-bit trap gate</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Available 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>A</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>Busy 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>32-bit call gate</td>
<td>Yes</td>
</tr>
<tr>
<td>D</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>E</td>
<td>32-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>F</td>
<td>32-bit trap gate</td>
<td>No</td>
</tr>
</tbody>
</table>

Operation

IF Offset(SRC) > descriptor table limit
    THEN
        ZF ← 0;
    ELSE
        SegmentDescriptor ← descriptor referenced by SRC;
        IF SegmentDescriptor(Type) ≠ conforming code segment
            and (CPL > DPL) or (RPL > DPL)
            or SegmentDescriptor(Type) is not valid for instruction
            THEN
                ZF ← 0;
            ELSE
                DEST ← access rights from SegmentDescriptor as given in Description section;
                ZF ← 1;
            FI;
        FI;

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The LAR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The LAR instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If the memory operand effective address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory operand effective address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.

#UD If the LOCK prefix is used.
LDDQU—Load Unaligned Integer 128 Bits

**Description**

The instruction is functionally similar to (V)MOVDQU ymm/xmm, m256/m128 for loading from memory. That is: 32/16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to (V)MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by (V)LDDQU be modified and stored to the same location, use (V)MOVDQU or (V)MOVDQA instead of (V)LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the (V)MOVDQA instruction.

**Implementation Notes**

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**LDDQU (128-bit Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)
VLDDQU (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0

VLDDQU (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent
LDDQU: __m128i _mm_lddqu_si128 (__m128i * p);
VLDDQU: __m256i _mm256_lddqu_si256 (__m256i * p);

Numeric Exceptions
None

Other Exceptions
See Exceptions Type 4;
Note treatment of #AC varies.
LDMXCSR—Load MXCSR Register

### Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See "MXCSR Control and Status Register" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the (V)STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

If VLDMXCSR is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

\[ \text{MXCSR} \leftarrow m32; \]

### C/C++ Compiler Intrinsic Equivalent

```c
_mm_setcsr(unsigned int i)
```

### Numeric Exceptions

None

### Other Exceptions

See Exceptions Type 5; additionally

- &#GP: For an attempt to set reserved bits in MXCSR.
- &#UD: If VEX.vvvv ≠ 1111B.
### LDS/LES/LFS/LGS/LSS—Load Far Pointer

**Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description**
--- | --- | --- | --- | --- | ---
C5 /r | LDS r16,m16:16 | RM | Invalid | True | Load DS:r16 with far pointer from memory.
C5 /r | LDS r32,m16:32 | RM | Invalid | True | Load DS:r32 with far pointer from memory.
0F B2 /r | LSS r16,m16:16 | RM | True | True | Load SS:r16 with far pointer from memory.
0F B2 /r | LSS r32,m16:32 | RM | True | True | Load SS:r32 with far pointer from memory.
REX + 0F B2 /r | LSS r64,m16:64 | RM | True | N.E. | Load SS:r64 with far pointer from memory.
C4 /r | LES r16,m16:16 | RM | Invalid | True | Load ES:r16 with far pointer from memory.
C4 /r | LES r32,m16:32 | RM | Invalid | True | Load ES:r32 with far pointer from memory.
0F B4 /r | LFS r16,m16:16 | RM | True | True | Load FS:r16 with far pointer from memory.
0F B4 /r | LFS r32,m16:32 | RM | True | True | Load FS:r32 with far pointer from memory.
REX + 0F B4 /r | LFS r64,m16:64 | RM | True | N.E. | Load FS:r64 with far pointer from memory.
0F B5 /r | LGS r16,m16:16 | RM | True | True | Load GS:r16 with far pointer from memory.
0F B5 /r | LGS r32,m16:32 | RM | True | True | Load GS:r32 with far pointer from memory.
REX + 0F B5 /r | LGS r64,m16:64 | RM | True | N.E. | Load GS:r64 with far pointer from memory.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
--- | --- | --- | --- | ---
RM | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

**Description**

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

**64-BIT_MODE**

IF SS is loaded

THEN

IF SegmentSelector = NULL and ((RPL = 3) or (RPL ≠ 3 and RPL ≠ CPL))

THEN #GP(0);
ELSE IF descriptor is in non-canonical space
THEN #GP(0); FI;
ELSE IF Segment selector index is not within descriptor table limits
   or segment selector RPL ≠ CPL
   or access rights indicate nonwritable data segment
   or DPL ≠ CPL
   THEN #GP(selector); FI;
ELSE IF Segment marked not present
   THEN #SS(selector); FI;
FI;
SS ← SegmentSelector(SRC);
SS ← SegmentDescriptor([SRC]);
ELSE IF attempt to load DS, or ES
THEN #UD;
ELSE IF FS, or GS is loaded with non-NULL segment selector
   THEN IF Segment selector index is not within descriptor table limits
      or access rights indicate segment neither data nor readable code segment
      or segment is data or nonconforming-code segment
      and (RPL > DPL or CPL > DPL)
      THEN #GP(selector); FI;
   ELSE IF Segment marked not present
      THEN #NP(selector); FI;
   FI;
   SegmentRegister ← SegmentSelector(SRC);
   SegmentRegister ← SegmentDescriptor([SRC]);
   FI;
ELSE IF FS, or GS is loaded with a NULL selector:
   THEN
      SegmentRegister ← NULLSelector;
      SegmentRegister(DescriptorValidBit) ← 0; FI; (* Hidden flag; not accessible by software *)
   FI;
   DEST ← Offset(SRC);

PRETECTED MODE OR COMPATIBILITY MODE;
   IF SS is loaded
   THEN
      IF SegmentSelector = NULL
         THEN #GP(0);
      ELSE IF Segment selector index is not within descriptor table limits
         or segment selector RPL ≠ CPL
         or access rights indicate nonwritable data segment
         or DPL ≠ CPL
         THEN #GP(selector); FI;
      ELSE IF Segment marked not present
         THEN #SS(selector); FI;
      FI;
      SS ← SegmentSelector(SRC);
      SS ← SegmentDescriptor([SRC]);
   ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
      THEN IF Segment selector index is not within descriptor table limits
         or access rights indicate segment neither data nor readable code segment
         or segment is data or nonconforming-code segment
         and (RPL > DPL or CPL > DPL)
         THEN #GP(selector); FI;
ELSE IF Segment marked not present
THEN #NP(selector); FI;

FI;
SegmentRegister ← SegmentSelector(SRC) AND RPL;
SegmentRegister ← SegmentDescriptor([SRC]);
FI;
ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
THEN
SegmentRegister ← NULLSelector;
SegmentRegister(DescriptorValidBit) ← 0; FI; (* Hidden flag; not accessible by software *)
FI;
DEST ← Offset(SRC);

Real-Address or Virtual-8086 Mode
SegmentRegister ← SegmentSelector(SRC); FI;
DEST ← Offset(SRC);

Flags Affected
None

Protected Mode Exceptions

#UD If source operand is not a memory location.
If the LOCK prefix is used.

#GP(0) If a NULL selector is loaded into the SS register.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#GP(selector) If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL.
If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#SS(selector) If the SS register is being loaded and the segment is marked not present.

#NP(selector) If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If source operand is not a memory location.
If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#UD If source operand is not a memory location.
If the LOCK prefix is used.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
If a NULL selector is attempted to be loaded into the SS register in compatibility mode.
If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode.
If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.

#GP(Selector) If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.

#SS(0) If a memory operand effective address is non-canonical

#SS(Selector) If the SS register is being loaded and the segment is marked not present.

#NP(selector) If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If source operand is not a memory location.
If the LOCK prefix is used.
LEA—Load Effective Address

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 3-53. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Address Size</th>
<th>Action Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>16-bit effective address is calculated and stored in requested 16-bit register destination.</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32-bit effective address is calculated and stored in the requested 32-bit register destination.</td>
</tr>
</tbody>
</table>

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

In 64-bit mode, the instruction’s destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-54.

Table 3-54. 64-bit Mode LEA Operation with Address and Operand Size Attributes

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Address Size</th>
<th>Action Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>32</td>
<td>32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination.</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination.</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W).</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W).</td>
</tr>
</tbody>
</table>
Operation

IF OperandSize = 16 and AddressSize = 16
    THEN
        DEST ← EffectiveAddress(SRC); (* 16-bit address *)
    ELSE IF OperandSize = 16 and AddressSize = 32
        THEN
            temp ← EffectiveAddress(SRC); (* 32-bit address *)
            DEST ← temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 16
        THEN
            temp ← EffectiveAddress(SRC); (* 16-bit address *)
            DEST ← ZeroExtend(temp); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 32
        THEN
            DEST ← EffectiveAddress(SRC); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 16 and AddressSize = 64
        THEN
            temp ← EffectiveAddress(SRC); (* 64-bit address *)
            DEST ← temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 64
        THEN
            temp ← EffectiveAddress(SRC); (* 64-bit address *)
            DEST ← temp[0:31]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 64 and AddressSize = 64
        THEN
            DEST ← EffectiveAddress(SRC); (* 64-bit address *)
        FI;
    FI;

Flags Affected

None

Protected Mode Exceptions

#UD If source operand is not a memory location.
If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
LEAVE—High Level Procedure Exit

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C9</td>
<td>LEAVE</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set SP to BP, then pop BP.</td>
</tr>
<tr>
<td>C9</td>
<td>LEAVE</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set ESP to EBP, then pop EBP.</td>
</tr>
<tr>
<td>C9</td>
<td>LEAVE</td>
<td>ZO</td>
<td>N.E.</td>
<td>N.E.</td>
<td>Set RSP to RBP, then pop RBP.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure’s stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See “Procedure Calls for Block-Structured Languages” in Chapter 7 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction’s default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF StackAddressSize = 32
  THEN
    ESP ← EBP;
  ELSE IF StackAddressSize = 64
    THEN RSP ← RBP; Fl;
    ELSE IF StackAddressSize = 16
      THEN SP ← BP; Fl;
  Fl;

IF OperandSize = 32
  THEN EBP ← Pop();
  ELSE IF OperandSize = 64
    THEN RBP ← Pop(); Fl;
    ELSE IF OperandSize = 16
      THEN BP ← Pop(); Fl;
  Fl;

Flags Affected

None
Protected Mode Exceptions
#SS(0) If the EBP register points to a location that is not within the limits of the current stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If the stack address is in a non-canonical form.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
LFENCE—Load Fence

**Description**

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete before the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the LFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an LFENCE instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction’s opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

**Operation**

`Wait_On_Following_Instructions_Until(preceding_instructions_complete);`

**Intel C/C++ Compiler Intrinsic Equivalent**

`void _mm_lfence(void)`

**Exceptions (All Modes of Operation)**

- `#UD` If CPUID.01H:EDX.SSE2[bit 26] = 0.
  
  If the LOCK prefix is used.
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /2</td>
<td>LGDT m16&amp;32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Load m into GDTR.</td>
</tr>
<tr>
<td>0F 01 /3</td>
<td>LIDT m16&amp;32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Load m into IDTR.</td>
</tr>
<tr>
<td>0F 01 /2</td>
<td>LGDT m16&amp;64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load m into GDTR.</td>
</tr>
<tr>
<td>0F 01 /3</td>
<td>LIDT m16&amp;64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load m into IDTR.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction’s operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See “SGDT—Store Global Descriptor Table Register” in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for information on storing the contents of the GDTR and IDTR.
Operation

IF Instruction is LIDT
THEN
  IF OperandSize = 16
  THEN
    IDTR(Limit) ← SRC[0:15];
    IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
  ELSE IF 32-bit Operand Size
  THEN
    IDTR(Limit) ← SRC[0:15];
    IDTR(Base) ← SRC[16:47];
  FI;
ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
THEN
  IDTR(Limit) ← SRC[0:15];
  IDTR(Base) ← SRC[16:79];
FI;
ELSE (* Instruction is LGDT *)
IF OperandSize = 16
THEN
  GDTR(Limit) ← SRC[0:15];
  GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
ELSE IF 32-bit Operand Size
THEN
  GDTR(Limit) ← SRC[0:15];
  GDTR(Base) ← SRC[16:47];
FI;
ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
THEN
  GDTR(Limit) ← SRC[0:15];
  GDTR(Base) ← SRC[16:79];
FI;
FI;

Flags Affected
None

Protected Mode Exceptions

#UD If the LOCK prefix is used.
#GP(0) If the current privilege level is not 0.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.
#GP If the current privilege level is not 0.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the current privilege level is not 0.
#UD If the LOCK prefix is used.
#PF(fault-code) If a page fault occurs.
LLDT—Load Local Descriptor Table Register

**Opcode**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bits</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF 00 /2</td>
<td>LLDT r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Load segment selector r/m16 into LDTR.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

**Operation**

IF SRC(Offset) > descriptor table limit
    THEN #GP(segment selector); FI;

IF segment selector is valid

    Read segment descriptor;

    IF SegmentDescriptor(Type) ≠ LDT
        THEN #GP(segment selector); FI;
    IF segment descriptor is not present
        THEN #NP(segment selector); FI;

    LDTR(SegmentSelector) ← SRC;
    LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
    ELSE LDTR ← INVALID
    FI;

**Flags Affected**

None
Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#GP(selector) If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
Segment selector is beyond GDT limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NP(selector) If the LDT descriptor is not present.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The LLDT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The LLDT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the current privilege level is not 0.
If the memory address is in a non-canonical form.

#GP(selector) If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
Segment selector is beyond GDT limit.

#NP(selector) If the LDT descriptor is not present.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.
**LMSW—Load Machine Status Word**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /6</td>
<td>LMSW r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Loads r/m16 in machine status word of CR0.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on IA-32 and Intel 64 processors beginning with Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

CR0[0:3] ← SRC[0:3];

**Flags Affected**

None

**Protected Mode Exceptions**

#GP(0)  
If the current privilege level is not 0.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)  
If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  
If a page fault occurs.
#UD  
If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP  
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD  
If the LOCK prefix is used.
Virtual-8086 Mode Exceptions
#GP(0)  The LMSW instruction is not recognized in virtual-8086 mode.
#UD    If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the current privilege level is not 0.
        If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#UD    If the LOCK prefix is used.
LOCK—Assert LOCK# Signal Prefix

**Description**
Causes the processor’s LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

In most IA-32 and all Intel 64 processors, locking may occur without the LOCK# signal being asserted. See the “IA-32 Architecture Compatibility” section below for more details.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**
Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor’s cache is locked. Here, the processor's cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See “Effects of a Locked Operation on Internal Processor Caches” in Chapter 8 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on locking of caches.

**Operation**
AssertLOCK#(DurationOfAccompanyingInstruction);

**Flags Affected**
None

**Protected Mode Exceptions**
#UD If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.
Other exceptions can be generated by the instruction when the LOCK prefix is applied.
Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
LODS/LODSB/LODSW/LODSD/LODSQ—Load String

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>LODS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.</td>
</tr>
<tr>
<td>AD</td>
<td>LODS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.</td>
</tr>
<tr>
<td>AD</td>
<td>LODS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.</td>
</tr>
<tr>
<td>REX.W + AD</td>
<td>LODS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load qword at address (R)SI into RAX.</td>
</tr>
<tr>
<td>AC</td>
<td>LODSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.</td>
</tr>
<tr>
<td>AD</td>
<td>LODSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.</td>
</tr>
<tr>
<td>AD</td>
<td>LODSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.</td>
</tr>
<tr>
<td>REX.W + AD</td>
<td>LODSQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load qword at address (R)SI into RAX.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.
In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a description of the REP prefix.

**Operation**

IF AL ← SRC; (* Byte load *)

THEN AL ← SRC; (* Byte load *)

IF DF = 0

THEN (E)SI ← (E)SI + 1;
ELSE (E)SI ← (E)SI – 1;
FI;

ELSE IF AX ← SRC; (* Word load *)

THEN IF DF = 0

THEN (E)SI ← (E)SI + 2;
ELSE (E)SI ← (E)SI – 2;
FI;

ELSE IF EAX ← SRC; (* Doubleword load *)

THEN IF DF = 0

THEN (E)SI ← (E)SI + 4;
ELSE (E)SI ← (E)SI – 4;
FI;

ELSE IF RAX ← SRC; (* Quadword load *)

THEN IF DF = 0

THEN (R)SI ← (R)SI + 8;
ELSE (R)SI ← (R)SI – 8;
FI;

FI;

**Flags Affected**

None

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
LOOP/LOOPcc—Loop According to ECX Counter

**Op/En** | **Operand 1** | **Operand 2** | **Operand 3** | **Operand 4**
---|---|---|---|---
D | Offset | NA | NA | NA

**Description**

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOPcc) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (cc) is associated with each instruction to indicate the condition being tested for. Here, the LOOPcc instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

**Operation**

IF (AddressSize = 32)
  THEN Count is ECX;
ELSE IF (AddressSize = 64)
  Count is RCX;
ELSE Count is CX;
FI;

Count ← Count – 1;

IF Instruction is not LOOP
  THEN
    IF (Instruction ← LOOPE) or (Instruction ← LOOPZ)
      THEN IF (ZF = 1) and (Count ≠ 0)
        THEN BranchCond ← 1;
        ELSE BranchCond ← 0;
      FI;
    ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
      IF (ZF = 0) and (Count ≠ 0)
        THEN BranchCond ← 1;
        ELSE BranchCond ← 0;
    FI;
  FI;
ELSE (* Instruction = LOOP *)

IF (Count ≠ 0)

THEN BranchCond ← 1;
ELSE BranchCond ← 0;
FI;
FI;
FI;

IF BranchCond = 1

THEN

IF OperandSize = 32

THEN EIP ← EIP + SignExtend(DEST);
ELSE IF OperandSize = 64

THEN RIP ← RIP + SignExtend(DEST);
FI;
ELSE IF OperandSize = 16

THEN EIP ← EIP AND 0000FFFFH;
FI;
FI;
FI;

IF OperandSize = (32 or 64)

THEN IF (R/E)IP < CS.Base or (R/E)IP > CS.Limit

#GP; FI;
FI;
ELSE

Terminate loop and continue program execution at (R/E)IP;
FI;
FI;
FI;

Flags Affected

None

Protected Mode Exceptions

#GP(0) If the offset being jumped to is beyond the limits of the CS segment.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the offset being jumped to is in a non-canonical form.
#UD If the LOCK prefix is used.
LSL—Load Segment Limit

**Description**
Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit “raw” limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**NOTES:**
* For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.
### Table 3-55. Segment and Gate Descriptor Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Protected Mode</th>
<th>IA-32e Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Available 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>LDT</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Busy 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>16-bit call gate</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>16-bit/32-bit task gate</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>16-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>16-bit trap gate</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Available 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>A</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>Busy 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>32-bit call gate</td>
<td>No</td>
</tr>
<tr>
<td>D</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>E</td>
<td>32-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>F</td>
<td>32-bit trap gate</td>
<td>No</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In this case, the descriptor comprises 16 bytes; bits 12:8 of the upper 4 bytes must be 0.

**Operation**

IF SRC(Offset) > descriptor table limit
    THEN ZF ← 0; Fi;

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ conforming code segment
and (CPL > DPL) OR (RPL > DPL)
or Segment type is not valid for instruction
    THEN
        ZF ← 0;
    ELSE
        temp ← SegmentLimit([SRC]);
        IF (G ← 1)
            THEN temp ← ShiftLeft(12, temp) OR 00000FFFH;
        ELSE IF OperandSize = 32
            THEN DEST ← temp; Fi;
        ELSE IF OperandSize = 64 (* REX.W used *)
            THEN DEST (* Zero-extended *) ← temp; Fi;
        ELSE (* OperandSize = 16 *)
            DEST ← temp AND FFFFH;
        Fi;
    Fi;
Flags Affected
The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD The LSL instruction cannot be executed in real-address mode.

Virtual-8086 Mode Exceptions
#UD The LSL instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory operand effective address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD If the LOCK prefix is used.
LTR—Load Task Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /3</td>
<td>LTR r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Load r/m16 into task register.</td>
</tr>
</tbody>
</table>

**Description**

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

**Operation**

IF SRC is a NULL selector
   THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
   THEN #GP(segment selector); Fl;

Read segment descriptor;

IF segment descriptor is not for an available TSS
   THEN #GP(segment selector); Fl;
IF segment descriptor is not present
   THEN #NP(segment selector); Fl;

TSSSegmentDescriptor(busy) ← 1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)

TaskRegister(SegmentSelector) ← SRC;
TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;

**Flags Affected**

None
Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the source operand contains a NULL segment selector.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#GP(selector) If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
If the selector points to LDT or is beyond the GDT limit.

#NP(selector) If the TSS is marked not present.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The LTR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The LTR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the current privilege level is not 0.
If the memory address is in a non-canonical form.
If the source operand contains a NULL segment selector.

#GP(selector) If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
If the selector points to LDT or is beyond the GDT limit.
If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.

#NP(selector) If the TSS is marked not present.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.
**LZCNT — Count the Number of Leading Zero Bits**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F BD /r</td>
<td>RM</td>
<td>V/V</td>
<td>LZCNT</td>
<td>Count the number of leading zero bits in r/m16, return result in r16.</td>
</tr>
<tr>
<td>LZCNT r16, r/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F BD /r</td>
<td>RM</td>
<td>V/V</td>
<td>LZCNT</td>
<td>Count the number of leading zero bits in r/m32, return result in r32.</td>
</tr>
<tr>
<td>LZCNT r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 REX.W 0F BD /r</td>
<td>RM</td>
<td>V/N.E.</td>
<td>LZCNT</td>
<td>Count the number of leading zero bits in r/m64, return result in r64.</td>
</tr>
<tr>
<td>LZCNT r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR. In 64-bit mode 64-bit operand size requires REX.W=1.

**Operation**

```plaintext
temp ← OperandSize - 1
DEST ← 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
    DO
        temp ← temp - 1
        DEST ← DEST+ 1
    OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI

IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

**Flags Affected**

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.
Intel C/C++ Compiler Intrinsic Equivalent
LZCNT: unsigned __int32 _lzcnt_u32(unsigned __int32 src);
LZCNT: unsigned __int64 _lzcnt_u64(unsigned __int64 src);

Protected Mode Exceptions
#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
    If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions
#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions
#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions
Same exceptions as in Protected Mode.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
CHAPTER 4
INSTRUCTION SET REFERENCE, M-U

4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM

The notations introduced in this section are referenced in the reference pages of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM. The operation of the immediate control byte is common to these four string text processing instructions of SSE4.2. This section describes the common operations.

4.1.1 General Description

The operation of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM is defined by the combination of the respective opcode and the interpretation of an immediate control byte that is part of the instruction encoding. The opcode controls the relationship of input bytes/words to each other (determines whether the inputs terminated strings or whether lengths are expressed explicitly) as well as the desired output (index or mask).

The Imm8 Control Byte for PCMPESTRM/PCMPESTRI/PCMPISTRM/PCMPISTRI encodes a significant amount of programmable control over the functionality of those instructions. Some functionality is unique to each instruction while some is common across some or all of the four instructions. This section describes functionality which is common across the four instructions.

The arithmetic flags (ZF, CF, SF, OF, AF, PF) are set as a result of these instructions. However, the meanings of the flags have been overloaded from their typical meanings in order to provide additional information regarding the relationships of the two inputs.

PCMPxSTRx instructions perform arithmetic comparisons between all possible pairs of bytes or words, one from each packed input source operand. The boolean results of those comparisons are then aggregated in order to produce meaningful results. The Imm8 Control Byte is used to affect the interpretation of individual input elements as well as control the arithmetic comparisons used and the specific aggregation scheme.

Specifically, the Imm8 Control Byte consists of bit fields that control the following attributes:

- **Source data format** — Byte/word data element granularity, signed or unsigned elements
- **Aggregation operation** — Encodes the mode of per-element comparison operation and the aggregation of per-element comparisons into an intermediate result
- **Polarity** — Specifies intermediate processing to be performed on the intermediate result
- **Output selection** — Specifies final operation to produce the output (depending on index or mask) from the intermediate result
4.1.2 Source Data Format

Table 4-1. Source Data Format

<table>
<thead>
<tr>
<th>Imm8[1:0]</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>Unsigned bytes</td>
<td>Both 128-bit sources are treated as packed, unsigned bytes.</td>
</tr>
<tr>
<td>01b</td>
<td>Unsigned words</td>
<td>Both 128-bit sources are treated as packed, unsigned words.</td>
</tr>
<tr>
<td>10b</td>
<td>Signed bytes</td>
<td>Both 128-bit sources are treated as packed, signed bytes.</td>
</tr>
<tr>
<td>11b</td>
<td>Signed words</td>
<td>Both 128-bit sources are treated as packed, signed words.</td>
</tr>
</tbody>
</table>

If the Imm8 Control Byte has bit[0] cleared, each source contains 16 packed bytes. If the bit is set each source contains 8 packed words. If the Imm8 Control Byte has bit[1] cleared, each input contains unsigned data. If the bit is set each source contains signed data.

4.1.3 Aggregation Operation

Table 4-2. Aggregation Operation

<table>
<thead>
<tr>
<th>Imm8[3:2]</th>
<th>Mode</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>Equal any</td>
<td>The arithmetic comparison is “equal.”</td>
</tr>
<tr>
<td>01b</td>
<td>Ranges</td>
<td>Arithmetic comparison is “greater than or equal” between even indexed bytes/words of reg and each byte/word of reg/mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arithmetic comparison is “less than or equal” between odd indexed bytes/words of reg and each byte/word of reg/mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(reg/mem[m] &gt;= reg[n] for n = even, reg/mem[m] &lt;= reg[n] for n = odd)</td>
</tr>
<tr>
<td>10b</td>
<td>Equal each</td>
<td>The arithmetic comparison is “equal.”</td>
</tr>
<tr>
<td>11b</td>
<td>Equal ordered</td>
<td>The arithmetic comparison is “equal.”</td>
</tr>
</tbody>
</table>

All 256 (64) possible comparisons are always performed. The individual Boolean results of those comparisons are referred by "BoolRes[Reg/Mem element index, Reg element index]." Comparisons evaluating to "True" are represented with a 1, False with a 0 (positive logic). The initial results are then aggregated into a 16-bit (8-bit) intermediate result (IntRes1) using one of the modes described in the table below, as determined by Imm8 Control Byte bit[3:2].
See Section 4.1.6 for a description of the overrideIfDataInvalid() function used in Table 4-3.

### Table 4-3. Aggregation Operation

<table>
<thead>
<tr>
<th>Mode</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal any</td>
<td>UpperBound = imm8[0] ? 7 : 15;</td>
</tr>
<tr>
<td></td>
<td>IntRes1 = 0;</td>
</tr>
<tr>
<td></td>
<td>For j = 0 to UpperBound, j++</td>
</tr>
<tr>
<td></td>
<td>For i = 0 to UpperBound, i++</td>
</tr>
<tr>
<td></td>
<td>IntRes1[j] OR= overrideIfDataInvalid(BoolRes[j,i])</td>
</tr>
<tr>
<td>Ranges</td>
<td>UpperBound = imm8[0] ? 7 : 15;</td>
</tr>
<tr>
<td></td>
<td>IntRes1 = 0;</td>
</tr>
<tr>
<td></td>
<td>For j = 0 to UpperBound, j++</td>
</tr>
<tr>
<td></td>
<td>For i = 0 to UpperBound, i+=2</td>
</tr>
<tr>
<td></td>
<td>IntRes1[j] OR= (overrideIfDataInvalid(BoolRes[j,i]) AND overrideIfDataInvalid(BoolRes[j,i+1]))</td>
</tr>
<tr>
<td>Equal each</td>
<td>UpperBound = imm8[0] ? 7 : 15;</td>
</tr>
<tr>
<td></td>
<td>IntRes1 = 0;</td>
</tr>
<tr>
<td></td>
<td>For i = 0 to UpperBound, i++</td>
</tr>
<tr>
<td></td>
<td>IntRes1[i] = overrideIfDataInvalid(BoolRes[i,i])</td>
</tr>
<tr>
<td>Equal ordered</td>
<td>UpperBound = imm8[0] ? FFH : FFFFH</td>
</tr>
<tr>
<td></td>
<td>IntRes1 = imm8[0] ? FFH : FFFFH</td>
</tr>
<tr>
<td></td>
<td>For j = 0 to UpperBound, j++</td>
</tr>
<tr>
<td></td>
<td>For i = 0 to UpperBound-j, k=j to UpperBound, k++, i++</td>
</tr>
<tr>
<td></td>
<td>IntRes1[j] AND= overrideIfDataInvalid(BoolRes[k,i])</td>
</tr>
</tbody>
</table>

### 4.1.4 Polarity

IntRes1 may then be further modified by performing a 1’s complement, according to the value of the Imm8 Control Byte bit[4]. Optionally, a mask may be used such that only those IntRes1 bits which correspond to “valid” reg/mem input elements are complemented (note that the definition of a valid input element is dependant on the specific opcode and is defined in each opcode’s description). The result of the possible negation is referred to as IntRes2.

### Table 4-4. Polarity

<table>
<thead>
<tr>
<th>Imm8[5:4]</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>Positive Polarity (+)</td>
<td>IntRes2 = IntRes1</td>
</tr>
<tr>
<td>01b</td>
<td>Negative Polarity (-)</td>
<td>IntRes2 = -1 XOR IntRes1</td>
</tr>
<tr>
<td>10b</td>
<td>Masked (+)</td>
<td>IntRes2 = IntRes1</td>
</tr>
</tbody>
</table>


### 4.1.5 Output Selection

<table>
<thead>
<tr>
<th>Imm8[6]</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b</td>
<td>Least significant index</td>
<td>The index returned to ECX is of the least significant set bit in IntRes2.</td>
</tr>
<tr>
<td>1b</td>
<td>Most significant index</td>
<td>The index returned to ECX is of the most significant set bit in IntRes2.</td>
</tr>
</tbody>
</table>

For PCMPESTRI/PCMPISTRI, the Imm8 Control Byte bit[6] is used to determine if the index is of the least significant or most significant bit of IntRes2.

<table>
<thead>
<tr>
<th>Imm8[6]</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b</td>
<td>Bit mask</td>
<td>IntRes2 is returned as the mask to the least significant bits of XMM0 with zero extension to 128 bits.</td>
</tr>
<tr>
<td>1b</td>
<td>Byte/word mask</td>
<td>IntRes2 is expanded into a byte/word mask (based on imm8[1]) and placed in XMM0. The expansion is performed by replicating each bit into all of the bits of the byte/word of the same index.</td>
</tr>
</tbody>
</table>

Specifically for PCMPESTRM/PCMPISTRM, the Imm8 Control Byte bit[6] is used to determine if the mask is a 16 (8) bit mask or a 128 bit byte/word mask.

### 4.1.6 Valid/Invalid Override of Comparisons

PCMPxSTRx instructions allow for the possibility that an end-of-string (EOS) situation may occur within the 128-bit packed data value (see the instruction descriptions below for details). Any data elements on either source that are determined to be past the EOS are considered to be invalid, and the treatment of invalid data within a comparison pair varies depending on the aggregation function being performed.

In general, the individual comparison result for each element pair BoolRes[i,j] can be forced true or false if one or more elements in the pair are invalid. See Table 4-7.

<table>
<thead>
<tr>
<th>xmm1 byte/word</th>
<th>xmm2/ m128 byte/word</th>
<th>Imm8[3:2] = 00b (equal any)</th>
<th>Imm8[3:2] = 01b (ranges)</th>
<th>Imm8[3:2] = 10b (equal each)</th>
<th>Imm8[3:2] = 11b (equal ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Force false</td>
<td>Force false</td>
<td>Force true</td>
<td>Force true</td>
</tr>
<tr>
<td>Invalid</td>
<td>Valid</td>
<td>Force false</td>
<td>Force false</td>
<td>Force false</td>
<td>Force true</td>
</tr>
<tr>
<td>Valid</td>
<td>Invalid</td>
<td>Force false</td>
<td>Force false</td>
<td>Force false</td>
<td>Force false</td>
</tr>
<tr>
<td>Valid</td>
<td>Valid</td>
<td>Do not force</td>
<td>Do not force</td>
<td>Do not force</td>
<td>Do not force</td>
</tr>
</tbody>
</table>
### 4.1.7 Summary of Imm8 Control byte

Table 4-8. Summary of Imm8 Control Byte

<table>
<thead>
<tr>
<th>Imm8</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>------0b</td>
<td>128-bit sources treated as 16 packed bytes.</td>
</tr>
<tr>
<td>------1b</td>
<td>128-bit sources treated as 8 packed words.</td>
</tr>
<tr>
<td>------0-b</td>
<td>Packed bytes/words are unsigned.</td>
</tr>
<tr>
<td>------1-b</td>
<td>Packed bytes/words are signed.</td>
</tr>
<tr>
<td>----00-b</td>
<td>Mode is equal any.</td>
</tr>
<tr>
<td>----01-b</td>
<td>Mode is ranges.</td>
</tr>
<tr>
<td>----10-b</td>
<td>Mode is equal each.</td>
</tr>
<tr>
<td>----11-b</td>
<td>Mode is equal ordered.</td>
</tr>
<tr>
<td>----0----b</td>
<td>IntRes1 is unmodified.</td>
</tr>
<tr>
<td>----1----b</td>
<td>IntRes1 is negated (1's complement).</td>
</tr>
<tr>
<td>--0-----b</td>
<td>Negation of IntRes1 is for all 16 (8) bits.</td>
</tr>
<tr>
<td>--1-----b</td>
<td>Negation of IntRes1 is masked by reg/mem validity.</td>
</tr>
<tr>
<td>0------b</td>
<td>Index of the least significant, set, bit is used (regardless of corresponding input element validity). IntRes2 is returned in least significant bits of XMM0.</td>
</tr>
<tr>
<td>1------b</td>
<td>Index of the most significant, set, bit is used (regardless of corresponding input element validity). Each bit of IntRes2 is expanded to byte/word.</td>
</tr>
<tr>
<td>0------b</td>
<td>This bit currently has no defined effect, should be 0.</td>
</tr>
<tr>
<td>1------b</td>
<td>This bit currently has no defined effect, should be 0.</td>
</tr>
</tbody>
</table>
4.1.8 Diagram Comparison and Aggregation Process

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1 and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- **f0():** A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.
  
  \[ f0(B, C, D) \leftarrow (B \text{ AND } C) \text{ XOR } ((\text{NOT}(B) \text{ AND } D) \text{ XOR } (B \text{ AND } D) \text{ XOR } (C \text{ AND } D) \]

- **f1():** A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.
  
  \[ f1(B, C, D) \leftarrow B \text{ XOR } C \text{ XOR } D \]

- **f2():** A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.
  
  \[ f2(B, C, D) \leftarrow (B \text{ AND } C) \text{ XOR } (B \text{ AND } D) \text{ XOR } (C \text{ AND } D) \]
• f3(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as f1().
  \[ f3(B, C, D) \leftarrow B \text{ XOR } C \text{ XOR } D \]

• Ch(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).
  \[ \text{Ch}(E, F, G) \leftarrow (E \text{ AND } F) \text{ XOR } ((\text{NOT } E) \text{ AND } G) \]

• Maj(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).
  \[ \text{Maj}(A, B, C) \leftarrow (A \text{ AND } B) \text{ XOR } (A \text{ AND } C) \text{ XOR } (B \text{ AND } C) \]

ROR is rotate right operation
  \[ (A \text{ ROR } N) \leftarrow A[N-1:0] \text{ || } A[\text{Width}-1:N] \]

ROL is rotate left operation
  \[ (A \text{ ROL } N) \leftarrow A \text{ ROR } (\text{Width}-N) \]

SHR is the right shift operation
  \[ (A \text{ SHR } N) \leftarrow \text{ZEROES}[N-1:0] \text{ || } A[\text{Width}-1:N] \]

• \( \Sigma_0() \): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  \[ \Sigma_0(A) \leftarrow (A \text{ ROR } 2) \text{ XOR } (A \text{ ROR } 13) \text{ XOR } (A \text{ ROR } 22) \]

• \( \Sigma_1() \): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  \[ \Sigma_1(E) \leftarrow (E \text{ ROR } 6) \text{ XOR } (E \text{ ROR } 11) \text{ XOR } (E \text{ ROR } 25) \]

• \( \sigma_0() \): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  \[ \sigma_0(W) \leftarrow (W \text{ ROR } 7) \text{ XOR } (W \text{ ROR } 18) \text{ XOR } (W \text{ SHR } 3) \]

• \( \sigma_1() \): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  \[ \sigma_1(W) \leftarrow (W \text{ ROR } 17) \text{ XOR } (W \text{ ROR } 19) \text{ XOR } (W \text{ SHR } 10) \]

• \( K_i \): SHA1 Constants dependent on immediate i.
  \[ K0 = 0x5A827999 \]
  \[ K1 = 0x6ED9EBA1 \]
  \[ K2 = 0x8F1BBCDC \]
  \[ K3 = 0xCA62C1D6 \]

### 4.3 INSTRUCTIONS (M-U)

**MASKMOVDQU—Store Selected Bytes of Double Quadword**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F F7 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Selectively write bytes from xmm1 to memory location using the byte mask in xmm2. The default memory location is specified by DS:DI/EDI/RDI.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1G F7 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Selectively write bytes from xmm1 to memory location using the byte mask in xmm2. The default memory location is specified by DS:DI/EDI/RDI.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>RM</td>
</tr>
</tbody>
</table>

### Description
Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:
- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If VMASKMOVDQU is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

---

1. ModRM.MOD = 011B required
**Operation**

IF (MASK[7] = 1)
   THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
   THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
   (* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
   THEN DEST[DI/EDI +15] ← SRC[127:120] ELSE (* Memory location unchanged *); FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

```c
void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)
```

**Other Exceptions**

See Exceptions Type 4; additionally

#UD 
   If VEX.L= 1 
   If VEX.vvvv ≠ 1111B.
MAKMOVQ—Store Selected Bytes of Quadword

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F F7 /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Selectively write bytes from mm1 to memory location using the byte mask in mm2. The default memory location is specified by DS:DI/EDI/RDI.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-overide prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10, of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.
Operation

IF (MASK[7] = 1)
  THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); Fl;
IF (MASK[15] = 1)
  THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); Fl;
  (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
  THEN DEST[DI/EDI +15] ← SRC[63:56] ELSE (* Memory location unchanged *); Fl;

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmove_si64(__m64d, __m64n, char * p)

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
MAXPD—Maximum of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the maximum double-precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128:66:0FWIG 5F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum double-precision floating-point values between xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256:66:0FWIG 5F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128:66:0FW1 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the maximum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256:66:0FW1 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512:66:0FW1 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the maximum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

\[
\text{MAX}(\text{SRC1}, \text{SRC2})
\]

\[
\begin{align*}
\text{IF} \ (\text{SRC1} = 0.0) \ \text{and} \ (\text{SRC2} = 0.0) \ \text{THEN} \ \text{DEST} &\leftarrow \text{SRC2}; \\
\text{ELSE IF} \ (\text{SRC1} = \text{SNaN}) \ \text{THEN} \ \text{DEST} &\leftarrow \text{SRC2}; \ \text{FI}; \\
\text{ELSE IF} \ (\text{SRC2} = \text{SNaN}) \ \text{THEN} \ \text{DEST} &\leftarrow \text{SRC2}; \ \text{FI}; \\
\text{ELSE IF} \ (\text{SRC1} > \text{SRC2}) \ \text{THEN} \ \text{DEST} &\leftarrow \text{SRC1}; \\
\text{ELSE} \ \text{DEST} &\leftarrow \text{SRC2}; \ \text{FI}; \\
\end{align*}
\]

**VMAXPD (EVEX encoded versions)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j \times 64\)

\(\text{IF} \ k1[j] \ \text{OR} \ *\text{no writemask}* \ \text{THEN} \)

\(\begin{align*}
\text{IF} \ (\text{EVEX.b} = 1) \ \text{AND} \ (\text{SRC2} \ *\text{is memory}*) \ \text{THEN} \\
\text{DEST}[i+63:i] &\leftarrow \text{MAX}(\text{SRC1}[i+63:i], \text{SRC2}[63:0]) \\
\text{ELSE} \\
\text{DEST}[i+63:i] &\leftarrow \text{MAX}(\text{SRC1}[i+63:i], \text{SRC2}[i+63:i]) \\
\text{FI}; \\
\text{ELSE} \\
\text{IF} \ \text{*merging-masking*} \\
\text{THEN} \ *\text{DEST}[i+63:i] \text{ remains unchanged}* \\
\text{ELSE} \ \text{DEST}[i+63:i] &\leftarrow 0 \ \text{; zeroing-masking} \\
\text{FI}
\end{align*}\)

\(\text{FI}; \ \text{ENDFOR}\)

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

**VMAXPD (VEX.256 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])\)

\(\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])\)

\(\text{DEST}[191:128] \leftarrow \text{MAX}(\text{SRC1}[191:128], \text{SRC2}[191:128])\)

\(\text{DEST}[255:192] \leftarrow \text{MAX}(\text{SRC1}[255:192], \text{SRC2}[255:192])\)

\(\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0\)

**VMAXPD (VEX.128 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])\)

\(\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])\)

\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0\)
MAXPD (128-bit Legacy SSE version)

\[
\text{DEST}[63:0] \leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0]) \\
\text{DEST}[127:64] \leftarrow \text{MAX}(\text{DEST}[127:64], \text{SRC}[127:64]) \\
\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}
\]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VMAXPD } & \text{ _m512d } \_\text{mm512 }\_\text{max }\_\text{pd}(\_\text{m512d }a, \_\text{m512d }b); \\
\text{VMAXPD } & \text{ _m512d } \_\text{mm512 }\_\text{mask }\_\text{max }\_\text{pd}(\_\text{m512d }s, \_\text{mmask8 }k, \_\text{m512d }a, \_\text{m512d }b); \\
\text{VMAXPD } & \text{ _m512d } \_\text{mm512 }\_\text{maskz }\_\text{max }\_\text{pd}(\_\text{mmask8 }k, \_\text{m512d }a, \_\text{m512d }b); \\
\text{VMAXPD } & \text{ _m512d } \_\text{mm512 }\_\text{max }\_\text{round }\_\text{pd}(\_\text{m512d }a, \_\text{m512d }b, \_\text{int}); \\
\text{VMAXPD } & \text{ _m512d } \_\text{mm512 }\_\text{mask }\_\text{max }\_\text{round }\_\text{pd}(\_\text{mmask8 }k, \_\text{m512d }a, \_\text{m512d }b, \_\text{int}); \\
\text{VMAXPD } & \text{ _m256d } \_\text{mm256 }\_\text{max }\_\text{pd}(\_\text{m256d }s, \_\text{mmask8 }k, \_\text{m256d }a, \_\text{m256d }b); \\
\text{VMAXPD } & \text{ _m256d } \_\text{mm256 }\_\text{mask }\_\text{max }\_\text{pd}(\_\text{mmask8 }k, \_\text{m256d }a, \_\text{m256d }b); \\
\text{VMAXPD } & \text{ _m128d } \_\text{mm128 }\_\text{max }\_\text{pd}(\_\text{m128d }s, \_\text{mmask8 }k, \_\text{m128d }a, \_\text{m128d }b); \\
\text{VMAXPD } & \text{ _m128d } \_\text{mm128 }\_\text{mask }\_\text{max }\_\text{pd}(\_\text{mmask8 }k, \_\text{m128d }a, \_\text{m128d }b); \\
\end{align*}
\]

SIMD Floating-Point Exceptions
Invalid (including QNaN Source Operand), Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
MAXPS—Maximum of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP   OF 5F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the maximum single-precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.W1G 5F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum single-precision floating-point values between xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.W1G 5F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum single-precision floating-point values between ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the maximum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the maximum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

VMAXPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE DEST[i+31:i] ← 0 ; zeroing-masking
                FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMAXPS (VEX.256 encoded version)

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] ← MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] ← MAX(SRC1[191:160], SRC2[191:160])
DEST[255:224] ← MAX(SRC1[255:224], SRC2[255:224])
DEST[MAXVL-1:256] ← 0

VMAXPS (VEX.128 encoded version)

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] ← 0
MAXPS (128-bit Legacy SSE version)
DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[63:32] ← MAX(DEST[63:32], SRC[63:32])
DEST[95:64] ← MAX(DEST[95:64], SRC[95:64])
DEST[127:96] ← MAX(DEST[127:96], SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMAXPS __m512 __m512 max_ps(__m512 a, __m512 b);
VMAXPS __m512 __mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 __mm512_maskz_max_ps(__mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 __mm512_max_round_ps(__m512 a, __m512 b, int);
VMAXPS __m512 __mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m512 __mm512_maskz_max_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m256 __m256 mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMAXPS __m256 __m256_mask_max_ps(__mmask8 k, __m256 a, __m256 b);
VMAXPS __m128 __m128 mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMAXPS __m128 __m128_mask_max_ps(__mmask8 k, __m128 a, __m128 b);
VMAXPS __m256 __m256_mask_max_ps(__m256 a, __m256 b);
MAXPS __m128 __mm_max_ps(__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Invalid (including QNaN Source Operand), Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

**Opcode/Instruction** | **Op / En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
---|---|---|---|---
F2 0F 5F /r | A | V/V | SSE2 | Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.LIG.F2.0F:W1 5F /r | B | V/V | AVX | Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F:W1 5F /r | C | V/V | AVX512F | Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

VMAXSD (EVEX encoded version)

IF k[1][0] or *no writemask*
    THEN DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
    ELSE *
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        DEST[63:0] ← 0
    FI;
    FI;
DEST[127:64] ← SRC[127:64]
DEST[MAXVL-1:128] ← 0

VMAXSD (VEX.128 encoded version)

DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← SRC[127:64]
DEST[MAXVL-1:128] ← 0

MAXSD (128-bit Legacy SSE version)

DEST[63:0] ← MAX(DEST[63:0], SRC[63:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXSD __m128d _mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 5F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F:W1G 5F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST  SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST  SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST  SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST  SRC1;
    ELSE DEST  SRC2;
    FI;
}

VMAXSS (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN DEST[31:0]  MAX(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged* 
            ELSE ; zeroing-masking
                THEN DEST[31:0]  0
            FI;
        FI;
    DEST[MAXVL-1:128]  0

VMAXSS (VEX.128 encoded version)
DEST[31:0]  MAX(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128]  0

MAXSS (128-bit Legacy SSE version)
DEST[31:0]  MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMAXSS __m128 __mm_max_round_ss( __m128 a, __m128 b, int);
VMAXSS __m128 __mm_mask_max_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 __mm_maskz_max_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 __mm_max_ss( __m128 a, __m128 b)

SIMD Floating-Point Exceptions
Invalid (Including QNaN Source Operand), Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
**MFENCE—Memory Fence**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE F0</td>
<td>MFENCE</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Serializes load and store operations.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.\(^1\) The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

**Operation**

```
Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
void _mm_mfence(void)
```

**Exceptions (All Modes of Operation)**

- **#UD**
  - If CPUID.01H:EDX.SSE2[bit 26] = 0.
  - If the LOCK prefix is used.

---

\(^1\) A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.
MINPD—Minimum of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the minimum double-precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 5D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum double-precision floating-point values between xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 5D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum packed double-precision floating-point values between ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 5D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the minimum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 5D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the minimum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 5D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an NaN, then NaN is forwarded unchanged to the destination. If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

**EVEX encoded versions:** The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
**Operation**

\[
\text{MIN}(\text{SRC1}, \text{SRC2})
\]

\[
\begin{array}{l}
\text{IF } ((\text{SRC1} = 0.0) \text{ and } (\text{SRC2} = 0.0)) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \\
\text{ELSE IF } (\text{SRC1} = \text{SNaN}) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \text{FI;}
\end{array}
\]

\[
\begin{array}{l}
\text{ELSE IF } (\text{SRC2} = \text{SNaN}) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \text{FI;}
\end{array}
\]

\[
\begin{array}{l}
\text{ELSE IF } (\text{SRC1} < \text{SRC2}) \text{ THEN } \text{DEST} \leftarrow \text{SRC1};
\end{array}
\]

\[
\begin{array}{l}
\text{ELSE } \text{DEST} \leftarrow \text{SRC2}; \text{FI;}
\end{array}
\]

**VMINPD (EVEX encoded version)**

\[(\text{KL}, \text{VL}) = (2, 128), (4, 256), (8, 512)\]

\[
\text{FOR } j \leftarrow 0 \text{ TO } \text{KL}-1
\]

\[
i \leftarrow j \ast 64
\]

\[
\text{IF } k1[j] \text{ OR } *\text{no writemask}*
\]

\[
\text{THEN}
\]

\[
\begin{array}{l}
\text{IF } (\text{EVEX.b} = 1) \text{ AND (SRC2 }^{*\text{is memory}})\
\text{THEN}
\end{array}
\]

\[
\begin{array}{l}
\text{DEST}[i+63:i] \leftarrow \text{MIN}(\text{SRC1}[i+63:i], \text{SRC2}[63:0])
\end{array}
\]

\[
\begin{array}{l}
\text{ELSE}
\end{array}
\]

\[
\begin{array}{l}
\text{DEST}[i+63:i] \leftarrow \text{MIN}(\text{SRC1}[i+63:i], \text{SRC2}[i+63:i])
\end{array}
\]

\[
\begin{array}{l}
\text{FI;}
\end{array}
\]

\[
\begin{array}{l}
\text{ELSE}
\end{array}
\]

\[
\begin{array}{l}
\text{IF } ^{*\text{merging-masking}}\text{ ; merging-masking}
\end{array}
\]

\[
\begin{array}{l}
\text{THEN } ^{*\text{DEST}[i+63:i] \text{ remains unchanged}}
\end{array}
\]

\[
\begin{array}{l}
\text{ELSE } ^{\text{DEST}[i+63:i] \leftarrow 0}\text{ ; zeroing-masking}
\end{array}
\]

\[
\begin{array}{l}
\text{FI}
\end{array}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0
\]

**VMINPD (VEX.256 encoded version)**

\[
\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])
\]

\[
\text{DEST}[191:128] \leftarrow \text{MIN}(\text{SRC1}[191:128], \text{SRC2}[191:128])
\]

\[
\text{DEST}[255:192] \leftarrow \text{MIN}(\text{SRC1}[255:192], \text{SRC2}[255:192])
\]

**VMINPD (VEX.128 encoded version)**

\[
\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\]

**MINPD (128-bit Legacy SSE version)**

\[
\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}
\]
**Intel C/C++ Compiler Intrinsic Equivalent**

VMINPD __m512d _mm512_min_pd(__m512d a, __m512d b);
VMINPD __m512d _mm512_mask_min_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_maskz_min_pd(__mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_min_round_pd(__m512d a, __m512d b, int);
VMINPD __m512d _mm512_mask_min_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d _mm512_maskz_min_round_pd(__mmask8 k, __m512d a, __m512d b);
VMINPD __m256d _mm256_mask_min_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d _mm256_maskz_min_pd(__mmask8 k, __m256d a, __m256d b);
VMINPD __m128d _mm_mask_min_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d _mm_maskz_min_pd(__mmask8 k, __m128d a, __m128d b);
MINPD __m128d _mm_min_pd(__m128d a, __m128d b);

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
MINPS—Minimum of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the minimum single-precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F:W1G 5D /r VMINPS xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F:W1G 5D /r VMINPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F:W0 5D /r VMINPS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the minimum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F:W0 5D /r VMINPS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the minimum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F:W0 5D /r VMINPS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst(sae)</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

\[
\text{MIN}(\text{SRC1}, \text{SRC2}) \{
\begin{align*}
\text{IF } & ((\text{SRC1} = 0.0) \text{ and } (\text{SRC2} = 0.0)) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \\
\text{ELSE IF } & (\text{SRC1} = \text{SNaN}) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \\
\text{ELSE IF } & (\text{SRC2} = \text{SNaN}) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \\
\text{ELSE IF } & (\text{SRC1} < \text{SRC2}) \text{ THEN } \text{DEST} \leftarrow \text{SRC1}; \\
\text{ELSE } & \text{DEST} \leftarrow \text{SRC2}; \\
\end{align*}
\}
\]

VMINPS (EVEX encoded version)

\[(\text{KL, VL}) = (4, 128), (8, 256), (16, 512)\]

\[
\text{FOR } j \leftarrow 0 \text{ TO KL-1} \\text{ FOR } i \leftarrow j \times 32 \\
\text{IF } k1[j] \text{ OR *no writemask*} \text{ THEN} \\
\text{IF (EVEX.b = 1) AND (SRC2 *is memory*)} \text{ THEN} \\
\text{DEST}[i+31:i] \leftarrow \text{MIN}(\text{SRC1}[i+31:i], \text{SRC2}[31:0]) \\
\text{ELSE} \\
\text{DEST}[i+31:i] \leftarrow \text{MIN}(\text{SRC1}[i+31:i], \text{SRC2}[i+31:i]) \\
\text{FI;} \\
\text{ELSE} \\
\text{IF *merging-masking*} \text{ THEN } \text{*DEST}[i+31:i] \text{ remains unchanged*} \\
\text{ELSE } \text{DEST}[i+31:i] \leftarrow 0 \text{ ; zeroing-masking} \\
\text{FI} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0
\]

VMINPS (VEX.256 encoded version)

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0]) \\
\text{DEST}[63:32] & \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32]) \\
\text{DEST}[95:64] & \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64]) \\
\text{DEST}[127:96] & \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96]) \\
\text{DEST}[159:128] & \leftarrow \text{MIN}(\text{SRC1}[159:128], \text{SRC2}[159:128]) \\
\text{DEST}[191:160] & \leftarrow \text{MIN}(\text{SRC1}[191:160], \text{SRC2}[191:160]) \\
\text{DEST}[223:192] & \leftarrow \text{MIN}(\text{SRC1}[223:192], \text{SRC2}[223:192]) \\
\text{DEST}[255:224] & \leftarrow \text{MIN}(\text{SRC1}[255:224], \text{SRC2}[255:224])
\end{align*}
\]

VMINPS (VEX.128 encoded version)

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0]) \\
\text{DEST}[63:32] & \leftarrow \text{MIN}(\text{SRC1}[63:32], \text{SRC2}[63:32]) \\
\text{DEST}[95:64] & \leftarrow \text{MIN}(\text{SRC1}[95:64], \text{SRC2}[95:64]) \\
\text{DEST}[127:96] & \leftarrow \text{MIN}(\text{SRC1}[127:96], \text{SRC2}[127:96]) \\
\text{DEST}[\text{MAXVL-1:128}] & \leftarrow 0
\end{align*}
\]
MINPS (128-bit Legacy SSE version)
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMINPS __m512 _mm512_min_ps(__m512 a, __m512 b);
VMINPS __m512 __m512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 __m512_maskz_min_ps(__mmask16 k, __m512 a, __m512 b);
VMINPS __m512 __m512_min_round_ps(__m512 a, __m512 b, int);
VMINPS __m512 __m512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m512 __m512_maskz_min_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMINPS __m256 __m256_mask_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMINPS __m256 __m256_maskz_min_ps(__mmask8 k, __m256 a, __m256 b);
VMINPS __m128 __m128_mask_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMINPS __m128 __m128_maskz_min_ps(__mmask8 k, __m128 a, __m128 b);
VMINPS __m256 __m256_min_ps(__m256 a, __m256 b);
MINPS __m128 __m128_min_ps(__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Invalid (including QNaN Source Operand), Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 5D /r MINSD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F:W1 5D /r VMINSN xmm1, xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F:W1 5D /r VMINSD xmm1 {k1}[z], xmm2, xmm3/m64{sae}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.</td>
</tr>
</tbody>
</table>

### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMINSN is encoded with VEX.L=0. Encoding VMINSN with VEX.L=1 may encounter unpredictable behavior across different processor generations.
**Operation**

\[
\text{MIN}(\text{SRC1}, \text{SRC2}) \\
\{ \\
\quad \text{IF } ((\text{SRC1} = 0.0) \text{ and } (\text{SRC2} = 0.0)) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \\
\quad \text{ELSE IF } (\text{SRC1} = \text{SNaN}) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \text{ FI}; \\
\quad \text{ELSE IF } (\text{SRC2} = \text{SNaN}) \text{ THEN } \text{DEST} \leftarrow \text{SRC2}; \text{ FI}; \\
\quad \text{ELSE IF } (\text{SRC1} < \text{SRC2}) \text{ THEN } \text{DEST} \leftarrow \text{SRC1}; \\
\quad \text{ELSE } \text{DEST} \leftarrow \text{SRC2}; \\
\quad \text{FI}; \\
\}
\]

**MINSD (EVEX encoded version)**

IF \( k1[0] \) or "*no writemask*"

THEN \( \text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0]) \)

ELSE

\begin{align*}
\quad & \text{IF } \text{*merging-masking*} \\
& \quad \text{THEN } \text{*DEST}[63:0] \text{ remains unchanged*} \\
& \quad \text{ELSE } \text*zeroing-masking* \\
& \quad \text{THEN } \text{DEST}[63:0] \leftarrow 0 \\
& \text{FI}; \\
\end{align*}

\( \text{FI}; \)

\begin{align*}
\text{DEST}[127:64] & \leftarrow \text{SRC1}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0 \\
\end{align*}

**MINSD (VEX.128 encoded version)**

\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0]) \\
\text{DEST}[127:64] & \leftarrow \text{SRC1}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0 \\
\end{align*}

**MINSD (128-bit Legacy SSE version)**

\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0]) \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0 \\
\end{align*}

**Intel C/C++ Compiler Intrinsic Equivalent**

\begin{align*}
\text{VMINSD } \_\text{m128d } \_\text{mm_min_round_sd}(\_\text{m128d } a, \_\text{m128d } b, \text{int}); \\
\text{VMINSD } \_\text{m128d } \_\text{mm_mask_min_round_sd}(\_\text{m128d } s, \_\text{mmask8 } k, \_\text{m128d } a, \_\text{m128d } b, \text{int}); \\
\text{VMINSD } \_\text{m128d } \_\text{mm_maskz_min_round_sd}(\_\text{mmask8 } k, \_\text{m128d } a, \_\text{m128d } b, \text{int}); \\
\text{MINSD } \_\text{m128d } \_\text{mm_min_sd}(\_\text{m128d } a, \_\text{m128d } b) \\
\end{align*}

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5D /r MINSS xmm1,xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F:WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 5D /r VMINSS xmm1 {k1}{k2}, xmm2, xmm3/m32{sa}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

MINSS (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[MAXVL-1:128] ← 0

VMINSS (VEX.128 encoded version)
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] ← 0

MINSS (128-bit Legacy SSE version)
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMINSS __m128 _mm_min_round_ss( __m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions
Invalid (Including QNaN Source Operand), Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
MONITOR—Set Up Monitor Address

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compats/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 C8</td>
<td>MONITOR</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:RAX/EAX/AX.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The address is specified in RAX/EAX/AX and the size is based on the effective address size of the encoded instruction. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, "Multiple-Processor Management" of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

### Intel C/C++ Compiler Intrinsic Equivalent

MONITOR: void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

### Numeric Exceptions

None
Protected Mode Exceptions

#GP(0)  If the value in EAX is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
   If ECX ≠ 0.

#SS(0)  If the value in EAX is outside the SS segment limit.

#PF(fault-code)  For a page fault.

#UD  If CPUID.01H:ECX.MONITOR[bit 3] = 0.
   If current privilege level is not 0.

Real Address Mode Exceptions

#GP  If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
   If ECX ≠ 0.

#SS  If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.

#UD  If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD  The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.
   If RCX ≠ 0.

#SS(0)  If the SS register is used to access memory and the value in EAX is in a non-canonical form.

#PF(fault-code)  For a page fault.

#UD  If the current privilege level is not 0.
   If CPUID.01H:ECX.MONITOR[bit 3] = 0.
## MOV—Move

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>88/8r</td>
<td>MOV r/m8,r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>REX + 88/8r</td>
<td>MOV r/m8, ⋆r8 **</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>89/8r</td>
<td>MOV r/m16,r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r16 to r/m16.</td>
</tr>
<tr>
<td>89/8r</td>
<td>MOV r/m32,r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 89/8r</td>
<td>MOV r/m64,r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to r/m64.</td>
</tr>
<tr>
<td>8A/8r</td>
<td>MOV r8,r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>REX + 8A/8r</td>
<td>MOV r8**,r/m8***</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>8B/8r</td>
<td>MOV r16,r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m16 to r16.</td>
</tr>
<tr>
<td>8B/8r</td>
<td>MOV r32,r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m32 to r32.</td>
</tr>
<tr>
<td>REX.W + 8B/8r</td>
<td>MOV r64,r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r/m64 to r32.</td>
</tr>
<tr>
<td>8C/8r</td>
<td>MOV r/m16,Sreg**</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move segment register to r/m16.</td>
</tr>
<tr>
<td>REX.W + 8C/8r</td>
<td>MOV r16/r32/r16, Sreg**</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move zero extended 16-bit segment register to r16/r32/r64/m16.</td>
</tr>
<tr>
<td>REX.W + 8C/8r</td>
<td>MOV r64/m16, Sreg**</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move zero extended 16-bit segment register to r64/m16.</td>
</tr>
<tr>
<td>8E/8r</td>
<td>MOV Sreg,r/m16**</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m16 to segment register.</td>
</tr>
<tr>
<td>REX.W + 8E/8r</td>
<td>MOV Sreg,r/m64**</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move lower 16 bits of r/m64 to segment register.</td>
</tr>
<tr>
<td>A0/8r</td>
<td>MOV AL,moffs8*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte at (seg:offset) to AL.</td>
</tr>
<tr>
<td>REX.W + A0/8r</td>
<td>MOV AL,moffs8*</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move byte at (offset) to AL.</td>
</tr>
<tr>
<td>A1/8r</td>
<td>MOV AX,moffs16*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move word at (seg:offset) to AX.</td>
</tr>
<tr>
<td>A1/8r</td>
<td>MOV EAX,moffs32*</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move doubleword at (seg:offset) to EAX.</td>
</tr>
<tr>
<td>REX.W + A1/8r</td>
<td>MOV RAX,moffs64*</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move quadword at (offset) to RAX.</td>
</tr>
<tr>
<td>A2/8r</td>
<td>MOV moffs8,AL</td>
<td>TD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move AL to (seg:offset).</td>
</tr>
<tr>
<td>REX.W + A2/8r</td>
<td>MOV moffs8,r.moffs8*</td>
<td>TD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move AL to (offset).</td>
</tr>
<tr>
<td>REX.W + A3/8r</td>
<td>MOV moffs64*,RAX</td>
<td>TD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move RAX to (offset).</td>
</tr>
<tr>
<td>B0+ rb ib</td>
<td>MOV r8, imm8</td>
<td>OI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm8 to r8.</td>
</tr>
<tr>
<td>REX + B0+ rb ib</td>
<td>MOV r8, ⋆imm8</td>
<td>OI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm8 to r8.</td>
</tr>
<tr>
<td>B8+ rw iw</td>
<td>MOV r16, imm16</td>
<td>OI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm16 to r16.</td>
</tr>
<tr>
<td>B8+ rd id</td>
<td>MOV r32, imm32</td>
<td>OI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm32 to r32.</td>
</tr>
<tr>
<td>REX.W + B8+ rd io</td>
<td>MOV r64, imm64</td>
<td>OI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm64 to r64.</td>
</tr>
<tr>
<td>C6 /l0 ib</td>
<td>MOV r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm8 to r/m8.</td>
</tr>
<tr>
<td>REX + C6 /l0 ib</td>
<td>MOV r/m8**, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm8 to r/m8.</td>
</tr>
<tr>
<td>C7 /l0 iw</td>
<td>MOV r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm16 to r/m16.</td>
</tr>
<tr>
<td>C7 /l0 id</td>
<td>MOV r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + C7 /l0 id</td>
<td>MOV r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm32 sign extended to 64-bits to r/m64.</td>
</tr>
</tbody>
</table>
**NOTES:**
* The `moffs8`, `moffs16`, `moffs32` and `moffs64` operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.
** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following “Description” section for further information).
*** In 64-bit mode, `r/m8` can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>FD</td>
<td>AL/AX/EAX/RAX</td>
<td>Moffs</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>TD</td>
<td>Moffs (w)</td>
<td>AL/AX/EAX/RAX</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>OI</td>
<td>opcode + rd (w)</td>
<td>imm8/16/32/64</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (w)</td>
<td>imm8/16/32/64</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the “Operation” algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, stack-pointer value) before an event can be delivered. See Section 6.8.3, "Masking Exceptions and Interrupts When Switching Stacks," in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

When executing MOV Reg, Sreg, the processor copies the content of Sreg to the 16 least significant bits of the general-purpose register. The upper bits of the destination register are zero for most IA-32 processors (Pentium Pro processors and later) and all Intel 64 processors, with the exception that bits 31:16 are undefined for Intel Quark X1000 processors, Pentium and earlier processors.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.
Operation

DEST ← SRC;

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

IF SS is loaded
   THEN
      IF segment selector is NULL
         THEN #GP(0); FI;
      IF segment selector index is outside descriptor table limits
         OR segment selector's RPL ≠ CPL
         OR segment is not a writable data segment
         OR DPL ≠ CPL
         THEN #GP(selector); FI;
      IF segment not marked present
         THEN #SS(selector);
         ELSE
            SS ← segment selector;
            SS ← segment descriptor; FI;
   FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector
   THEN
      IF segment selector index is outside descriptor table limits
         OR segment is not a data or readable code segment
         OR ((segment is a data or nonconforming code segment) AND ((RPL > DPL) or (CPL > DPL)))
         THEN #GP(selector); FI;
      IF segment not marked present
         THEN #NP(selector);
         ELSE
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor; FI;
   FI;

IF DS, ES, FS, or GS is loaded with NULL selector
   THEN
      SegmentRegister ← segment selector;
      SegmentRegister ← segment descriptor;
   FI;

Flags Affected

None
Protected Mode Exceptions

#GP(0)  If attempt is made to load SS register with NULL segment selector.
       If the destination operand is in a non-writable segment.
       If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
       If the DS, ES, FS, or GS register contains a NULL segment selector.

#GP(selector)  If segment selector index is outside descriptor table limits.
            If the SS register is being loaded and the segment selector’s RPL and the segment descriptor’s
            DPL are not equal to the CPL.
            If the SS register is being loaded and the segment pointed to is a non-writable data segment.
            If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or
            readable code segment.
            If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or
            nonconforming code segment, and either the RPL or the CPL is greater than the DPL.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#SS(selector)  If the SS register is being loaded and the segment pointed to is marked not present.

#NP  If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the
        current privilege level is 3.

#UD  If attempt is made to load the CS register.
       If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS  If a memory operand effective address is outside the SS segment limit.

#UD  If attempt is made to load the CS register.
       If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.

#UD  If attempt is made to load the CS register.
       If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
   If an attempt is made to load SS register with NULL segment selector when CPL = 3.
   If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.

#GP(selector) If segment selector index is outside descriptor table limits.
   If the memory access to the descriptor table is non-canonical.
   If the SS register is being loaded and the segment selector’s RPL and the segment descriptor’s DPL are not equal to the CPL.
   If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
   If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
   If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.

#SS(0) If the stack address is in a non-canonical form.

#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If attempt is made to load the CS register.
   If the LOCK prefix is used.
**MOV—Move to/from Control Registers**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 20/r MOV r32, CR0-CR7</td>
<td>MR</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move control register to r32.</td>
</tr>
<tr>
<td>0F 20/r MOV r64, CR0-CR7</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move extended control register to r64.</td>
</tr>
<tr>
<td>REX.R + 0F 20 /0 MOV r64, CR8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move extended CR8 to r64.¹</td>
</tr>
<tr>
<td>0F 22 /r MOV CR0-CR7, r32</td>
<td>RM</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move r32 to control register.</td>
</tr>
<tr>
<td>0F 22 /r MOV CR0-CR7, r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to extended control register.</td>
</tr>
<tr>
<td>REX.R + 0F 22 /0 MOV CR8, r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to extended CR8.¹</td>
</tr>
</tbody>
</table>

**NOTE:**
1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the reg field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the mod field are ignored. The r/m field specifies the general-purpose register loaded or read. Attempts to reference CR1, CR5, CR6, CR7, and CR9–CR15 result in undefined opcode (#UD) exceptions.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4’s reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction’s default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. Use of

---

1. MOV CR8 instructions are serializing instructions.
the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

DEST ← SRC;

**Flags Affected**

The OF, SF, ZF, AF, PF, and CF flags are undefined.

**Protected Mode Exceptions**

- **#GP(0)** If the current privilege level is not 0.
- If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
- If an attempt is made to write a 1 to any reserved bit in CR4.
- If an attempt is made to write 1 to CR4.PCIDE.
- If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.

- **#UD** If the LOCK prefix is used.
- If an attempt is made to access CR1, CR5, CR6, or CR7.

**Real-Address Mode Exceptions**

- **#GP** If an attempt is made to write a 1 to any reserved bit in CR4.
- If an attempt is made to write 1 to CR4.PCIDE.
- If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).

- **#UD** If the LOCK prefix is used.
- If an attempt is made to access CR1, CR5, CR6, or CR7.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** These instructions cannot be executed in virtual-8086 mode.

**Compatibility Mode Exceptions**

- **#GP(0)** If the current privilege level is not 0.
- If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
- If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
- If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.
- If an attempt is made to write a 1 to any reserved bit in CR3.
- If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].

- **#UD** If the LOCK prefix is used.
- If an attempt is made to access CR1, CR5, CR6, or CR7.
64-Bit Mode Exceptions

#GP(0)     If the current privilege level is not 0.
            If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
            If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
            If an attempt is made to clear CR0.PG[bit 31].
            If an attempt is made to write a 1 to any reserved bit in CR4.
            If an attempt is made to write a 1 to any reserved bit in CR8.
            If an attempt is made to write a 1 to any reserved bit in CR3.
            If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].

#UD        If the LOCK prefix is used.
            If an attempt is made to access CR1, CR5, CR6, or CR7.
            If the REX.R prefix is used to specify a register other than CR8.
MOV—Move to/from Debug Registers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 21/r MOV r32, DR0–DR7</td>
<td>MR</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move debug register to r32.</td>
</tr>
<tr>
<td>0F 21/r MOV r64, DR0–DR7</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move extended debug register to r64.</td>
</tr>
<tr>
<td>0F 23/r MOV DR0–DR7, r32</td>
<td>RM</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move r32 to debug register.</td>
</tr>
<tr>
<td>0F 23/r MOV DR0–DR7, r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to extended debug register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 17.2, “Debug Registers”, of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

At the opcode level, the reg field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the mod field are ignored. The r/m field specifies the general-purpose register loaded or read.

In 64-bit mode, the instruction’s default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8–R15). Use of the REX.W or 66H prefix is ignored. Use of the REX.R prefix causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST ← SRC;
    FI;

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.
Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

#UD If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.

#DB If the LOCK prefix is used.

#DB If any debug register is accessed while the DR7.GD[bit 13] = 1.

Real-Address Mode Exceptions

#UD If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.

#DB If any debug register is accessed while the DR7.GD[bit 13] = 1.

Virtual-8086 Mode Exceptions

#GP(0) The debug registers cannot be loaded or read when in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.

If an attempt is made to write a 1 to any of bits 63:32 in DR6.

If an attempt is made to write a 1 to any of bits 63:32 in DR7.

#UD If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.

If the LOCK prefix is used.

If the REX.R prefix is used.

#DB If any debug register is accessed while the DR7.GD[bit 13] = 1.
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 28 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>MOVAPD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 29 /r</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>MOVAPD xmm2/m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F:W1G 28 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VMOVAPD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F:W1G 29 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VMOVAPD xmm2/m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F:W1G 28 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed double-precision floating-point values from ymm2/mem to ymm1.</td>
</tr>
<tr>
<td>VMOVAPD ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F:W1G 29 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem.</td>
</tr>
<tr>
<td>VMOVAPD ymm2/m256, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 28 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVAPD xmm1 [k1]z, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 28 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVAPD ymm1 [k1]z, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 28 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVAPD zmm1 [k1]z, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 29 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed double-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>VMOVAPD xmm2/m128 [k1]z, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 29 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed double-precision floating-point values from ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>VMOVAPD ymm2/m256 [k1]z, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 29 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>VMOVAPD zmm2/m512 [k1]z, zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description

Moves 2, 4 or 8 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register destination are zeroed.

Operation

VMOVAPD (EVEX encoded versions, register-copy form)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

\[\text{FOR } j \leftarrow 0 \text{ TO } KL-1\]

\[i \leftarrow j \times 64\]

\[\text{IF } k1[j] \text{ OR } \text{no writemask}\]

\[\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{SRC}[i+63:i]\]

\[\text{ELSE}\]

\[\text{IF } \text{merging-masking}\]

\[\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged}\]

\[\text{ELSE } \text{DEST}[i+63:i] \leftarrow 0\]

\[\text{FI}\]

\[\text{FI}\]

\[\text{ENDFOR}\]

\[\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\]
VMOVAPD (EVEX encoded versions, store-form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
      ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;

VMOVAPD (EVEX encoded versions, load-form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE DEST[i+63:i] ← 0 ; zeroing-masking
      FI
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVAPD (VEX.256 encoded version, load - and register copy)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0

VMOVAPD (VEX.256 encoded version, store-form)
DEST[255:0] ← SRC[255:0]

VMOVAPD (VEX.128 encoded version, load - and register copy)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)

(V)MOVAPD (128-bit store-form version)
DEST[127:0] ← SRC[127:0]
Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m512d _mm512_load_pd( void * m);
VMOVAPD __m512d _mm512_mask_load_pd( __m512d s, __mmask8 k, void * m);
VMOVAPD __m512d _mm512_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void _mm512_store_pd( void * d, __m512d a);
VMOVAPD void _mm512_mask_store_pd( void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d _mm256_mask_load_pd( __m256d s, __mmask8 k, void * m);
VMOVAPD __m256d _mm256_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void _mm256_mask_store_pd( void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d _mm_mask_load_pd( __m128d s, __mmask8 k, void * m);
VMOVAPD __m128d _mm_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void _mm_mask_store_pd( void * d, __mmask8 k, __m128d a);
MOVAPD __m256d _mm256_load_pd( double * p);
MOVAPD void _mm256_store_pd( double * p, __m256d a);
MOVAPD __m128d _mm_load_pd( double * p);
MOVAPD void _mm_store_pd( double * p, __m128d a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1.
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
### MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 28 /r MOVAPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>NP 0F 29 /r MOVAPS xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>SSE</td>
<td>Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128:0F:W1G 28 /r VMOVAPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VEX.128:0F:W1G 29 /r VMOVAPS ymm1, xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.256:0F:W1G 28 /r VMOVAPS ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single-precision floating-point values from ymm2/mem to ymm1.</td>
</tr>
<tr>
<td>VEX.256:0F:W1G 29 /r VMOVAPS ymm2/m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.</td>
</tr>
<tr>
<td>EVEX.128:0F:W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256:0F:W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512:0F:W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128:0F:W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256:0F:W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512:0F:W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Moves 4, 8 or 16 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.
Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:
Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the
destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32
memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between
two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on
a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-
point values to and from unaligned memory locations, use the VMOVUPS instruction.

EVEX.256 and EVEX.256 encoded version:
Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the
destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory
location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM
registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte
boundary or a general-protection exception (#GP) will be generated.

128-bit versions:
Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the
destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory
location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM
registers. When the source or destination operand is a memory operand, the operand must be aligned on a
16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-
point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain
unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

Operation

VMOVAPS (EVEX encoded versions, register-copy form)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]
\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1
\]
\[
i \leftarrow j * 32
\]
\[
\text{IF k1}[j] \text{ OR } \text{no writemask*}
\]
\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{SRC}[i+31:i]
\]
\[
\text{ELSE}
\]
\[
\text{IF *merging-masking* ; merging-masking}
\]
\[
\text{THEN *DEST}[i+31:i] \text{ remains unchanged*}
\]
\[
\text{ELSE DEST}[i+31:i] \leftarrow 0 \text{ ; zeroing-masking}
\]
\[
\text{FI}
\]
\[
\text{FI;}
\]
\[
\text{ENDFOR}
\]
\[
\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0
\]

VMOVAPS (EVEX encoded versions, store-form)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]
\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1
\]
\[
i \leftarrow j * 32
\]
\[
\text{IF k1}[j] \text{ OR } \text{no writemask*}
\]
\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{SRC}[i+31:i]
\]
\[
\text{ELSE}
\]
\[
\text{IF *merging-masking* ; merging-masking}
\]
\[
\text{THEN *DEST}[i+31:i] \text{ remains unchanged*}
\]
\[
\text{ELSE DEST}[i+31:i] \leftarrow 0 \text{ ; zeroing-masking}
\]
\[
\text{FI}
\]
\[
\text{FI;}
\]
\[
\text{ENDFOR}
\]
VMOVAPS (EVEX encoded versions, load-form)

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 32\)
  IF \(k1[j]\) OR *no writemask*
    THEN \(DEST[i+31:i] \leftarrow SRC[i+31:i]\)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged* ; zeroing-masking
    ELSE \(DEST[i+31:i] \leftarrow 0\)
    FI
  FI;
ENDFOR

\(DEST[\text{MAXVL-1:VL}] \leftarrow 0\)

VMOVAPS (VEX.256 encoded version, load - and register copy)

\(DEST[255:0] \leftarrow SRC[255:0]\)

\(DEST[\text{MAXVL-1:256}] \leftarrow 0\)

VMOVAPS (VEX.256 encoded version, store-form)

\(DEST[255:0] \leftarrow SRC[255:0]\)

VMOVAPS (VEX.128 encoded version, load - and register copy)

\(DEST[127:0] \leftarrow SRC[127:0]\)

\(DEST[\text{MAXVL-1:128}] \leftarrow 0\)

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)

\(DEST[127:0] \leftarrow SRC[127:0]\)

\(DEST[\text{MAXVL-1:128}] \leftarrow 0\)

(V)MOVAPS (128-bit store-form version)

\(DEST[127:0] \leftarrow SRC[127:0]\)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPS __m512 _mm512_load_ps( void * m);
VMOVAPS __m512 _mm512_mask_load_ps(__m512 s, __mmask16 k, void * m);
VMOVAPS __m512 _mm512_maskz_load_ps( __mmask16 k, void * m);
VMOVAPS void _mm512_store_ps( void * d, __m512 a);
VMOVAPS void _mm512_mask_store_ps( void * d, __mmask16 k, __m512 a);
VMOVAPS __m256 _mm256_mask_load_ps( __mmask16 k, void * s);
VMOVAPS __m256 _mm256_maskz_load_ps( __mmask8 k, void * s);
VMOVAPS void _mm256_mask_store_ps( void * d, __mmask8 k, __m256 a);
VMOVAPS __m128 _mm128_mask_load_ps( __mmask8 k, void * s);
VMOVAPS __m128 _mm128_maskz_load_ps( __mmask8 k, void * s);
VMOVAPS void _mm128_mask_store_ps( void * d, __mmask8 k, __m128 a);
MOVAPS _m256 _mm256_load_ps (float * p);
MOVAPS _mm256_store_ps(float * p, __m256 a);
MOVAPS _m128 _mm_load_ps (float * p);
MOVAPS void _mm_store_ps(float * p, __m128 a);

SIMD Floating-Point Exceptions

None
Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally
#UD               If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E1.
MOVBE—Move Data After Swapping Bytes

**Description**

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[ TEMP \leftarrow SRC \]

\[
\text{IF (OperandSize = 16)}
\]

\[
\text{THEN}
\]

\[
\text{DEST}[7:0] \leftarrow \text{TEMP}[15:8];
\]

\[
\text{DEST}[15:8] \leftarrow \text{TEMP}[7:0];
\]

\[
\text{ELSE IF (OperandSize = 32)}
\]

\[
\text{DEST}[7:0] \leftarrow \text{TEMP}[31:24];
\]

\[
\text{DEST}[15:8] \leftarrow \text{TEMP}[23:16];
\]

\[
\text{DEST}[23:16] \leftarrow \text{TEMP}[15:8];
\]

\[
\text{DEST}[31:23] \leftarrow \text{TEMP}[7:0];
\]

\[
\text{ELSE IF (OperandSize = 64)}
\]

\[
\text{DEST}[7:0] \leftarrow \text{TEMP}[63:56];
\]

\[
\text{DEST}[15:8] \leftarrow \text{TEMP}[55:48];
\]

\[
\text{DEST}[23:16] \leftarrow \text{TEMP}[47:40];
\]

\[
\text{DEST}[31:24] \leftarrow \text{TEMP}[39:32];
\]

\[
\text{DEST}[39:32] \leftarrow \text{TEMP}[31:24];
\]

\[
\text{DEST}[47:40] \leftarrow \text{TEMP}[23:16];
\]

\[
\text{DEST}[55:48] \leftarrow \text{TEMP}[15:8];
\]

\[
\text{DEST}[63:56] \leftarrow \text{TEMP}[7:0];
\]

\[
\text{FI;}
\]
Flags Affected
None

Protected Mode Exceptions
#GP(0)  If the destination operand is in a non-writable segment.
         If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
         If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If CPUID.01H:ECX.MOVBE[bit 22] = 0.
       If the LOCK prefix is used.
       If REP (F3H) prefix is used.

Real-Address Mode Exceptions
#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS  If a memory operand effective address is outside the SS segment limit.
#UD  If CPUID.01H:ECX.MOVBE[bit 22] = 0.
       If the LOCK prefix is used.
       If REP (F3H) prefix is used.

Virtual-8086 Mode Exceptions
#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If CPUID.01H:ECX.MOVBE[bit 22] = 0.
       If the LOCK prefix is used.
       If REP (F3H) prefix is used.
       If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0)  If the memory address is in a non-canonical form.
#SS(0)  If the stack address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If CPUID.01H:ECX.MOVBE[bit 22] = 0.
       If the LOCK prefix is used.
       If REP (F3H) prefix is used.
### MOVD/MOVQ—Move Doubleword/Move Quadword

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 6E /r MOVQ mm, r/m32</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Move quadword from r/m64 to mm.</td>
</tr>
<tr>
<td>NP REX.W + 0F 6E /r MOVQ mm, r/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>MMX</td>
<td>Move quadword from r/m64 to mm.</td>
</tr>
<tr>
<td>NP 0F 7E /r MOVQ r/m64, mm</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Move doubleword from mm to r/m32.</td>
</tr>
<tr>
<td>NP REX.W + 0F 7E /r MOVQ r/m64, mm</td>
<td>B</td>
<td>V/N.E.</td>
<td>MMX</td>
<td>Move doubleword from mm to r/m64.</td>
</tr>
<tr>
<td>66 0F 6E /r MOVQ xmm, r/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move doubleword from r/m32 to xmml.</td>
</tr>
<tr>
<td>66 REX.W 0F 6E /r MOVQ xmm, r/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Move quadword from r/m64 to xmml.</td>
</tr>
<tr>
<td>66 0F 7E /r MOVQ r/m32, xmm</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move doubleword from xmm register to r/m32.</td>
</tr>
<tr>
<td>66 REX.W 0F 7E /r MOVQ r/m64, xmm</td>
<td>B</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Move quadword from xmm register to r/m64.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move doubleword from r/m32 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Move quadword from r/m64 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move doubleword from xmm1 register to r/m32.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Move quadword from xmm1 register to r/m64.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move doubleword from r/m32 to xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64</td>
<td>C</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Move quadword from r/m64 to xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move doubleword from xmm1 register to r/m32.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1</td>
<td>D</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Move quadword from xmm1 register to r/m64.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**MOVD/Q with XMM destination:**

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

**MOVD/Q with 32/64 reg/mem destination:**

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VMOVQ or VMOVQ is encoded with VEX.L=1, an attempt to execute the instruction encoded with VEX.L=1 will cause an #UD exception.

**Operation**

**MOVD (when destination operand is MMX technology register)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC}; \\
\text{DEST}[63:32] & \leftarrow 00000000H;
\end{align*}
\]

**MOVD (when destination operand is XMM register)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC}; \\
\text{DEST}[127:32] & \leftarrow 0000000000000000000000000H; \\
\text{DEST}[\text{MAXVL}-1:128] & \text{Unmodified}
\end{align*}
\]
MOVDD (when source operand is MMX technology or XMM register)
DEST ← SRC[31:0];

VMOVD (VEX-encoded version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] ← 0

MOVQ (when destination operand is XMM register)
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
DEST[MAXVL-1:128] (Unmodified)

MOVQ (when destination operand is r/m64)
DEST[63:0] ← SRC[63:0];

MOVQ (when source operand is XMM register or r/m64)
DEST ← SRC[63:0];

VMOVD (EVEX-encoded version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] ← 0

VMOVQ (EVEX-encoded version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
MOVDD: __m64 _mm_cvtsi32_si64 (int i )
MOVDD: int _mm_cvtsi64_si32 (__m64m )
MOVDD: __m128i _mm_cvtsi32_si128 (int a)
MOVDD: int _mm_cvtsi128_si32 (__m128i a)
MOVQ: __int64 _mm_cvtsi128_si64(__m128i); 
MOVQ: __m128i _mm_cvtsi64_si128(__int64);
MOVQ: __m128i _mm_cvtsi64_si128(__int64);
VMOVQ: __m128i _mm_cvtsi128_si128(__m128i);
VMOVQ: _mm_loadl_epi64( __m128i * s);
VMOVQ: void _mm_storel_epi64( __m128i * d, __m128i s);

Flags Affected
None

SIMD Floating-Point Exceptions
None
**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD  If VEX.L = 1.

If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
### MOVDDUP—Replicate Double FP Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 12 /r MOVDDUP xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE3</td>
<td>Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.</td>
</tr>
<tr>
<td>VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.</td>
</tr>
<tr>
<td>VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 [k1]{z}, xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move double-precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 [k1]{z}, ymm2/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move even index double-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 [k1]{z}, zmm2/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move even index double-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>MOVDDUP</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

For 256-bit or higher versions: Duplicates even-indexed double-precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double-precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

**VMOVDDUP (EVEX encoded versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]
\[
\begin{align*}
\text{TMP}_{\text{SRC}}[63:0] & \leftarrow \text{SRC}[63:0] \\
\text{TMP}_{\text{SRC}}[127:64] & \leftarrow \text{SRC}[63:0] \\
\text{IF } VL \geq 256 & \leftarrow \\
\text{TMP}_{\text{SRC}}[191:128] & \leftarrow \text{SRC}[191:128] \\
\text{TMP}_{\text{SRC}}[255:192] & \leftarrow \text{SRC}[191:128] \\
\text{IF } VL \geq 512 & \leftarrow \\
\text{TMP}_{\text{SRC}}[319:256] & \leftarrow \text{SRC}[319:256] \\
\text{TMP}_{\text{SRC}}[383:320] & \leftarrow \text{SRC}[319:256] \\
\text{TMP}_{\text{SRC}}[477:384] & \leftarrow \text{SRC}[477:384] \\
\text{TMP}_{\text{SRC}}[511:484] & \leftarrow \text{SRC}[477:384] \\
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 & \leftarrow \\
\text{i } \leftarrow j \times 64 & \leftarrow \\
\text{IF } k1[j] \text{ OR } \text{*no writemask*} & \leftarrow \\
\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{TMP}_{\text{SRC}}[i+63:i] & \leftarrow \\
\text{ELSE} & \leftarrow \\
\text{IF } \text{*merging-masking*} & \leftarrow \\
\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged*} & \leftarrow \\
\text{ELSE} & \leftarrow \\
\text{DEST}[i+63:i] \leftarrow 0 & \leftarrow \\
\text{FI} & \leftarrow \\
\text{FI} & \leftarrow \\
\text{ENDFOR} & \leftarrow \\
\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 & \leftarrow \\
\end{align*}
\]

**VMOVDDUP (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{SRC}[63:0] \\
\text{DEST}[127:64] & \leftarrow \text{SRC}[63:0] \\
\text{DEST}[191:128] & \leftarrow \text{SRC}[191:128] \\
\text{DEST}[255:192] & \leftarrow \text{SRC}[191:128] \\
\text{DEST}[\text{MAXVL}-1:256] & \leftarrow 0 \\
\end{align*}
\]

**VMOVDDUP (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{SRC}[63:0] \\
\text{DEST}[127:64] & \leftarrow \text{SRC}[63:0] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0 \\
\end{align*}
\]
MOVDDUP (128-bit Legacy SSE version)
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMOVDDUP _m512d _mm512_movedup_pd(_m512d a);
VMOVDDUP _m512d _mm512_mask_movedup_pd(_m512d s, _mmask8 k, _m512d a);
VMOVDDUP _m512d _mm512_maskz_movedup_pd(_mmask8 k, _m512d a);
VMOVDDUP _m256d _mm256_mask_movedup_pd(_m256d s, _mmask8 k, _m256d a);
VMOVDDUP _m256d _mm256_maskz_movedup_pd(_mmask8 k, _m256d a);
VMOVDDUP _m128d _mm128_mask_movedup_pd(_m128d s, _mmask8 k, _m128d a);
VMOVDDUP _m128d _mm128_maskz_movedup_pd(_mmask8 k, _m128d a);
MOVDDUP _m256d _mm256_movedup_pd (_m256d a);
MOVDDUP _m128d _mm128_movedup_pd (_m128d a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 5;
EVEX-encoded instruction, see Exceptions Type E5NF.
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
### MOVQAD, VMOVQAD32/64—Move Aligned Packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 6F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move aligned packed integer values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>MOVQAD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 7F /r</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move aligned packed integer values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>MOVQAD xmm2/m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.6F.WIG 6F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed integer values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VMOVQAD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.6F.WIG 7F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed integer values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VMOVQAD xmm2/m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.6F.WIG 6F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed integer values from ymm2/mem to ymm1.</td>
</tr>
<tr>
<td>VMOVQAD ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.6F.W0 6F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD32 [k1][z], xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.6F.W0 6F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD32 [k1][z], ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.6F.W0 6F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD32 [k1][z], zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.6F.W0 7F /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD32 [k1][z], xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.6F.W0 7F /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD32 [k1][z], ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.6F.W0 7F /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD32 [k1][z], zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.6F.W1 6F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned quadword integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD64 [k1][z], xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.6F.W1 6F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned quadword integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD64 [k1][z], ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.6F.W1 6F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD64 [k1][z], zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.6F.W1 7F /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD64 [k1][z], xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.6F.W1 7F /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD64 [k1][z], ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.6F.W1 7F /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>VMOVQAD64 [k1][z], zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.
Operation

**VMOVDQA32 (EVEX encoded versions, register-copy form)**

KL, VL = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE
        IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
        ELSE  DEST[i+31:i] ← 0 ; zeroing-masking
    FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQA32 (EVEX encoded versions, store-form)**

KL, VL = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
    FI
ENDFOR

**VMOVDQA32 (EVEX encoded versions, load-form)**

KL, VL = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE
        IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
        ELSE  DEST[i+31:i] ← 0 ; zeroing-masking
    FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
VMOVQDQA64 (EVEX encoded versions, register-copy form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
      IF *merging-masking*
        THEN *DEST[i+63:i] remains unchanged*
      ELSE DEST[i+63:i] ← 0 ; zeroing-masking
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVQDQA64 (EVEX encoded versions, store-form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;

VMOVQDQA64 (EVEX encoded versions, load-form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
      IF *merging-masking*
        THEN *DEST[i+63:i] remains unchanged*
      ELSE  DEST[i+63:i] ← 0 ; zeroing-masking
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVQDA (VEX.256 encoded version, load - and register copy)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0

VMOVQDA (VEX.256 encoded version, store-form)
DEST[255:0] ← SRC[255:0]

VMOVQDA (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0

VMOVQDA (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
(V)MOVDQA (128-bit store-form version)
DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent
VMOVQDA32 __m512i _mm512_load_epi32( void * sa);
VMOVQDA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVQDA32 __m512i _mm512_maskz_load_epi32(__mmask16 k, void * sa);
VMOVQDA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVQDA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVQDA32 __m256i _mm256_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVQDA32 __m256i _mm256_mask_load_epi32(__mmask8 k, void * sa);
VMOVQDA32 __m256i _mm256_maskz_load_epi32(__mmask8 k, void * sa);
VMOVQDA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVQDA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVQDA32 __m128i _mm128_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVQDA32 __m128i _mm128_mask_load_epi32(__mmask8 k, void * sa);
VMOVQDA32 __m128i _mm128_maskz_load_epi32(__mmask8 k, void * sa);
VMOVQDA32 void _mm128_store_epi32(void * d, __m128i a);
VMOVQDA32 void _mm128_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVQDA64 __m512i _mm512_load_epi64( void * sa);
VMOVQDA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVQDA64 __m512i _mm512_maskz_load_epi64(__mmask8 k, void * sa);
VMOVQDA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVQDA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVQDA64 __m256i _mm256_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVQDA64 __m256i _mm256_mask_load_epi64(__mmask8 k, void * sa);
VMOVQDA64 __m256i _mm256_maskz_load_epi64(__mmask8 k, void * sa);
VMOVQDA64 void _mm256_store_epi64(void * d, __m256i a);
VMOVQDA64 void _mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
MOVDQA void __m256i _mm256_load_si256(__m256i * p);
MOVDQA _mm256_load_si256(__m256i * p);
MOVDQA void _mm256_store_si256(__m256i * p);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1.
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
## MOVQ, MOVDDQ/16/32/64—Move Unaligned Packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 6F /r MOVQDQU xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move unaligned packed integer values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>F3 0F 7F /r MOVQDQU xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move unaligned packed integer values from xmm1 to xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:W0 6F /r VMOVQDQU xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:W0 7F /r VMOVQDQU xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from xmm1 to xmm2/m128.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:W0 6F /r VMOVQDQU ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from ymm2/m256 to ymm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:W0 7F /r VMOVQDQU ymm2/m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from ymm1 to ymm2/m256.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:W0 6F /r VMOVQDQU8 xmm1 {k1}{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W0 6F /r VMOVQDQU8 ymm1 {k1}{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W0 6F /r VMOVQDQU8 zmm1 {k1}{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:W0 7F /r VMOVQDQU8 xmm2/m128 {k1}{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W0 7F /r VMOVQDQU8 ymm2/m256 {k1}{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W0 7F /r VMOVQDQU8 zmm2/m512 {k1}{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:W1 6F /r VMOVQDQU16 xmm1 {k1}{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W1 6F /r VMOVQDQU16 ymm1 {k1}{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W1 6F /r VMOVQDQU16 zmm1 {k1}{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:W1 7F /r VMOVQDQU16 xmm2/m128 {k1}{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W1 7F /r VMOVQDQU16 ymm2/m256 {k1}{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W1 7F /r VMOVQDQU16 zmm2/m512 {k1}{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F:W0 6F /r VMOVQDQU32 xmm1 {k1}{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
### Instruction Set Reference, M-U

**MOVDQU, VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.256.F3.0F.6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F.7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Note: VEX.ffff and EVEX.ffff are reserved and must be 1111b otherwise instructions will #UD.

**EVEX encoded versions:**

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.
The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

**VEX.256 encoded version:**
Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. 

Bits (MAXVL-1:256) of the destination register are zeroed.

**128-bit versions:**
Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

**VEX.128 encoded version:** Bits (MAXVL-1:128) of the destination register are zeroed.

**Operation**

**VMOVDQU8 (EVEX encoded versions, register-copy form)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC[i+7:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE DEST[i+7:i] ← 0 ; zeroing-masking
  FI
ENDFOR;

**VMOVDQU16**

**VMOVDQU32**

**VMOVDQU64**
**INSTRUCTION SET REFERENCE, M-U**

**VMOVDQU8 (EVEX encoded versions, load-form)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC[i+7:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
      ELSE  DEST[i+7:i] ← 0 ; zeroing-masking
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQU16 (EVEX encoded versions, register-copy form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC[i+15:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
      ELSE  DEST[i+15:i] ← 0 ; zeroing-masking
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQU16 (EVEX encoded versions, store-form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC[i+15:i]
  ELSE *DEST[i+15:i] remains unchanged* ; merging-masking
    FL
ENDFOR;
VMOVQU16 (EVEX encoded versions, load-form)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC[i+15:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE DEST[i+15:i] ← 0; zeroing-masking
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVQ32 (EVEX encoded versions, register-copy form)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE DEST[i+31:i] ← 0; zeroing-masking
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVQ32 (EVEX encoded versions, store-form)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;
VMOVQDU32 (EVEX encoded versions, load-form)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE DEST[i+31:i] ← 0 ; zeroing-masking
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVQDU64 (EVEX encoded versions, register-copy form)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE DEST[i+63:i] ← 0 ; zeroing-masking
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVQDU64 (EVEX encoded versions, store-form)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
  ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
    FI;
ENDFOR;
VMOVQDU64 (EVEX encoded versions, load-form)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ← SRC[i+63:i]
   ELSE
      IF *merging-mask* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE DEST[i+63:i] ← 0 ; zeroing-masking
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVQDU (VEX.256 encoded version, load - and register copy)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0

VMOVQDU (VEX.256 encoded version, store-form)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0

VMOVQDU (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0

(V)MOVQDU (128-bit store-form version)
DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVQDU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVQDU16 __m512i _mm512_maskz_loadu_epi16( __mmask32 k, void * sa);
VMOVQDU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVQDU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVQDU16 __m256i _mm256_maskz_loadu_epi16( __mmask16 k, void * sa);
VMOVQDU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVQDU16 __m128i _mm128_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVQDU16 __m128i _mm128_maskz_loadu_epi16( __mmask8 k, void * sa);
VMOVQDU16 void _mm128_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVQDU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVQDU32 __m512i _mm512_maskz_loadu_epi32( __mmask16 k, void * sa);
VMOVQDU32 void _mm512_mask_storeu_epi32(void * d, __mmask16 k, __m512i a);
VMOVQDU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVQDU32 __m256i _mm256_maskz_loadu_epi32( __mmask8 k, void * sa);
VMOVQDU32 void _mm256_mask_storeu_epi32(void * d, __mmask8 k, __m256i a);
VMOVQDU32 __m128i _mm128_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);
VMOVQDU32 __m128i _mm128_maskz_loadu_epi32( __mmask8 k, void * sa);
INSTRUCTION SET REFERENCE, M-U

VMOVQDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVQDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVQDQU64 __m512i _mm512_loadu_epi64( void * sa);
VMOVQDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVQDQU64 __m512i _mm512_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVQDQU64 void _mm512_storeu_epi64( void * d, __m512i a);
VMOVQDQU64 void _mm512_mask_storeu_epi64( void * d, __mmask8 k, __m512i a);
VMOVQDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVQDQU64 __m256i _mm256_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVQDQU64 void _mm256_storeu_epi64( void * d, __m256i a);
VMOVQDQU64 void _mm256_mask_storeu_epi64( void * d, __mmask8 k, __m256i a);
VMOVQDQU64 __m128i _mm128_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVQDQU64 __m128i _mm128_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVQDQU64 void _mm128_storeu_epi64( void * d, __m128i a);
VMOVQDQU64 void _mm128_mask_storeu_epi64( void * d, __mmask8 k, __m128i a);
VMOVQDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVQDQU8 __m512i _mm512_maskz_loadu_epi8(__mmask64 k, void * sa);
VMOVQDQU8 void _mm512_mask_storeu_epi8(__mmask64 k, __m512i a);
VMOVQDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVQDQU8 __m256i _mm256_maskz_loadu_epi8(__mmask32 k, void * sa);
VMOVQDQU8 void _mm256_mask_storeu_epi8(__mmask32 k, __m256i a);
VMOVQDQU8 __m128i _mm128_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVQDQU8 __m128i _mm128_maskz_loadu_epi8(__mmask16 k, void * sa);
VMOVQDQU8 void _mm128_mask_storeu_epi8(__mmask16 k, __m128i a);
MOVQDQU _m256i _mm256_loadu_si256(__m256i * p);
MOVQDQU _mm256_storeu_si256(__m256i * p);
MOVQDQU _m128i _mm128_loadu_si128(__m128i * p);
MOVQDQU _mm128_storeu_si128(__m128i * p);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4.nb.
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
MOVQ—Move Quadword from XMM to MMX Technology Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leaf Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F D6 /r</td>
<td>MOVQ mm, xmm</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move low quadword from xmm to mmx register.</td>
</tr>
</tbody>
</table>

**Description**

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST ← SRC[63:0];

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVQ: __m64 _mm_movepi64_pi64 (__m128i a)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM If CR0.TS[bit 3] = 1.

#UD If CR0.EM[bit 2] = 1.

If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE2[bit 26] = 0.

If the LOCK prefix is used.

#MF If there is a pending x87 FPU exception.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 12 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 12 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.WO 12 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op / En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

1. ModRM.MOD = 011B required

**Description**

This instruction cannot be used for memory to register moves.

**128-bit two-argument form:**

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

**128-bit and EVEX three-argument form**

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

**Operation**

MOVHLPS (128-bit two-argument form)

DEST[63:0] ← SRC[127:64]
DEST[MAXVL-1:64] (Unmodified)

VMOVHLPS (128-bit three-argument form - VEX & EVEX)

DEST[63:0] ← SRC2[127:64]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

**SIMD Floating-Point Exceptions**

None

---

1. ModRM.MOD = 011B required
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 7; additionally
#UD
   If VEX.L = 1.
EVEX-encoded instruction, see Exceptions Type E7NM.128.
MOVHPD—Move High Packed Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 16 /r MOVHPD xmm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move double-precision floating-point value from m64 to high quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge double-precision floating-point value from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge double-precision floating-point value from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>66 0F 17 /r MOVHPD m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move double-precision floating-point value from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move double-precision floating-point value from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move double-precision floating-point value from high quadword of xmm1 to m64.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>NA</td>
<td>ModRMreg/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64-bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.
**Operation**

MOVHPD (128-bit Legacy SSE load)
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

VMOVHPD (VEX.128 & EVEX encoded load)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] ← 0

VMOVHPD (store)
DEST[63:0] ← SRC[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)
MOVHPD void _mm_storeh_pd (double *p, __m128d a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally
#UD If VEX.L = 1.
EVEX-encoded instruction, see Exceptions Type E9NF.
**MOVHPS—Move High Packed Single-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 16 /r MOVHPS xmm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single-precision floating-point values from m64 to high quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>NP 0F 17 /r MOVHPS m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single-precision floating-point values from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move two packed single-precision floating-point values from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move two packed single-precision floating-point values from high quadword of xmm1 to m64.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>Tuple2</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction cannot be used for register to register or memory to memory moves.

**128-bit Legacy SSE load:**

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64-bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

**VEX.128 & EVEX encoded load:**

Loads two single-precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

**128-bit store:**

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing OF 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD. If VMOVHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.
Operation

MOVHPS (128-bit Legacy SSE load)
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

VMOVHPS (VEX.128 and EVEX encoded load)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] ← 0

VMOVHPS (store)
DEST[63:0] ← SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128 _mm_loadh_pi ( __m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi ( __m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally
#UD If VEX.L = 1.
EVEX-encoded instruction, see Exceptions Type E9NF.
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 16 /r , xmm1, xmm2</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 16 /r , xmm1, xmm2, xmm3</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 16 /r , xmm1, xmm2, xmm3</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.</td>
</tr>
</tbody>
</table>

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

**MOVLHPS (128-bit two-argument form)**

DEST[63:0] (Unmodified)

DEST[127:64] ← SRC[63:0]

DEST[MAXVL-1:128] (Unmodified)

**VMOVLHPS (128-bit three-argument form - VEX & EVEX)**

DEST[63:0] ← SRC1[63:0]

DEST[127:64] ← SRC2[63:0]

DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

SCREEN_FUC

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 7; additionally
#UD  If VEX.L = 1.
EVEX-encoded instruction, see Exceptions Type E7NM.128.
MOVLPD—Move Low Packed Double-Precision Floating-Point Value

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (r)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

This instruction cannot be used for register to register or memory to memory moves.

**128-bit Legacy SSE load:**

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64-bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

**VEX.128 & EVEX encoded load:**

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

**128-bit store:**

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13/r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'= 1 will cause an #UD exception.

### Operation

**MOVLPD (128-bit Legacy SSE load)**

DEST[63:0] ← SRC[63:0]

DEST[MAXVL-1:64] (Unmodified)
VMOVLPD (VEX.128 & EVEX encoded load)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

VMOVLPD (store)
DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent
MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)
MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 5; additionally
#UD If VEX.L = 1.
EVEX-encoded instruction, see Exceptions Type E9NF.
**MOVLPNS—Move Low Packed Single-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 12 /r MOVLPS xmm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single-precision floating-point values from m64 to low quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 12 /r VMOVLPNS xmm2, xmm1, m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 12 /r VMOVLPNS xmm2, xmm1, m64</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.</td>
</tr>
<tr>
<td>0F 13/r MOVLPS m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single-precision floating-point values from low quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 13/r VMOVLPNS m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move two packed single-precision floating-point values from low quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 13/r VMOVLPNS m64, xmm1</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move two packed single-precision floating-point values from low quadword of xmm1 to m64.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>Tuple2</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction cannot be used for register to register or memory to memory moves.

**128-bit Legacy SSE load:**

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64-bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

**VEX.128 & EVEX encoded load:**

Loads two packed single-precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

**128-bit store:**

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPNS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPNS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPNS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.
Operation

MOVLPS (128-bit Legacy SSE load)
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

VMOVLPS (VEX.128 & EVEX encoded load)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

VMOVLPS (store)
DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent
MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 5; additionally
#UD           If VEX.L = 1.
EVEX-encoded instruction, see Exceptions Type E9NF.
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 50 /r MOVMSKPD reg, xmm</td>
<td>RM V/V</td>
<td>SSE2</td>
<td>Extract 2-bit sign mask from xmm and store in reg. The upper bits of r32 or r64 are filled with zeros.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F:WIG 50 /r VMOVMSKPD reg, xmm2</td>
<td>RM V/V</td>
<td>AVX</td>
<td>Extract 2-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F:WIG 50 /r VMOVMSKPD reg, ymm2</td>
<td>RM V/V</td>
<td>AVX</td>
<td>Extract 4-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)MOVMSKPD (128-bit versions)
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
IF DEST = r32
   THEN DEST[31:2] ← 0;
   ELSE DEST[63:2] ← 0;
FI

VMOVMSKPD (VEX.256 encoded version)
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
DEST[2] ← SRC[191]
DEST[3] ← SRC[255]
IF DEST = r32
   THEN DEST[31:4] ← 0;
   ELSE DEST[63:4] ← 0;
FI
Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD: int _mm_movemask_pd (__m128d a)
VMOVMSKPD: _mm256_movemask_pd(__m256d a)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 7; additionally
#UD If VEX.vvv ≠ 1111B.
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

<table>
<thead>
<tr>
<th>Opcode/Description</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 50 /r MOVMSKPS reg, xmm</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Extract 4-bit sign mask from xmm and store in reg. The upper bits of r32 or r64 are filled with zeros.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 50 /r VMOVMSKPS reg, xmm2</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 4-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 50 /r VMOVMSKPS reg, ymm2</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>RM</td>
</tr>
</tbody>
</table>

Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];

IF DEST = r32
   THEN DEST[31:4] ← ZeroExtend;
ELSE DEST[63:4] ← ZeroExtend;
FI;

¹. ModRM.MOD = 011B required
(V)MOVMSKPS (128-bit version)
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
  THEN DEST[31:4] ← 0;
  ELSE DEST[63:4] ← 0;
FI

VMOVMSKPS (VEX.256 encoded version)
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
  THEN DEST[31:8] ← 0;
  ELSE DEST[63:8] ← 0;
FI

Intel C/C++ Compiler Intrinsic Equivalent
int _mm_movemask_ps(__m128 a)
int _mm256_movemask_ps(__m256 a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 7; additionally
#UD If VEX.vvvv ≠ 1111B.
MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

**Description**

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the non-temporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor’s implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementa-

---

1. ModRM.MOD = 011B required
tion may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.
The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.
The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

**Operation**

**MOVNTDQA (128bit- Legacy SSE form)**

\[ \text{DEST} \leftarrow \text{SRC} \]
\[ \text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)} \]

**VMOVNTDQA (VEX.128 and EVEX.128 encoded form)**

\[ \text{DEST} \leftarrow \text{SRC} \]
\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

**VMOVNTDQA (VEX.256 and EVEX.256 encoded forms)**

\[ \text{DEST}[255:0] \leftarrow \text{SRC}[255:0] \]
\[ \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \]

**VMOVNTDQA (EVEX.512 encoded form)**

\[ \text{DEST}[511:0] \leftarrow \text{SRC}[511:0] \]
\[ \text{DEST}[\text{MAXVL}-1:512] \leftarrow 0 \]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[ \text{VMOVNTDQA} \_\_m512\_\_mm512\_stream\_load\_si512(\_\_m512\_ const* \_p); \]
\[ \text{MOVNTDQA} \_\_m128i\_\_mm\_stream\_load\_si128 (\_\_m128i const* \_p); \]
\[ \text{VMOVNTDQA} \_\_m256i\_\_mm256\_stream\_load\_si256 (\_\_m256i const* \_p); \]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1;
EVEX-encoded instruction, see Exceptions Type E1NF.

#UD \[ \text{If VEX.vvvv} \neq 1111B \text{ or EVEX.vvvv} \neq 1111B. \]
MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F E7 /r MOVNTDQ m128, xmm1</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move packed integer values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1G E7 /r VMOVNTDQ m128, xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed integer values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VEX.256.66.0F:W1G E7 /r VMOVNTDQ m256, ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed integer values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W0 E7 /r VMOVNTDQ m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed integer values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W0 E7 /r VMOVNTDQ m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed integer values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W0 E7 /r VMOVNTDQ m512, zmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move packed integer values in zmm1 to m512 using non-temporal hint.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTDQ(EVEX encoded versions)

VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAXVL-1:VL] ← 0

1. ModRM.MOD = 011B required
MOVNTDQ (Legacy and VEX versions)
DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent
VMOVNTDQ void _mm512_stream_si512(void * p, __m512i a);
VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);
MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1NF.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
MOVNTI—Store Doubleword Using Non-Temporal Hint

**Description**
Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST ← SRC;

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVNTI: void _mm_stream_si32 (int *p, int a)

MOVNTI: void _mm_stream_si64(__int64 *p, __int64 a)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0) For an illegal address in the SS segment.
#PF(fault-code) For a page fault.
#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
   If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.
#PF(fault-code) For a page fault.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
   If the LOCK prefix is used.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2B /r MOVNTPD m128, xmm1</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move packed double-precision values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed double-precision values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed double-precision values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed double-precision values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed double-precision values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move packed double-precision values in zmm1 to m512 using non-temporal hint.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTPD (EVEX encoded versions)

VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAXVL-1:VL] ← 0

1. ModRM.MOD = 011B required
MOVNTPD (Legacy and VEX versions)

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void _mm512_stream_pd(double * p, __m512d a);
VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);
MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1NF.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

### Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VMOVNTPS (EVEX encoded versions)**

\[VL = 128, 256, 512\]

\[DEST[VL-1:0] \leftarrow SRC[VL-1:0]\]

\[DEST[MAXVL-1:VL] \leftarrow 0\]

---

1. ModRM.MOD = 011B required
MOVNTPS
DEST ← SRC

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);
MOVNTPS void _mm_stream_ps (float * p, __m128d a);
VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally
EVEX-encoded instruction, see Exceptions Type E1NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
MOVNTQ—Store of Quadword Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E7 7r</td>
<td>MOVNTQ m64, mm</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move quadword from mm to m64 using non-temporal hint.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ: void _mm_stream_pi(__m64 * p, __m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
MOVQ—Move Quadword

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 6F /r MOVQ mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Move quadword from mm/m64 to mm.</td>
</tr>
<tr>
<td>NP 0F 7F /r MOVQ mm/m64, mm</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Move quadword from mm to mm/m64.</td>
</tr>
<tr>
<td>F3 0F 7E /r MOVQ xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move quadword from xmm2/mem64 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F.WI 7E /r VMOVQ xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move quadword from xmm2 to xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 7E /r VMOVQ xmm1, xmm2/m64</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move quadword from xmm2/m64 to xmm1.</td>
</tr>
<tr>
<td>66 0F D6 /r MOVQ xmm2/m64, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move quadword from xmm1 to xmm2/mem64.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WI D6 /r VMOVQ xmm1/m64, xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move quadword from xmm2 register to xmm1/m64.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move quadword from xmm2 register to xmm1/m64.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.
**Operation**

MOVQ instruction when operating on MMX technology registers and memory locations

\[ \text{DEST} \leftarrow \text{SRC}; \]

MOVQ instruction when source and destination operands are XMM registers

\[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0]; \]
\[ \text{DEST}[127:64] \leftarrow 0000000000000000H; \]

MOVQ instruction when source operand is XMM register and destination operand is memory location:

\[ \text{DEST} \leftarrow \text{SRC}[63:0]; \]

MOVQ instruction when source operand is memory location and destination operand is XMM register:

\[ \text{DEST}[63:0] \leftarrow \text{SRC}; \]
\[ \text{DEST}[127:64] \leftarrow 0000000000000000H; \]

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination

\[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0] \]
\[ \text{DEST}[\text{MAXVL}-1:64] \leftarrow 0 \]

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination

\[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0] \]
\[ \text{DEST}[\text{MAXVL}-1:64] \leftarrow 0 \]

VMOVQ (7E - EVEX encoded version) with XMM register source and destination

\[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0] \]
\[ \text{DEST}[\text{MAXVL}-1:64] \leftarrow 0 \]

VMOVQ (D6 - EVEX encoded version) with XMM register source and destination

\[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0] \]
\[ \text{DEST}[\text{MAXVL}-1:64] \leftarrow 0 \]

VMOVQ (7E) with memory source

\[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0] \]
\[ \text{DEST}[\text{MAXVL}-1:64] \leftarrow 0 \]

VMOVQ (7E - EVEX encoded version) with memory source

\[ \text{DEST}[63:0] \leftarrow \text{SRC}[63:0] \]
\[ \text{DEST}[\text{MAXVL}-1:64] \leftarrow 0 \]

VMOVQ (D6) with memory dest

\[ \text{DEST}[63:0] \leftarrow \text{SRC2}[63:0] \]

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVQ __m128i _mm_loadu_si64( void * s);
VMOVQ void _mm_storeu_si64( void * d, __m128i s);
MOVQ m128i _mm_move_epi64(__m128i a)
SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F D6 /r</td>
<td>MOVQ2DQ xmm, mm</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move quadword from mmx to low quadword of xmm.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ: __m128i _mm_movpi64_epi64 (__m64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM If CR0.TS[bit 3] = 1.
#UD If CR0.EF[bit 2] = 1.
If CR4.OSFXSR[bit 9] = 0.
If CPUID.01H:EDX.SSE2[bit 26] = 0.
If the LOCK prefix is used.
#MF If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
### Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct location. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSB (byte move), MOVSW (word move), or MOVSD (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incre-
mented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

**NOTE**

To improve performance, more recent processors support modifications to the processor’s operation during the string store operations initiated with MOV$ and MOV$B. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for additional information on fast-string operation.

The MOV$, MOV$B, MOV$W, and MOV$D instructions can be preceded by the REP prefix (see "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction’s default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST ← SRC;

Non-64-bit Mode:

IF (Byte move)
THEN IF DF = 0
THEN
    (E)SI ← (E)SI + 1;
    (E)DI ← (E)DI + 1;
ELSE
    (E)SI ← (E)SI - 1;
    (E)DI ← (E)DI - 1;
FI;
ELSE IF (Word move)
THEN IF DF = 0
THEN
    (E)SI ← (E)SI + 2;
    (E)DI ← (E)DI + 2;
FI;
ELSE
    (E)SI ← (E)SI - 2;
    (E)DI ← (E)DI - 2;
FI;
ELSE IF (Doubleword move)
THEN IF DF = 0
THEN
    (E)SI ← (E)SI + 4;
    (E)DI ← (E)DI + 4;
FI;
ELSE
    (E)SI ← (E)SI - 4;
    (E)DI ← (E)DI - 4;
FI;
FI;

64-bit Mode:

IF (Byte move)
THEN IF DF = 0
THEN
(R|E)SI ← (R|E)SI + 1;
(R|E)DI ← (R|E)DI + 1;
ELSE
  (R|E)SI ← (R|E)SI - 1;
  (R|E)DI ← (R|E)DI - 1;
  FI;
ELSE IF (Word move)
  THEN IF DF = 0
    (R|E)SI ← (R|E)SI + 2;
    (R|E)DI ← (R|E)DI + 2;
    FI;
  ELSE
    (R|E)SI ← (R|E)SI - 2;
    (R|E)DI ← (R|E)DI - 2;
    FI;
  ELSE IF (Doubleword move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 4;
      (R|E)DI ← (R|E)DI + 4;
      FI;
    ELSE
      (R|E)SI ← (R|E)SI - 4;
      (R|E)DI ← (R|E)DI - 4;
      FI;
  ELSE IF (Quadword move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 8;
      (R|E)DI ← (R|E)DI + 8;
      FI;
    ELSE
      (R|E)SI ← (R|E)SI - 8;
      (R|E)DI ← (R|E)DI - 8;
      FI;
  FI;

Flags Affected
None

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
##MOVSD—Move or Merge Scalar Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 10 /r MOVSD xmm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move scalar double-precision floating-point value from xmm2 to xmm1 register.</td>
</tr>
<tr>
<td>F2 0F 10 /r MOVSD xmm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Load scalar double-precision floating-point value from m64 to xmm1 register.</td>
</tr>
<tr>
<td>F2 0F 11 /r MOVSD xmm1/m64, xmm2</td>
<td>C</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move scalar double-precision floating-point value from xmm2 register to xmm1/m64.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F.W0 10 /r VMOVSD xmm1, xmm2, xmm3</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W0 10 /r VMOVSD xmm1, m64</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Load scalar double-precision floating-point value from m64 to xmm1 register.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F.W0 11 /r VMOVSD xmm1, xmm2, xmm3</td>
<td>E</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W0 11 /r VMOVSD m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Store scalar double-precision floating-point value from xmm1 register to m64.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W1 10 /r VMOVSD xmm1[k1]{z}, xmm2, xmm3</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 10 /r VMOVSD xmm1[k1]{z}, m64</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Load scalar double-precision floating-point value from m64 to xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W1 11 /r VMOVSD xmm1[k1]{z}, xmm2, xmm3</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 11 /r VMOVSD m64[k1], xmm1</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Store scalar double-precision floating-point value from xmm1 register to m64 under writemask k1.</td>
</tr>
</tbody>
</table>

###Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMsreg(r, w)</td>
<td>ModRMsr/m(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMsreg(w)</td>
<td>VEX.vvvv(r)</td>
<td>ModRMsr/m(r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>NA</td>
<td>ModRMsr/m(w)</td>
<td>ModRMsreg(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>NA</td>
<td>ModRMsreg(w)</td>
<td>ModRMsr/m(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>NA</td>
<td>ModRMsr/m(w)</td>
<td>vvvv(r)</td>
<td>ModRMsreg(r)</td>
<td>NA</td>
</tr>
<tr>
<td>F</td>
<td>Tuple1 Scalar</td>
<td>ModRMsreg(r, w)</td>
<td>ModRMsr/m(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>G</td>
<td>Tuple1 Scalar</td>
<td>ModRMsr/m(w)</td>
<td>ModRMsreg(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.
Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.
VEX and EVEX encoded register-register syntax: Moves a scalar double-precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.
VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.
EVEX encoded versions: The low quadword of the destination is updated according to the writemask.
Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

Operation
VMOVSD (EVEX.NDS.LIG.F2.0F 10 /r: VMOVSD xmm1, m64 with support for 32 registers)
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[MAXVL-1:64] ← 0

VMOVSD (EVEX.NDS.LIG.F2.0F 11 /r: VMOVSD m64, xmm1 with support for 32 registers)
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC[63:0]
    ELSE *DEST[63:0] remains unchanged* ; merging-masking
    FI;

VMOVSD (EVEX.NDS.LIG.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)
DEST[63:0] ← SRC[63:0]

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← 0
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VMOVSD __m128d _mm_mask_load_sd(__m128d s, __mmask8 k, double * p);
VMOVSD __m128d _mm_maskz_load_sd( __mmask8 k, double * p);
VMOVSD __m128d _mm_mask_move_sd(__m128d sh, __mmask8 k, __m128d sl, __m128d a);
VMOVSD __m128d _mm_maskz_move_sd( __mmask8 k, __m128d s, __m128d a);
VMOVSD void _mm_mask_store_sd(double * p, __mmask8 k, __m128d s);
MOVSD __m128d __m128d:mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d __m128d:mm_move_sd ( __m128d a, __m128d b)

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 5; additionally
#UD If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E10.
MOVSHDUP—Replicate Single FP Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 16 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE3</td>
<td>Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:W1G 16 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:W1G 16 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F:W0 16 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move odd index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F:W0 16 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move odd index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F:W0 16 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move odd index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Duplicates odd-indexed single-precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

![Figure 4-3. MOVSHDUP Operation]
Operation

**VMOVSHDUP (EVEX encoded versions)**

\((K_L, V_L) = (4, 128), (8, 256), (16, 512)\)

\[
\begin{align*}
\text{TMP}_\text{SRC}[31:0] & \leftarrow \text{SRC}[63:32] \\
\text{TMP}_\text{SRC}[63:32] & \leftarrow \text{SRC}[63:32] \\
\text{TMP}_\text{SRC}[95:64] & \leftarrow \text{SRC}[127:96] \\
\text{TMP}_\text{SRC}[127:96] & \leftarrow \text{SRC}[127:96]
\end{align*}
\]

**If** \(V_L \geq 256\)

\[
\begin{align*}
\text{TMP}_\text{SRC}[159:128] & \leftarrow \text{SRC}[191:160] \\
\text{TMP}_\text{SRC}[191:160] & \leftarrow \text{SRC}[191:160] \\
\text{TMP}_\text{SRC}[223:192] & \leftarrow \text{SRC}[255:224] \\
\text{TMP}_\text{SRC}[255:224] & \leftarrow \text{SRC}[255:224]
\end{align*}
\]

**If** \(V_L \geq 512\)

\[
\begin{align*}
\text{TMP}_\text{SRC}[287:256] & \leftarrow \text{SRC}[319:288] \\
\text{TMP}_\text{SRC}[319:288] & \leftarrow \text{SRC}[319:288] \\
\text{TMP}_\text{SRC}[351:320] & \leftarrow \text{SRC}[383:352] \\
\text{TMP}_\text{SRC}[383:352] & \leftarrow \text{SRC}[383:352] \\
\text{TMP}_\text{SRC}[415:384] & \leftarrow \text{SRC}[447:416] \\
\text{TMP}_\text{SRC}[447:416] & \leftarrow \text{SRC}[447:416] \\
\text{TMP}_\text{SRC}[479:448] & \leftarrow \text{SRC}[511:480] \\
\text{TMP}_\text{SRC}[511:480] & \leftarrow \text{SRC}[511:480]
\end{align*}
\]

**For** \(j \leftarrow 0\) **To** \(K_L - 1\)

\[
\begin{align*}
i & \leftarrow j * 32 \\
\text{IF} \ k1[j] \text{ OR } \text{*no writemask*} \\
\text{THEN} \ & \text{DEST}[i+31:i] \leftarrow \text{TMP}_\text{SRC}[i+31:i] \\
\text{ELSE} \\
\text{IF} \ & \text{*merging-masking*} \\
\text{THEN} \ & \text{DEST}[i+31:i] \text{ remains unchanged*} \\
\text{ELSE} \\
\text{DEST}[i+31:i] & \leftarrow 0 \\
\text{FI} \\
\text{FI}
\end{align*}
\]

\[
\begin{align*}
\text{DEST}[\text{MAXVL}-1:256] & \leftarrow 0
\end{align*}
\]

**VMOVSHDUP (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC}[63:32] \\
\text{DEST}[63:32] & \leftarrow \text{SRC}[63:32] \\
\text{DEST}[95:64] & \leftarrow \text{SRC}[127:96] \\
\text{DEST}[127:96] & \leftarrow \text{SRC}[127:96] \\
\text{DEST}[159:128] & \leftarrow \text{SRC}[191:160] \\
\text{DEST}[191:160] & \leftarrow \text{SRC}[191:160] \\
\text{DEST}[223:192] & \leftarrow \text{SRC}[255:224] \\
\text{DEST}[255:224] & \leftarrow \text{SRC}[255:224] \\
\text{DEST}[\text{MAXVL}-1:256] & \leftarrow 0
\end{align*}
\]

**VMOVSHDUP (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow \text{SRC}[63:32] \\
\text{DEST}[63:32] & \leftarrow \text{SRC}[63:32] \\
\text{DEST}[95:64] & \leftarrow \text{SRC}[127:96] \\
\text{DEST}[127:96] & \leftarrow \text{SRC}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0
\end{align*}
\]
MOVSHDUP (128-bit Legacy SSE version)
DEST[31:0] \leftarrow SRC[63:32]
DEST[63:32] \leftarrow SRC[63:32]
DEST[95:64] \leftarrow SRC[127:96]
DEST[127:96] \leftarrow SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMOVSHDUP _m512 _mm512_movehdup_ps(_m512 a);
VMOVSHDUP _m512 _mm512_mask_movehdup_ps(_m512 s, _mmask16 k, _m512 a);
VMOVSHDUP _m512 _mm512_maskz_movehdup_ps(_mmask16 k, _m512 a);
VMOVSHDUP _m256 _mm256_movehdup_ps(_m256 s, _mmask8 k, _m256 a);
VMOVSHDUP _m256 _mm256_mask_movehdup_ps(_mmask8 k, _m256 a);
VMOVSHDUP _m256 _mm256_maskz_movehdup_ps(_mmask8 k, _m256 a);
VMOVSHDUP _m128 _mm128_movehdup_ps(_m128 s, _mmask8 k, _m128 a);
VMOVSHDUP _m128 _mm128_mask_movehdup_ps(_mmask8 k, _m128 a);
VMOVSHDUP _m128 _mm128_maskz_movehdup_ps(_mmask8 k, _m128 a);
VMOVSHDUP _m256 _mm256_movehdup_ps(_m256 a);
VMOVSHDUP _m128 _mm128_movehdup_ps(_m128 a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
### MOVSLDUP—Replicate Single FP Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE3</td>
<td>Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:WIG 12 /r VMOVSLDUP xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:WIG 12 /r VMOVSLDUP ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F:W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move even index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F:W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move even index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F:W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move even index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Duplicates even-indexed single-precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

---

**Figure 4-4. MOVSLDUP Operation**
Operation

VMOVSLDUP (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP_SRC[31:0] ← SRC[31:0]
TMP_SRC[63:32] ← SRC[31:0]
TMP_SRC[95:64] ← SRC[95:64]
TMP_SRC[127:96] ← SRC[95:64]

IF VL >= 256
    TMP_SRC[159:128] ← SRC[159:128]
FI;

IF VL >= 512
    TMP_SRC[287:256] ← SRC[287:256]
    TMP_SRC[319:288] ← SRC[287:256]
    TMP_SRC[351:320] ← SRC[351:320]
    TMP_SRC[479:448] ← SRC[479:448]
    TMP_SRC[511:480] ← SRC[479:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_SRC[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

VMOVSLDUP (VEX.256 encoded version)

DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[159:128] ← SRC[159:128]
DEST[MAXVL-1:256] ← 0

VMOVSLDUP (VEX.128 encoded version)

DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[MAXVL-1:128] ← 0
MOVSLDUP (128-bit Legacy SSE version)
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMOVSLDUP __m512 __m512_moveldup_ps( __m512 a);
VMOVSLDUP __m512 __mm512_mask_moveldup_ps( __m512 s, __mmask16 k, __m512 a);
VMOVSLDUP __m512 __mm512_maskz_moveldup_ps( __mmask16 k, __m512 a);
VMOVSLDUP __m256 __mm256_mask_moveldup_ps( __m256 s, __mmask8 k, __m256 a);
VMOVSLDUP __m256 __mm256_maskz_moveldup_ps( __mmask8 k, __m256 a);
VMOVSLDUP __m128 __mm128_mask_moveldup_ps( __m128 s, __mmask8 k, __m128 a);
VMOVSLDUP __m128 __mm128_maskz_moveldup_ps( __mmask8 k, __m128 a);
VMOVSLDUP __m256 __mm256_moveldup_ps( __m256 a);
VMOVSLDUP __m128 __mm128_moveldup_ps( __m128 a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
## MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Merge scalar single-precision floating-point value from xmm2 to xmm1 register.</td>
</tr>
<tr>
<td>MOVSS xmm1, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Load scalar single-precision floating-point value from m32 to xmm1 register.</td>
</tr>
<tr>
<td>MOVSS xmm1, m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F:W1G 10 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register.</td>
</tr>
<tr>
<td>VMOVSS xmm1, xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F3.0F:W1G 10 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Load scalar single-precision floating-point value from m32 to xmm1 register.</td>
</tr>
<tr>
<td>VMOVSS xmm1, m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F 11 /r</td>
<td>C</td>
<td>V/V</td>
<td>SSE</td>
<td>Move scalar single-precision floating-point value from xmm1 register to xmm2/m32.</td>
</tr>
<tr>
<td>MOVSS xmm2/m32, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F:W1G 11 /r</td>
<td>E</td>
<td>V/V</td>
<td>AVX</td>
<td>Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register.</td>
</tr>
<tr>
<td>VMOVSS xmm1, xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F3.0F:W1G 11 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move scalar single-precision floating-point value from xmm1 register to xmm3/m32.</td>
</tr>
<tr>
<td>VMOVSS xmm1, m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 10 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>VMOVSS xmm1 {k1}{z}, xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F:W0 10 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single-precision floating-point values from m32 to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VMOVSS xmm1 {k1}{z}, m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 11 /r</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>VMOVSS xmm1 {k1}{z}, xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F:W0 11 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single-precision floating-point values from xmm1 to m32 under writemask k1.</td>
</tr>
<tr>
<td>VMOVSS m32 {k1}, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
</tr>
<tr>
<td>F</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>G</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single-precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction “VMOVSS m32, xmm1”, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction “VMOVSS mv \{k1\}, xmm1”, EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMOVSS (EVEX.NDS.LIG.F3.0F.W0 11 /r when the source operand is memory and the destination is an XMM register)
IF k1[0] or *no writemask*
   THEN DEST[31:0] ← SRC[31:0]
ELSE
   IF *merging-masking*  ; merging-masking
      THEN *DEST[31:0] remains unchanged*
   ELSE  ; zeroing-masking
      THEN DEST[31:0] ← 0
   FI;
FI;
DEST[MAXVL-1:32] ← 0

VMOVSS (EVEX.NDS.LIG.F3.0F.W0 10 /r when the source operand is an XMM register and the destination is memory)
IF k1[0] or *no writemask*
   THEN DEST[31:0] ← SRC[31:0]
ELSE  *DEST[31:0] remains unchanged*  ; merging-masking
FI;
INSTRUCTION SET REFERENCE, M-U

VMOVSS (EVEX.NDS.LIG.F3.0F.W0 10/11 /r where the source and destination are XMM registers)

IF k1[0] or *no writemask*
THEN DEST[31:0] ← SRC2[31:0]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[31:0] remains unchanged*
ELSE ; zeroing-masking
THEN DEST[31:0] ← 0
FI;

DEST[128:128] ← 0

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)

DEST[31:0] ← SRC[31:0]
DEST[128:128] ← 0

VMOVSS (EVEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)

DEST[31:0] ← SRC[31:0]
DEST[128:128] ← 0

VMOVSS (EVEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)

DEST[31:0] ← SRC[31:0]
DEST[128:128] ← 0

VMOVSS (EVEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)

DEST[31:0] ← SRC[31:0]
DEST[128:128] ← 0

MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)

DEST[31:0] ← SRC[31:0]

MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)

DEST[31:0] ← SRC[31:0]
DEST[128:128] ← 0
DEST[128:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSS __m128 _mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 _mm_maskz_load_ss( __mmask8 k, float * p);
VMOVSS __m128 _mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 _mm_maskz_move_ss( __mmask8 k, __m128 s, __m128 a);
VMOVSS void _mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 _mm_load_ss(float * p)
MOVSS void _mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions
None
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 5; additionally
#UD  If VEX.vvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E10.
## MOVSX/MOVSD—Move with Sign-Extension

### Description
Covers the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation
\[
\text{DEST} \leftarrow \text{SignExtend} (\text{SRC})
\]

### Flags Affected
None.

### Protected Mode Exceptions
- **#GP(0)**
  - If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)**
  - If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**
  - If a page fault occurs.
- **#AC(0)**
  - If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**
  - If the LOCK prefix is used.

### Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode/En</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F BE /r</td>
<td>MOVSX r16, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to word with sign-extension.</td>
</tr>
<tr>
<td>0F BE /r</td>
<td>MOVSX r32, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to doubleword with sign-extension.</td>
</tr>
<tr>
<td>REX.W + 0F BE /r</td>
<td>MOVSX r64, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move byte to quadword with sign-extension.</td>
</tr>
<tr>
<td>0F BF /r</td>
<td>MOVSX r32, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move word to doubleword with sign-extension.</td>
</tr>
<tr>
<td>REX.W + 0F BF /r</td>
<td>MOVSX r64, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move word to quadword with sign-extension.</td>
</tr>
<tr>
<td>63 /r*</td>
<td>MOVSXD r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move doubleword to doubleword with sign-extension.</td>
</tr>
<tr>
<td>63 /r*</td>
<td>MOVSXD r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move doubleword to quadword with sign-extension.</td>
</tr>
</tbody>
</table>

**NOTES:**
- The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.
**MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>66 0F 11 /r</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.66.0F:WIG 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:WIG 11 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F:WIG 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed double-precision floating-point from ymm2/m256 to ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F:WIG 11 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed double-precision floating-point from ymm1 to ymm2/m256.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 10 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed double-precision floating-point from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 11 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed double-precision floating-point from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 10 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed double-precision floating-point from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 11 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed double-precision floating-point from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 10 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 11 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**EVEX.512 encoded version:**

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.
**VEX.256 encoded version:**
Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

**128-bit versions:**
Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

**VEX.128 and EVEX.128 encoded versions:** Bits (MAXVL-1:128) of the destination register are zeroed.

**Operation**

**VMOVUPD (EVEX encoded versions, register-copy form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] ← 0 ; zeroing-masking
    FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVUPD (EVEX encoded versions, store-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking

    FI;
ENDFOR;
VMOVUPD (EVEX encoded versions, load-form)

\( (KL, VL) = (2, 128), (4, 256), (8, 512) \)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)
   \( i \leftarrow j \times 64 \)
   IF \( k1[j] \) OR *no writemask*
      THEN \( \text{DEST}[i+63:i] \leftarrow \text{SRC}[i+63:i] \)
   ELSE
      IF *merging-masking*
         THEN *\( \text{DEST}[i+63:i] \) remains unchanged*
      ELSE \( \text{DEST}[i+63:i] \leftarrow 0 \); zeroing-masking
   FI
ENDFOR
\( \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \)

VMOVUPD (VEX.256 encoded version, load - and register copy)
\( \text{DEST}[255:0] \leftarrow \text{SRC}[255:0] \)
\( \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \)

VMOVUPD (VEX.256 encoded version, store-form)
\( \text{DEST}[255:0] \leftarrow \text{SRC}[255:0] \)

VMOVUPD (VEX.128 encoded version)
\( \text{DEST}[127:0] \leftarrow \text{SRC}[127:0] \)
\( \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \)

MOVUPD (128-bit load- and register-copy- form Legacy SSE version)
\( \text{DEST}[127:0] \leftarrow \text{SRC}[127:0] \)
\( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)

(V)MOVUPD (128-bit store-form version)
\( \text{DEST}[127:0] \leftarrow \text{SRC}[127:0] \)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPD __m512d _mm512_loadu_pd( void * s);
VMOVUPD __m512d _mm512_mask_loadu_pd(__m512d a, __mmask8 k, void * s);
VMOVUPD __m512d _mm512_maskz_loadu_pd( __mmask8 k, void * s);
VMOVUPD void _mm512_storeu_pd( void * d, __m512d a);
VMOVUPD void _mm512_mask_storeu_pd( void * d, __mmask8 k, __m512d a);
VMOVUPD __m256d _mm256_mask_loadu_pd(__m256d s, __mmask8 k, void * m);
VMOVUPD __m256d _mm256_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm256_mask_storeu_pd( void * d, __mmask8 k, __m256d a);
VMOVUPD __m128d _mm128_mask_loadu_pd( __m128d s, __mmask8 k, void * m);
VMOVUPD __m128d _mm128_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm128_mask_storeu_pd( void * d, __mmask8 k, __m128d a);
MOVUPD __m256d _mm256_loadu_pd( double * p);
MOVUPD void _mm256_storeu_pd( double *p, __m256d a);
MOVUPD __m128d _mm_loadu_pd (double * p);
MOVUPD void _mm_storeu_pd (double *p, __m128d a);

SIMD Floating-Point Exceptions

None
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
Note treatment of #AC varies; additionally
#UD If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E4.nb.
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 10 /r</td>
<td>A</td>
<td>V/V SSE</td>
<td></td>
<td>Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>MOVUPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 11 /r</td>
<td>B</td>
<td>V/V SSE</td>
<td></td>
<td>Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>MOVUPS xmm2/m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.0FWIG 10 /r</td>
<td>A</td>
<td>V/V AVX</td>
<td></td>
<td>Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VMOVUPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.0FWIG 11 /r</td>
<td>B</td>
<td>V/V AVX</td>
<td></td>
<td>Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VMOVUPS xmm2/m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.0FWIG 10 /r</td>
<td>A</td>
<td>V/V AVX</td>
<td></td>
<td>Move unaligned packed single-precision floating-point from ymm2/m256 to ymm1.</td>
</tr>
<tr>
<td>VMOVUPS ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.0FWIG 11 /r</td>
<td>B</td>
<td>V/V AVX</td>
<td></td>
<td>Move unaligned packed single-precision floating-point from ymm1 to ymm2/m256.</td>
</tr>
<tr>
<td>VMOVUPS ymm2/m256, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.0FW0 10 /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512F</td>
<td></td>
<td>Move unaligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVUPS xmm1 <a href="z">k1</a>, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0FW0 10 /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512F</td>
<td></td>
<td>Move unaligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVUPS ymm1 <a href="z">k1</a>, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0FW0 10 /r</td>
<td>C</td>
<td>V/V AVX512F</td>
<td></td>
<td>Move unaligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VMOVUPS zmm1 <a href="z">k1</a>, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.0FW0 11 /r</td>
<td>D</td>
<td>V/V AVX512VL AVX512F</td>
<td></td>
<td>Move unaligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>VMOVUPS xmm2/m128 <a href="z">k1</a>, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0FW0 11 /r</td>
<td>D</td>
<td>V/V AVX512VL AVX512F</td>
<td></td>
<td>Move unaligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>VMOVUPS ymm2/m256 <a href="z">k1</a>, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0FW0 11 /r</td>
<td>D</td>
<td>V/V AVX512F</td>
<td></td>
<td>Move unaligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>VMOVUPS zmm2/m512 <a href="z">k1</a>, zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**EVEX.512 encoded version:**

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.
**VEX.256 and EVEX.256 encoded versions:**
Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

**128-bit versions:**
Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

**VEX.128 and EVEX.128 encoded versions:** Bits (MAXVL-1:128) of the destination register are zeroed.

**Operation**

**VMOVUPS (EVEX encoded versions, register-copy form)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j \times 32\]

IF \(k1[j] \text{ OR } \text{*no writemask*}\)

THEN \(\text{DEST}[i+31:i] \leftarrow \text{SRC}[i+31:i]\)

ELSE

IF \text{*merging-masking*} ; merging-masking

THEN \text{*DEST}[i+31:i] remains unchanged* ; zeroing-masking

ELSE \(\text{DEST}[i+31:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

**VMOVUPS (EVEX encoded versions, store-form)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j \times 32\]

IF \(k1[j] \text{ OR } \text{*no writemask*}\)

THEN \(\text{DEST}[i+31:i] \leftarrow \text{SRC}[i+31:i]\)

ELSE \text{*DEST}[i+31:i] remains unchanged* ; merging-masking

FI;

ENDFOR;
VMOVUPS (EVEX encoded versions, load-form)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE  DEST[i+31:i] ← 0 ; zeroing-masking
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMOVUPS (VEX.256 encoded version, load - and register copy)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0

VMOVUPS (VEX.256 encoded version, store-form)
DEST[255:0] ← SRC[255:0]

VMOVUPS (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0

MOVUPS (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)

(V)MOVUPS (128-bit store-form version)
DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent
VMOVUPS __m512 __mm512_loadu_ps( void * s);
VMOVUPS __m512 __mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 __mm512_maskz_loadu_ps(__mmask16 k, void * s);
VMOVUPS void __mm512_storeu_ps( void * d, __m512 a);
VMOVUPS void __mm512_mask_storeu_ps(__mmask8 k, __m512 a);
VMOVUPS __m256 __mm256_mask_loadu_ps(__m256 a, __mmask8 k, void * s);
VMOVUPS __m256 __mm256_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void __mm256_mask_storeu_ps(__mmask8 k, __m256 a);
VMOVUPS __m128 __mm128_mask_loadu_ps(__m128 a, __mmask8 k, void * s);
VMOVUPS __m128 __mm128_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void __mm128_mask_storeu_ps(__mmask8 k, __m128 a);
MOVUPS __m256 __mm256_loadu_ps ( float * p);
MOVUPS __m256 __mm256_storeu_ps( float *p, __m256 a);
MOVUPS __m128 __mm_loadu_ps ( float * p);
MOVUPS void __mm_storeu_ps( float *p, __m128 a);

SIMD Floating-Point Exceptions
None
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
Note treatment of #AC varies;
EVEX-encoded instruction, see Exceptions Type E4.nb.
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
MOVZX—Move with Zero-Extend

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F B6 /r</td>
<td>MOVZX r16, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to word with zero-extension.</td>
</tr>
<tr>
<td>0F B6 /r</td>
<td>MOVZX r32, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to doubleword, zero-extension.</td>
</tr>
<tr>
<td>REX.W + 0F B6 /r</td>
<td>MOVZX r64, r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move byte to quadword, zero-extension.</td>
</tr>
<tr>
<td>0F B7 /r</td>
<td>MOVZX r32, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move word to doubleword, zero-extension.</td>
</tr>
<tr>
<td>REX.W + 0F B7 /r</td>
<td>MOVZX r64, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move word to quadword, zero-extension.</td>
</tr>
</tbody>
</table>

**NOTES:**
* In 64-bit mode, r/m8 cannot be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg(w)</td>
<td>ModRM:r/m(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**
Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**
DEST ← ZeroExtend(SRC);

**Flags Affected**
None.

**Protected Mode Exceptions**
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If the DS, ES, FS, or GS register contains a NULL segment selector.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**
- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
### MPSADBW — Compute Multiple Packed Sums of Absolute Difference

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 42 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPSADBW xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm1 and xmm2/m128 and writes the results in xmm1. Starting offsets within xmm1 and xmm2/m128 are determined by imm8.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.WIG 42 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMPSADBW xmm1, xmm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and xmm3/m128 and writes the results in xmm1. Starting offsets within xmm2 and xmm3/m128 are determined by imm8.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F3A.WIG 42 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMPSADBW ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in ymm2 and ymm3/m256 and writes the results in ymm1. Starting offsets within ymm2 and ymm3/m256 are determined by imm8.</td>
</tr>
</tbody>
</table>

#### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

#### Description

(V)MPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block_1 is selectable using a one bit select control, multiplied by 32 bits.

128-bit Legacy SSE version: Immm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Immm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Immm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Immm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Immm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Immm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.
The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L = 1, an attempt to execute the instruction encoded with VEX.L = 1 will cause an #UD exception.

Figure 4-5. 256-bit VMPSADBW Operation
VMPSADBW (VEX.256 encoded version)
BLK2_OFFSET ← imm8[1:0]*32
BLK1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ← SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]
SRC2_BYTE0 ← SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

BLK2_OFFSET ← imm8[4:3]*32 + 128
BLK1_OFFSET ← imm8[5]*32 + 128
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ← SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 ← SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[143:128] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[159:144] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[175:160] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
VMPSADBW (VEX.128 encoded version)
BLK2_OFFSET ← imm8[1:0]*32
BLK1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ← SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]
SRC2_BYTE0 ← SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]
TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

DEST[MAXVL-1:128] ← 0
MPSADBW (128-bit Legacy SSE version)

SRC_OFFSET ← imm8[1:0]*32
DEST_OFFSET ← imm8[2]*32
DEST_BYTE0 ← DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYTE1 ← DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYTE2 ← DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYTE3 ← DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYTE4 ← DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYTE5 ← DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYTE6 ← DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYTE7 ← DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYTE8 ← DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYTE9 ← DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYTE10 ← DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYTE0 ← SRC[src_OFFSET+7:src_OFFSET]
SRC_BYTE1 ← SRC[src_OFFSET+15:src_OFFSET+8]
SRC_BYTE2 ← SRC[src_OFFSET+23:src_OFFSET+16]
SRC_BYTE3 ← SRC[src_OFFSET+31:src_OFFSET+24]

TEMP0 ← ABS(DEST_BYTE0 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE1 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE2 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE1 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE2 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE3 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE4 - SRC_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE2 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE3 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE4 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE5 - SRC_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE3 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE4 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE5 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE6 - SRC_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE4 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE5 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE6 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE7 - SRC_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE5 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE6 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE7 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE6 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE7 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE8 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE7 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE8 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE9 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)MPSADBW:  __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);
VMPSADBW:  __m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);

**Flags Affected**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD  If VEX.L = 1.
MUL—Unsigned Multiply

### Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /4</td>
<td>MUL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (AX ← AL * r/m8).</td>
</tr>
<tr>
<td>REX + F6 /4</td>
<td>MUL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned multiply (AX ← AL * r/m8).</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (DX:AX ← AX * r/m16).</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (EDX:EAX ← EAX * r/m32).</td>
</tr>
<tr>
<td>REX.W + F7 /4</td>
<td>MUL r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned multiply (RDX:RAX ← RAX * r/m64).</td>
</tr>
</tbody>
</table>

**NOTES:**
- In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 4-9.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

### Table 4-9. MUL Results

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>AL</td>
<td>r/m8</td>
<td>AX</td>
</tr>
<tr>
<td>Word</td>
<td>AX</td>
<td>r/m16</td>
<td>DX:AX</td>
</tr>
<tr>
<td>Doubleword</td>
<td>EAX</td>
<td>r/m32</td>
<td>EDX:EAX</td>
</tr>
<tr>
<td>Quadword</td>
<td>RAX</td>
<td>r/m64</td>
<td>RDX:RAX</td>
</tr>
</tbody>
</table>
**Operation**

IF (Byte operation)
    THEN
        AX ← AL * SRC;
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                DX:AX ← AX * SRC;
            ELSE IF OperandSize = 32
                THEN EDX:EAX ← EAX * SRC; FI;
            ELSE (* OperandSize = 64 *)
                RDX:RAX ← RAX * SRC;
        FI;
FI;

**Flags Affected**

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
MULPD—Multiply Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 59 /r MULPD xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Multiply packed double-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Multiply packed double-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX</td>
<td>Multiply packed double-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 59 /r VMULPD xmm1 <a href="z">k1</a>, xmm2, xmm3/m128/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 59 /r VMULPD ymm1 <a href="z">k1</a>, ymm2, ymm3/m256/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 59 /r VMULPD zmm1 <a href="z">k1</a>, zmm2, zmm3/m512/m64bcst(er)</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Multiply packed double-precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
VMULPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[i+63:i]
                    FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VMULPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[MAXVL-1:256] ← 0;

VMULPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[MAXVL-1:128] ← 0

MULPD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[127:64] ← DEST[127:64] * SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent
VMULPD __m512d _mm512_mul_pd(__m512d a, __m512d b);
VMULPD __m512d _mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_maskz_mul_pd(__mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_mul_round_pd(__m512d a, __m512d b, int);
VMULPD __m512d _mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d _mm256_mul_pd(__m256d a, __m256d b);
MULPD __m128d _mm_mul_pd(__m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
MULPS—Multiply Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 59 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 59 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG 59 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 59 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 59 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 59 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Multiply the packed single-precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

VMULPS (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] ← SRC1[i+31:i] * SRC2[31:0]
        ELSE
          DEST[i+31:i] ← SRC1[i+31:i] * SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
    FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

VMULPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:256] ← 0;

VMULPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:128] ← 0

MULPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)
**Intel C/C++ Compiler Intrinsic Equivalent**

VMULPS __m512 _mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps(__m256 a, __m256 b);
VMULPS __m128 _mm_mul_ps(__m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.
MULSD—Multiply Scalar Double-Precision Floating-Point Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 59 /r MULSD xmm1,xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F:W1 59 /r VMULSD xmm1,xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F:W1 59 /r VMULSD xmm1 {k1}[z]. xmm2, xmm3/m64 {er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in the first source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

**VMULSD (EVEX encoded version)**

IF (EVEX.b = 1) AND SRC2 *is a register*
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
THEN DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
ELSE
  IF *merging-masking* ; merging-masking
  THEN *DEST[63:0] remains unchanged*
  ELSE ; zeroing-masking
  THEN DEST[63:0] ← 0
  FI
  FI;
ENDIF

**VMULSD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**MULSD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
MULSS—Multiply Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 59 /r MULSS xmm1,xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Multiply the low single-precision floating-point value in xmm2/m32 by the low single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W0 59 /r VMULSS xmm1 <a href="z">k1</a>, xmm2, xmm3/m32 [er]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VMULSS (EVEX encoded version)
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] <- SRC1[31:0] * SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*  
            ELSE ; zeroing-masking
                THEN DEST[31:0] <- 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:128] <- 0

VMULSS (VEX.128 encoded version)
DEST[31:0] <- SRC1[31:0] * SRC2[31:0]
DEST[MAXVL-1:128] <- 0

MULSS (128-bit Legacy SSE version)
DEST[31:0] <- DEST[31:0] * SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 __m128_mask_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 __m128_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 __m128_mul_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions
Underflow, Overflow, Invalid, Precision, Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
MULX — Unsigned Multiply Without Affecting Flags

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDD.LZ.F2.0F38.W0 F6 r/MULX r32a, r32b, r/m32</td>
<td>RVM</td>
<td>V/V</td>
<td>BMI2</td>
<td>Unsigned multiply of r/m32 with EDX without affecting arithmetic flags.</td>
</tr>
<tr>
<td>VEX.NDD.LZ.F2.0F38.W1 F6 r/MULX r64a, r64b, r/m64</td>
<td>RVM</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Unsigned multiply of r/m64 with RDX without affecting arithmetic flags.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>RDX/EDX is implied 64/32 bits source</td>
</tr>
</tbody>
</table>

### Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 ← EDX;
    DEST2 ← (SRC1*SRC2)[31:0];
    DEST1 ← (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 ← RDX;
    DEST2 ← (SRC1*SRC2)[63:0];
    DEST1 ← (SRC1*SRC2)[127:64];
FI

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.

```c
unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);
unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);
```

### SIMD Floating-Point Exceptions

None
Other Exceptions
See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.
MWAIT—Monitor Wait

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 C9</td>
<td>MWAIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX[bit 1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.
MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[bit 0] reporting 1.

EAX and ECX are used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. Implementation-specific conditions may cause a processor to ignore the hint and enter a different optimized state. Future processor implementations may implement several optimized “waiting” states and will select among those states based on the hint argument. Table 4-10 describes the meaning of ECX and EAX registers for MWAIT extensions.

Table 4-10. MWAIT Extension Register (ECX)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Treat interrupts as break events even if masked (e.g., even if EFLAGS.IF=0). May be set only if CPUID.05H:ECX[bit 1] = 1.</td>
</tr>
<tr>
<td>31: 1</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 4-11. MWAIT Hints Register (EAX)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 : 0</td>
<td>Sub C-state within a C-state, indicated by bits [7:4]</td>
</tr>
</tbody>
</table>
| 7 : 4| Target C-state*  
Value of 0 means C1; 1 means C2 and so on  
Value of 01111B means C0  
Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states |
| 31: 8| Reserved |

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent might not cause the processor to exit MWAIT in such cases.

For additional details of MWAIT extensions, see Chapter 14, “Power and Thermal Management,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Operation

(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension MWAIT EAX, ECX *)

```
{  
  WHILE ("Monitor Hardware is in armed state") {  
    implementation_dependent_optimized_state(EAX, ECX); }  
Set the state of Monitor Hardware as triggered;  
}
```

Intel C/C++ Compiler Intrinsic Equivalent

MWAIT: _mm_mwait(unsigned extensions, unsigned hints)
Example

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
EDX = 0 (* Hints *)

IF ( !trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)       If ECX[31:1] ≠ 0.
            If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD          If CPUID.01H:ECX.MONITOR[bit 3] = 0.
            If current privilege level is not 0.

Real Address Mode Exceptions

#GP          If ECX[31:1] ≠ 0.
            If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD          If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD          The MWAIT instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)       If RCX[63:1] ≠ 0.
            If RCX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD          If the current privilege level is not 0.
            If CPUID.01H:ECX.MONITOR[bit 3] = 0.
NEG—Two’s Complement Negation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /3</td>
<td>NEG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Two's complement negate r/m8.</td>
</tr>
<tr>
<td>REX + F6 /3</td>
<td>NEG r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Two's complement negate r/m8.</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Two's complement negate r/m16.</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Two's complement negate r/m32.</td>
</tr>
<tr>
<td>REX.W + F7 /3</td>
<td>NEG r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Two's complement negate r/m64.</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Replaces the value of operand (the destination operand) with its two’s complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF DEST = 0
    THEN CF ← 0;
    ELSE CF ← 1;
FI;
DEST ← [- (DEST)]

Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same as for protected mode exceptions.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
NOP—No Operation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 90</td>
<td>NOP</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>One byte no-operation instruction.</td>
</tr>
<tr>
<td>NP 0F 1F /0</td>
<td>NOP r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Multi-byte no-operation instruction.</td>
</tr>
<tr>
<td>NP 0F 1F /0</td>
<td>NOP r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Multi-byte no-operation instruction.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>M</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of “no operation” as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

<table>
<thead>
<tr>
<th>Length</th>
<th>Assembly</th>
<th>Byte Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>66 NOP</td>
<td>66 90H</td>
</tr>
<tr>
<td>3 bytes</td>
<td>NOP DWORD ptr [EAX]</td>
<td>0F 1F 00H</td>
</tr>
<tr>
<td>4 bytes</td>
<td>NOP DWORD ptr [EAX + 00H]</td>
<td>0F 1F 40 00H</td>
</tr>
<tr>
<td>5 bytes</td>
<td>NOP DWORD ptr [EAX + EAX*1 + 00H]</td>
<td>0F 1F 44 00 00H</td>
</tr>
<tr>
<td>6 bytes</td>
<td>66 NOP DWORD ptr [EAX + EAX*1 + 00H]</td>
<td>66 0F 1F 44 00 00H</td>
</tr>
<tr>
<td>7 bytes</td>
<td>NOP DWORD ptr [EAX + 00000000H]</td>
<td>0F 1F 80 00 00 00 00H</td>
</tr>
<tr>
<td>8 bytes</td>
<td>NOP DWORD ptr [EAX + EAX*1 + 00000000H]</td>
<td>0F 1F 84 00 00 00 00 00H</td>
</tr>
<tr>
<td>9 bytes</td>
<td>66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]</td>
<td>66 0F 1F 84 00 00 00 00 00H</td>
</tr>
</tbody>
</table>

Flags Affected

None

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
NOT—One’s Complement Negation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /2</td>
<td>NOT r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Reverse each bit of r/m8.</td>
</tr>
<tr>
<td>REX + F6 /2</td>
<td>NOT r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Reverse each bit of r/m8.</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Reverse each bit of r/m16.</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Reverse each bit of r/m32.</td>
</tr>
<tr>
<td>REX.W + F7 /2</td>
<td>NOT r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Reverse each bit of r/m64.</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Description
Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location. This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
DEST ← NOT DEST;

Flags Affected
None

Protected Mode Exceptions

#GP(0) If the destination operand points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
OR—Logical Inclusive OR

### Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compatability Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C ib</td>
<td>OR AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AL OR imm8.</td>
</tr>
<tr>
<td>0D iw</td>
<td>OR AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AX OR imm16.</td>
</tr>
<tr>
<td>0D id</td>
<td>OR EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>EAX OR imm32.</td>
</tr>
<tr>
<td>REX.W + 0D id</td>
<td>OR RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>RAX OR imm32 (sign-extended).</td>
</tr>
<tr>
<td>80 /1 ib</td>
<td>OR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 OR imm8.</td>
</tr>
<tr>
<td>REX + 80 /1 ib</td>
<td>OR r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 OR imm8.</td>
</tr>
<tr>
<td>81 /1 iw</td>
<td>OR r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 OR imm16.</td>
</tr>
<tr>
<td>81 /1 id</td>
<td>OR r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 OR imm32.</td>
</tr>
<tr>
<td>REX.W + 81 /1 id</td>
<td>OR r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 OR imm32 (sign-extended).</td>
</tr>
<tr>
<td>83 /1 ib</td>
<td>OR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 OR imm8 (sign-extended).</td>
</tr>
<tr>
<td>83 /1 ib</td>
<td>OR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 OR imm8 (sign-extended).</td>
</tr>
<tr>
<td>REX.W + 83 /1 ib</td>
<td>OR r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 OR imm8 (sign-extended).</td>
</tr>
<tr>
<td>08 /r</td>
<td>OR r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 OR r8.</td>
</tr>
<tr>
<td>REX + 08 /r</td>
<td>OR r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 OR r8.</td>
</tr>
<tr>
<td>09 /r</td>
<td>OR r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 OR r16.</td>
</tr>
<tr>
<td>09 /r</td>
<td>OR r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 OR r32.</td>
</tr>
<tr>
<td>REX.W + 09 /r</td>
<td>OR r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 OR r64.</td>
</tr>
<tr>
<td>0A /r</td>
<td>OR r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r8 OR r/m8.</td>
</tr>
<tr>
<td>REX + 0A /r</td>
<td>OR r8*, r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r8 OR r/m8.</td>
</tr>
<tr>
<td>0B /r</td>
<td>OR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 OR r/m16.</td>
</tr>
<tr>
<td>0B /r</td>
<td>OR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r32 OR r/m32.</td>
</tr>
<tr>
<td>REX.W + 0B /r</td>
<td>OR r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r64 OR r/m64.</td>
</tr>
</tbody>
</table>

### Notes:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m (r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.
In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST ← DEST OR SRC;

**Flags Affected**

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

**Protected Mode Exceptions**

- **#GP(0)** If the destination operand points to a non-writable segment.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.
ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 56/r ORPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical OR of packed double-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:W1 56 /r VORPD xmm1[k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:W1 56 /r VORPD ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:W1 56 /r VORPD zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical OR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise logical OR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
**Operation**

**VORPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b == 1) AND (SRC2 *is memory*)

THEN

DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]

ELSE

DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VORPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]


DEST[MAXVL-1:256] ← 0

**VORPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]

DEST[MAXVL-1:128] ← 0

**ORPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]

DEST[127:64] ← DEST[127:64] BITWISE OR SRC[127:64]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VORPD __m512d __m512d __mm512_or_pd (__m512d a, __m512d b);

VORPD __m512d __mm512_mask_or_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);

VORPD __m512d __mm512_maskz_or_pd (__mmask8 k, __m512d a, __m512d b);

VORPD __m256d __m256d __mm256_or_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);

VORPD __m256d __mm256_mask_or_pd (__mmask8 k, __m256d a, __m256d b);

VORPD __m128d __m128d __mm128_or_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);

VORPD __m128d __mm128_mask_or_pd (__mmask8 k, __m128d a, __m128d b);

VORPD __m128d __mm128_or_pd (__m128d a, __m128d b);

ORPD __m128d __m128d __mm_or_pd (__m128d a, __m128d b);
SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
### ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 56 /r ORPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F:W0 56 /r VORPS xmm1 [k1]{[2]}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F:W0 56 /r VORPS ymm1 {k1}[2], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F:W0 56 /r VORPS zmm1 {k1}[2], zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical OR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Performs a bitwise logical OR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

**VORPS (EVEX encoded versions)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\[ i \leftarrow j \times 32 \]

IF \( k1[j] \) OR *no writemask*

THEN

IF (EVEX.b == 1) AND (SRC2 *is memory*)

THEN

\[ \text{DEST}[i+31:i] \leftarrow \text{SRC1}[i+31:i] \text{ BITWISE OR SRC2}[31:0] \]

ELSE

\[ \text{DEST}[i+31:i] \leftarrow \text{SRC1}[i+31:i] \text{ BITWISE OR SRC2}[i+31:i] \]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+31:i] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\[ \text{DEST}[i+31:i] \leftarrow 0 \]

FI

FI

ENDFOR

\[ \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \]

**VORPS (VEX.256 encoded version)**

\[ \text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE OR SRC2}[31:0] \]

\[ \text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE OR SRC2}[63:32] \]

\[ \text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE OR SRC2}[95:64] \]

\[ \text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE OR SRC2}[127:96] \]

\[ \text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE OR SRC2}[159:128] \]

\[ \text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE OR SRC2}[191:160] \]

\[ \text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE OR SRC2}[223:192] \]

\[ \text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE OR SRC2}[255:224]. \]

\[ \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \]

**VORPS (VEX.128 encoded version)**

\[ \text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE OR SRC2}[31:0] \]

\[ \text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE OR SRC2}[63:32] \]

\[ \text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE OR SRC2}[95:64] \]

\[ \text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE OR SRC2}[127:96] \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

**ORPS (128-bit Legacy SSE version)**

\[ \text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE OR SRC2}[31:0] \]

\[ \text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE OR SRC2}[63:32] \]

\[ \text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE OR SRC2}[95:64] \]

\[ \text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE OR SRC2}[127:96] \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow \text{Unmodified} \]
Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 _mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 _mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 _mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 _mm256_mask_or_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VORPS __m256 _mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 _mm_mask_or_ps ( __m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 _mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
OUT—Output to Port

**Description**
Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 18, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**
After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**OUT—Output to Port**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E6 ib</td>
<td>OUT imm8, AL</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Output byte in AL to I/O port address imm8.</td>
</tr>
<tr>
<td>E7 ib</td>
<td>OUT imm8, AX</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Output word in AX to I/O port address imm8.</td>
</tr>
<tr>
<td>E7 ib</td>
<td>OUT imm8, EAX</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Output doubleword in EAX to I/O port address imm8.</td>
</tr>
<tr>
<td>EE</td>
<td>OUT DX, AL</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output byte in AL to I/O port address in DX.</td>
</tr>
<tr>
<td>EF</td>
<td>OUT DX, AX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output word in AX to I/O port address in DX.</td>
</tr>
<tr>
<td>EF</td>
<td>OUT DX, EAX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output doubleword in EAX to I/O port address in DX.</td>
</tr>
</tbody>
</table>

**NOTES:**

* See IA-32 Architecture Compatibility section below.
Operation

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
   THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
      IF (Any I/O Permission Bit for I/O port being accessed = 1)
         THEN (* I/O operation is not allowed *)
            #GP(0);
      ELSE (* I/O operation is allowed *)
         DEST ← SRC; (* Writes to selected I/O port *)
   FI;
ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
   DEST ← SRC; (* Writes to selected I/O port *)
FI;

Flags Affected

None

Protected Mode Exceptions

#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

Same as protected mode exceptions.
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6E</td>
<td>OUTS DX, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.</td>
</tr>
<tr>
<td>6F</td>
<td>OUTS DX, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.</td>
</tr>
<tr>
<td>6F</td>
<td>OUTS DX, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.</td>
</tr>
<tr>
<td>6E</td>
<td>OUTSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.</td>
</tr>
<tr>
<td>6F</td>
<td>OUTSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.</td>
</tr>
<tr>
<td>6F</td>
<td>OUTSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.</td>
</tr>
</tbody>
</table>

NOTES:
* See IA-32 Architecture Compatibility section below.
** In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the DS:(E)SI or RSI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the SI/ESI/RSI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI register is incremented; if the DF flag is 1, the SI/ESI/RSI register is decremented.) The SI/ESI/RSI register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.
The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ /REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 18, "Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is support using the prefix 67H, but 16-bit address is not supported in 64-bit mode.

**IA-32 Architecture Compatibility**

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

**Operation**

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))

  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)

    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST ← SRC; (* Writes to I/O port *)
    FI;

  ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Writes to I/O port *)
  FI;

Byte transfer:

  IF 64-bit mode
    Then
      IF 64-Bit Address Size
        THEN
          IF DF = 0
            THEN RSI ← RSI RSI + 1;
            ELSE RSI ← RSI or – 1;
          FI;
          ELSE (* 32-Bit Address Size *)
            IF DF = 0
              THEN ESI ← ESI + 1;
              ELSE ESI ← ESI – 1;
            FI;
        FI;
      ELSE
        IF DF = 0
          THEN (E)SI ← (E)SI + 1;
          ELSE (E)SI ← (E)SI – 1;
        FI;
    FI;

Word transfer:

  IF 64-bit mode
Then
  IF 64-Bit Address Size
    THEN
      IF DF = 0
        THEN RSI ← RSI + 2;
        ELSE RSI ← RSI or -2;
      FI;
      ELSE (* 32-Bit Address Size *)
        IF DF = 0
          THEN ESI ← ESI + 2;
          ELSE ESI ← ESI - 2;
        FI;
    FI;
ELSE
  IF DF = 0
    THEN (E)SI ← (E)SI + 2;
    ELSE (E)SI ← (E)SI - 2;
  FI;
FI;

Doubleword transfer:
  IF 64-bit mode
    THEN
      IF 64-Bit Address Size
        THEN
          IF DF = 0
            THEN RSI ← RSI + 4;
            ELSE RSI ← RSI or -4;
          FI;
          ELSE (* 32-Bit Address Size *)
            IF DF = 0
              THEN ESI ← ESI + 4;
              ELSE ESI ← ESI - 4;
            FI;
        FI;
      ELSE
        IF DF = 0
          THEN (E)SI ← (E)SI + 4;
          ELSE (E)SI ← (E)SI - 4;
        FI;
    FI;

Flags Affected

None
Protected Mode Exceptions

- #GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
  - If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment.
  - If the segment register contains a NULL segment selector.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
  - If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 1C /r¹</td>
<td>A V/V</td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.</td>
</tr>
<tr>
<td>PABSB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 1C /r</td>
<td>A V/V</td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>PABSB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 38 1D /r¹</td>
<td>A V/V</td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.</td>
</tr>
<tr>
<td>PABSw mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 1D /r</td>
<td>A V/V</td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>PABSw xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 38 1E /r¹</td>
<td>A V/V</td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.</td>
</tr>
<tr>
<td>PABSD mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 1E /r</td>
<td>A V/V</td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>PABSD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VEX.128.66.0F38.WIG 1C /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSB xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>AVX</td>
<td></td>
<td></td>
<td>Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
</tbody>
</table>

**VEX.128.66.0F38.WIG 1D /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSW xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>AVX</td>
<td></td>
<td></td>
<td>Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
</tbody>
</table>

**VEX.128.66.0F38.WIG 1E /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSD xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>AVX</td>
<td></td>
<td></td>
<td>Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
</tbody>
</table>

**VEX.256.66.0F38.WIG 1C /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSB ymm1, ymm2/m256</td>
<td>A V/V</td>
<td>AVX2</td>
<td></td>
<td></td>
<td>Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1.</td>
</tr>
</tbody>
</table>

**VEX.256.66.0F38.WIG 1D /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSW ymm1, ymm2/m256</td>
<td>A V/V</td>
<td>AVX2</td>
<td></td>
<td></td>
<td>Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.</td>
</tr>
</tbody>
</table>

**VEX.256.66.0F38.WIG 1E /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSD ymm1, ymm2/m256</td>
<td>A V/V</td>
<td>AVX2</td>
<td></td>
<td></td>
<td>Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.</td>
</tr>
</tbody>
</table>

**EVEX.128.66.0F38.WIG 1C /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSB xmm1 [k1][z], xmm2/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td></td>
<td>Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**EVEX.256.66.0F38.WIG 1C /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSB ymm1 [k1][z], ymm2/m256</td>
<td>B V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td></td>
<td>Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**EVEX.512.66.0F38.WIG 1C /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSB zmm1 [k1][z], zmm2/m512</td>
<td>B V/V</td>
<td>AVX512Bw</td>
<td></td>
<td></td>
<td>Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**EVEX.128.66.0F38.WIG 1D /r**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPABSW xmm1 [k1][z], xmm2/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td></td>
<td>Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

**EvX encoded VPABSD/Q:** The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

**EvX encoded VPABSB/W:** The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

**VEx.256 encoded versions:** The source operand is a ZMM/YMM/XMM register, and a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

**VEx.128 encoded versions:** The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

**Notes:**
128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PABSB with 128 bit operands:
Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] ← ABS(SRC[127:120])

VPABSB with 128 bit operands:
Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] ← ABS(SRC[127:120])

VPABSB with 256 bit operands:
Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 31st bytes
Unsigned DEST[255:248] ← ABS(SRC[255:248])

VPABSB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN
    Unsigned DEST[i+7:i] ← ABS(SRC[i+7:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] ← 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0

PABSW with 128 bit operands:
Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112] ← ABS(SRC[127:112])

VPABSW with 128 bit operands:
Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112] ← ABS(SRC[127:112])

VPABSW with 256 bit operands:
Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 15th 16-bit words
Unsigned DEST[255:240] ← ABS(SRC[255:240])
VPABSW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      Unsigned DEST[i+15:i] ← ABS(SRC[i+15:i])
    ELSE
      IF *(merging-masking)* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *(zeroing-masking)* ; zeroing-masking
        DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

PABSD with 128 bit operands:
Unsigned DEST[31:0]←ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96]←ABS(SRC[127:96])

VPABSD with 128 bit operands:
Unsigned DEST[31:0]←ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96]←ABS(SRC[127:96])

VPABSD with 256 bit operands:
Unsigned DEST[31:0]←ABS(SRC[31:0])
Repeat operation for 2nd through 7th 32-bit double words
Unsigned DEST[255:224]←ABS(SRC[255:224])

VPABSD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN
          Unsigned DEST[i+31:i] ← ABS(SRC[31:0])
        ELSE
          Unsigned DEST[i+31:i] ← ABS(SRC[i+31:i])
        FI;
      ELSE
        IF *(merging-masking)* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE *(zeroing-masking)* ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
    FI
  ELSE
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
VPABSQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+63:i] ← ABS(SRC[63:0])
                ELSE
                    Unsigned DEST[i+63:i] ← ABS(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR;

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

VPABSB__m512i _mm512_abs_epi8 ( __m512i a)
VPABSW__m512i _mm512_abs_epi16 ( __m512i a)
VPABSB__m512i _mm512_mask_abs_epi8 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i _mm512_mask_abs_epi16 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i _mm512_maskz_abs_epi8 ( __mmask64 m, __m512i a)
VPABSW__m512i _mm512_maskz_abs_epi16 ( __mmask32 m, __m512i a)
VPABSB__m256i _mm256_mask_abs_epi8 ( __m256i s, __mmask32 m, __m256i a)
VPABSW__m256i _mm256_mask_abs_epi16 ( __m256i s, __mmask16 m, __m256i a)
VPABSB__m256i _mm256_maskz_abs_epi8 ( __mmask32 m, __m256i a)
VPABSW__m256i _mm256_maskz_abs_epi16 ( __mmask16 m, __m256i a)
VPABSB__m128i _mm_mask_abs_epi8 ( __m128i s, __mmask16 m, __m128i a)
VPABSW__m128i _mm_mask_abs_epi16 ( __m128i s, __mmask8 m, __m128i a)
VPABSB__m128i _mm_maskz_abs_epi8 ( __mmask16 m, __m128i a)
VPABSW__m128i _mm_maskz_abs_epi16 ( __mmask8 m, __m128i a)
VPABSD __m256i _mm256_mask_abs_epi32( __m256i s, __mmask8 k, __m256i a)
VPABSD __m256i _mm256_maskz_abs_epi32( __mmask8 k, __m256i a)
VPABSD __m128i _mm_mask_abs_epi32( __m128i s, __mmask8 k, __m128i a)
VPABSD __m128i _mm_maskz_abs_epi32( __mmask8 k, __m128i a)
VPABSD  __m512i _mm512_mask_abs_epi32( __m512i s, __mmask16 k, __m512i a)
VPABSD  __m512i _mm512_maskz_abs_epi32( __mmask16 k, __m512i a)
VPABSD  __m512i _mm512_mask_abs_epi64( __m512i s, __mmask8 k, __m512i a)
VPABSD  __m512i _mm512_maskz_abs_epi64( __mmask8 k, __m512i a)
VPABSQ __m512i _mm512_mask_abs_epi64( __m512i s, __mmask8 k, __m512i a)
VPABSQ __m512i _mm512_maskz_abs_epi64( __mmask8 k, __m512i a)
VPABSB __m128i _mm_abs_epi8 ( __m128i a)
VPABSQ __m128i _mm_abs_epi8 ( __m128i a)
VPABSB __m256i_mm256_abs_epi8 (__m256i a)
PABSW __m128i_mm_abs_epi16 (__m128i a)
VPABSW __m128i_mm_abs_epi16 (__m128i a)
VPABSW __m256i_mm256_abs_epi16 (__m256i a)
PABSD __m128i_mm_abs_epi32 (__m128i a)
VPABSD __m128i_mm_abs_epi32 (__m128i a)
VPABSD __m256i_mm256_abs_epi32 (__m256i a)

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPABSD/Q, see Exceptions Type E4.
EVEX-encoded VPABSB/W, see Exceptions Type E4.nb.
## PACKSSWB/PACKSSDW—Pack with Signed Saturation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 63 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Converts 4 packed signed word integers from <code>mm1</code> and from <code>mm2/m64</code> into 8 packed signed byte integers in <code>mm1</code> using signed saturation.</td>
</tr>
<tr>
<td>PACKSSWB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 63 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Converts 8 packed signed word integers from <code>xmm1</code> and from <code>xmm2/m128</code> into 16 packed signed byte integers in <code>xmm1</code> using signed saturation.</td>
</tr>
<tr>
<td>PACKSSWB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 6B /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Converts 2 packed signed doubleword integers from <code>mm1</code> and from <code>mm2/m64</code> into 4 packed signed word integers in <code>mm1</code> using signed saturation.</td>
</tr>
<tr>
<td>PACKSSDW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 6B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Converts 4 packed signed doubleword integers from <code>xmm1</code> and from <code>xmm2/m128</code> into 8 packed signed word integers in <code>xmm1</code> using signed saturation.</td>
</tr>
<tr>
<td>PACKSSDW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 63 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Converts 8 packed signed word integers from <code>xmm2</code> and from <code>xmm3/m128</code> into 16 packed signed byte integers in <code>xmm1</code> using signed saturation.</td>
</tr>
<tr>
<td>VPACKSSWB xmm1,xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 6B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Converts 4 packed signed doubleword integers from <code>ymm1</code> and from <code>ymm2/m128</code> into 8 packed signed word integers in <code>ymm1</code> using signed saturation.</td>
</tr>
<tr>
<td>VPACKSSDW ymm1, ymm2, xmm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 63 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Converts 16 packed signed word integers from <code>ymm2</code> and from <code>ymm3/m256</code> into 32 packed signed byte integers in <code>ymm1</code> using signed saturation.</td>
</tr>
<tr>
<td>VPACKSSWB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 6B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Converts 8 packed signed doubleword integers from <code>ymm2</code> and from <code>ymm3/m256</code> into 16 packed signed word integers in <code>ymm1</code> using signed saturation.</td>
</tr>
<tr>
<td>VPACKSSDW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 63 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Converts packed signed word integers from <code>xmm2</code> and from <code>xmm3/m128</code> into packed signed byte integers in <code>xmm1</code> using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKSSWB xmm1 {k1}[z], xmm2, xmm3/m128</td>
<td></td>
<td>AVX512BW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG 63 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Converts packed signed word integers from <code>ymm2</code> and from <code>ymm3/m256</code> into packed signed byte integers in <code>ymm1</code> using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKSSWB ymm1 {k1}[z], ymm2, ymm3/m256</td>
<td></td>
<td>AVX512BW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG 63 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Converts packed signed word integers from <code>zmm2</code> and from <code>zmm3/m512</code> into packed signed byte integers in <code>zmm1</code> using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKSSWB zmm1 {k1}[z], zmm2, zmm3/m512</td>
<td></td>
<td>AVX512BW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W0 6B /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Converts packed signed doubleword integers from <code>xmm2</code> and from <code>xmm3/m128/m32bcst</code> into packed signed word integers in <code>xmm1</code> using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKSSDW xmm1 {k1}[z], xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td>AVX512BW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed doubleword value is beyond the range of an unsigned word (i.e., greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.
VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM destination register destination are unmodified.

Operation

PACKSSwB instruction (128-bit Legacy SSE version)

\[
\begin{align*}
\text{DEST}[7:0] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[15:0]); \\
\text{DEST}[15:8] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[31:16]); \\
\text{DEST}[23:16] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[47:32]); \\
\text{DEST}[31:24] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[63:48]); \\
\text{DEST}[39:32] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[79:64]); \\
\text{DEST}[47:40] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[95:80]); \\
\text{DEST}[55:48] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[111:96]); \\
\text{DEST}[63:56] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{DEST}[127:112]); \\
\text{DEST}[71:64] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[15:0]); \\
\text{DEST}[79:72] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[31:16]); \\
\text{DEST}[87:80] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[47:32]); \\
\text{DEST}[95:88] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[63:48]); \\
\text{DEST}[103:96] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[79:64]); \\
\text{DEST}[111:104] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[95:80]); \\
\text{DEST}[119:112] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[111:96]); \\
\text{DEST}[127:120] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC}[127:112]); \\
\text{DEST}[\text{MAXVL}-1:128] & \text{(Unmodified)}
\end{align*}
\]

PACKSSDw instruction (128-bit Legacy SSE version)

\[
\begin{align*}
\text{DEST}[15:0] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{DEST}[31:0]); \\
\text{DEST}[31:16] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{DEST}[63:32]); \\
\text{DEST}[47:32] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{DEST}[95:64]); \\
\text{DEST}[63:48] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{DEST}[127:96]); \\
\text{DEST}[79:64] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC}[31:0]); \\
\text{DEST}[95:80] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC}[63:32]); \\
\text{DEST}[111:96] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC}[95:64]); \\
\text{DEST}[127:112] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC}[127:96]); \\
\text{DEST}[\text{MAXVL}-1:128] & \text{(Unmodified)}
\end{align*}
\]
VPACKSSWB instruction (VEX.128 encoded version)

\[
\begin{align*}
    \text{DEST}[7:0] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[15:0]); \\
    \text{DEST}[15:8] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[31:16]); \\
    \text{DEST}[23:16] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[47:32]); \\
    \text{DEST}[31:24] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[63:48]); \\
    \text{DEST}[39:32] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[79:64]); \\
    \text{DEST}[47:40] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[95:80]); \\
    \text{DEST}[55:48] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[111:96]); \\
    \text{DEST}[63:56] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[127:112]); \\
    \text{DEST}[71:64] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[15:0]); \\
    \text{DEST}[79:72] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[31:16]); \\
    \text{DEST}[87:80] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[47:32]); \\
    \text{DEST}[95:88] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[63:48]); \\
    \text{DEST}[103:96] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[79:64]); \\
    \text{DEST}[111:104] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[95:80]); \\
    \text{DEST}[119:112] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[111:96]); \\
    \text{DEST}[127:120] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[127:112]); \\
    \text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0;
\end{align*}
\]

VPACKSSDw instruction (VEX.128 encoded version)

\[
\begin{align*}
    \text{DEST}[15:0] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC1}[31:0]); \\
    \text{DEST}[31:16] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC1}[63:32]); \\
    \text{DEST}[47:32] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC1}[95:64]); \\
    \text{DEST}[63:48] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC1}[127:96]); \\
    \text{DEST}[79:64] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC2}[31:0]); \\
    \text{DEST}[95:80] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC2}[63:32]); \\
    \text{DEST}[111:96] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC2}[95:64]); \\
    \text{DEST}[127:112] & \leftarrow \text{SaturateSignedDwordToSignedWord} (\text{SRC2}[127:96]); \\
    \text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0;
\end{align*}
\]

VPACKSSWB instruction (VEX.256 encoded version)

\[
\begin{align*}
    \text{DEST}[7:0] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[15:0]); \\
    \text{DEST}[15:8] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[31:16]); \\
    \text{DEST}[23:16] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[47:32]); \\
    \text{DEST}[31:24] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[63:48]); \\
    \text{DEST}[39:32] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[79:64]); \\
    \text{DEST}[47:40] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[95:80]); \\
    \text{DEST}[55:48] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[111:96]); \\
    \text{DEST}[63:56] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[127:112]); \\
    \text{DEST}[71:64] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[15:0]); \\
    \text{DEST}[79:72] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[31:16]); \\
    \text{DEST}[87:80] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[47:32]); \\
    \text{DEST}[95:88] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[63:48]); \\
    \text{DEST}[103:96] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[79:64]); \\
    \text{DEST}[111:104] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[95:80]); \\
    \text{DEST}[119:112] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[111:96]); \\
    \text{DEST}[127:120] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC2}[127:112]); \\
    \text{DEST}[135:128] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[143:128]); \\
    \text{DEST}[143:136] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[159:144]); \\
    \text{DEST}[151:144] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[175:160]); \\
    \text{DEST}[159:152] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[191:176]); \\
    \text{DEST}[167:160] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[207:192]); \\
    \text{DEST}[175:168] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[223:208]); \\
    \text{DEST}[183:176] & \leftarrow \text{SaturateSignedWordToSignedByte} (\text{SRC1}[239:224]);
\end{align*}
\]
DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
DEST[MAXVL-1:256] ← 0;

VPACKSSDW instruction (VEX.256 encoded version)
DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);
DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);
DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);
DEST[MAXVL-1:256] ← 0;

VPACKSSWB (EVEX encoded versions)
KL, VL = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
TMP_DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
IF VL >= 256
TMP_DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
TMP_DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);

4-190 Vol. 2B PACKSSWB/PACKSSDW—Pack with Signed Saturation
TMP_DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
TMP_DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);

FI;
IF VL >= 512
TMP_DEST[279:272] ← SaturateSignedWordToSignedByte (SRC1[303:288]);
TMP_DEST[295:288] ← SaturateSignedWordToSignedByte (SRC1[335:320]);
TMP_DEST[319:312] ← SaturateSignedWordToSignedByte (SRC1[383:368]);
TMP_DEST[335:328] ← SaturateSignedWordToSignedByte (SRC2[287:272]);
TMP_DEST[343:336] ← SaturateSignedWordToSignedByte (SRC2[303:288]);
TMP_DEST[359:352] ← SaturateSignedWordToSignedByte (SRC2[335:320]);

FI;
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN
    DEST[i+7:i] ← TMP_DEST[i+7:i]
ELSE
   IF *merging-masking* ; merging-masking
      THEN "DEST[i+7:i] remains unchanged"
   ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i]  0
   FI
FI;
ENDFOR;
DEST[MAXVL-1:VL]  0

VPACKSSDW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j  0 TO ((KL/2) - 1)
   i  j * 32
   IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN
         TMP_SRC2[i+31:i]  SRC2[31:0]
      ELSE
         TMP_SRC2[i+31:i]  SRC2[i+31:i]
      FI;
ENDFOR;

TMP_DEST[15:0]  SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16]  SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32]  SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48]  SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64]  SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80]  SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96]  SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
IF VL >= 256
   TMP_DEST[143:128]  SaturateSignedDwordToSignedWord (SRC1[159:128]);
   TMP_DEST[159:144]  SaturateSignedDwordToSignedWord (SRC1[191:160]);
   TMP_DEST[191:176]  SaturateSignedDwordToSignedWord (SRC1[255:224]);
   TMP_DEST[207:192]  SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
FI;
IF VL >= 512
   TMP_DEST[271:256]  SaturateSignedDwordToSignedWord (SRC1[287:256]);
   TMP_DEST[287:272]  SaturateSignedDwordToSignedWord (SRC1[319:288]);
   TMP_DEST[303:288]  SaturateSignedDwordToSignedWord (SRC1[351:320]);
   TMP_DEST[335:320]  SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
   TMP_DEST[399:384]  SaturateSignedDwordToSignedWord (SRC1[415:384]);
   TMP_DEST[415:400]  SaturateSignedDwordToSignedWord (SRC1[447:416]);
   TMP_DEST[431:416]  SaturateSignedDwordToSignedWord (SRC1[479:448]);
TMP_DEST[447:432] ← SaturateSignedDwordToSignedWord (SRC1[511:480]);
TMP_DEST[463:448] ← SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
TMP_DEST[495:480] ← SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
TMP_DEST[511:496] ← SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents
VPACKSSDW__m512i _mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_maskz_packs_epi32( __mmask32 k, __m512i m1, __m512i m2);

VPACKSSDW__m256i _mm256_packs_epi16(__m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_mask_packs_epi16(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_maskz_packs_epi16( __mmask16 k, __m256i m1, __m256i m2);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPACKSSDW, see Exceptions Type E4NF.
EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb.
**PACKUSDW—Pack with Unsigned Saturation**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Convert 4 packed signed doubleword integers from xmm1 and 4 packed signed doubleword integers from xmm2/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert 4 packed signed doubleword integers from xmm2 and 4 packed signed doubleword integers from xmm3/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Convert 8 packed signed doubleword integers from ymm2 and 8 packed signed doubleword integers from ymm3/m256 into 16 packed unsigned word integers in ymm1 using unsigned saturation.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38,W0 2B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Convert packed signed doubleword integers from xmm2 and packed signed doubleword integers from xmm3/m128/m32bcst into packed unsigned word integers in xmm1 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38,W0 2B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Convert packed signed doubleword integers from ymm2 and packed signed doubleword integers from ymm3/m256/m32bcst into packed unsigned word integers in ymm1 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38,W0 2B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Convert packed signed doubleword integers from zmm2 and packed signed doubleword integers from zmm3/m512/m32bcst into packed unsigned word integers in zmm1 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVENX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding destination register destination are unmodified.
Operation

PACKUSDW (Legacy SSE instruction)
TMP[15:0] ← (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] ← (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[47:32] ← (DEST[95:64] < 0) ? 0 : DEST[79:64];
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
TMP[79:64] ← (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[79:64] ← (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[111:96] ← (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] ← (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[MAXVL-1:128] ← 0;

PACKUSDW (VEX.128 encoded version)
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[MAXVL-1:128] ← 0;

VPACKUSDW (VEX.256 encoded version)
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
\[
\text{TMP[127:112]} \leftarrow (\text{SRC2}[127:96] < 0) \ ? 0 : \text{SRC2}[111:96]; \\
\text{DEST[127:112]} \leftarrow (\text{SRC2}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[127:112]}; \\
\text{TMP[143:128]} \leftarrow (\text{SRC1}[159:128] < 0) \ ? 0 : \text{SRC1}[143:128]; \\
\text{DEST[143:128]} \leftarrow (\text{SRC1}[159:128] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[143:128]}; \\
\text{TMP[159:144]} \leftarrow (\text{SRC1}[191:160] < 0) \ ? 0 : \text{SRC1}[175:160]; \\
\text{DEST[159:144]} \leftarrow (\text{SRC1}[191:160] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[159:144]}; \\
\text{TMP[175:160]} \leftarrow (\text{SRC1}[223:192] < 0) \ ? 0 : \text{SRC1}[207:192]; \\
\text{DEST[175:160]} \leftarrow (\text{SRC1}[223:192] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[175:160]}; \\
\text{TMP[191:176]} \leftarrow (\text{SRC1}[255:224] < 0) \ ? 0 : \text{SRC1}[239:224]; \\
\text{DEST[191:176]} \leftarrow (\text{SRC1}[255:224] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[191:176]}; \\
\text{TMP[207:192]} \leftarrow (\text{SRC2}[159:128] < 0) \ ? 0 : \text{SRC2}[143:128]; \\
\text{DEST[207:192]} \leftarrow (\text{SRC2}[159:128] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[207:192]}; \\
\text{TMP[223:208]} \leftarrow (\text{SRC2}[191:160] < 0) \ ? 0 : \text{SRC2}[175:160]; \\
\text{DEST[223:208]} \leftarrow (\text{SRC2}[191:160] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[223:208]}; \\
\text{TMP[239:224]} \leftarrow (\text{SRC2}[223:192] < 0) \ ? 0 : \text{SRC2}[207:192]; \\
\text{DEST[239:224]} \leftarrow (\text{SRC2}[223:192] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[239:224]}; \\
\text{TMP[255:240]} \leftarrow (\text{SRC2}[255:224] < 0) \ ? 0 : \text{SRC2}[239:224]; \\
\text{DEST[255:240]} \leftarrow (\text{SRC2}[255:224] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[255:240]}; \\
\text{DEST[MAXVL-1:256]} \leftarrow 0;
\]

**VPACKUSDw (EVEX encoded versions)**

\((KL, VL) = (8, 128), (16, 256), (32, 512)\)

FOR \(j \leftarrow 0 \ TO ((KL/2) - 1)\)

\(i \leftarrow j \times 32\)

IF (EVEX.b == 1) AND (SRC2 *is memory*)

THEN

\(\text{TMP_SRC2}[i+31:i] \leftarrow \text{SRC2}[31:0]\)

ELSE

\(\text{TMP_SRC2}[i+31:i] \leftarrow \text{SRC2}[i+31:i]\)

FI;

ENDFOR;

\(\text{TMP[15:0]} \leftarrow (\text{SRC1}[31:0] < 0) \ ? 0 : \text{SRC1}[15:0];\)

\(\text{DEST[15:0]} \leftarrow (\text{SRC1}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[15:0]};\)

\(\text{TMP[31:16]} \leftarrow (\text{SRC1}[63:32] < 0) \ ? 0 : \text{SRC1}[47:32];\)

\(\text{DEST[31:16]} \leftarrow (\text{SRC1}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[31:16]};\)

\(\text{TMP[47:32]} \leftarrow (\text{SRC1}[95:64] < 0) \ ? 0 : \text{SRC1}[79:64];\)

\(\text{DEST[47:32]} \leftarrow (\text{SRC1}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[47:32]};\)

\(\text{TMP[63:48]} \leftarrow (\text{SRC1}[127:96] < 0) \ ? 0 : \text{SRC1}[111:96];\)

\(\text{DEST[63:48]} \leftarrow (\text{SRC1}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[63:48]};\)

\(\text{TMP[79:64]} \leftarrow (\text{TMP_SRC2}[31:0] < 0) \ ? 0 : \text{TMP_SRC2}[15:0];\)

\(\text{DEST[79:64]} \leftarrow (\text{TMP_SRC2}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[79:64]};\)

\(\text{TMP[95:80]} \leftarrow (\text{TMP_SRC2}[63:32] < 0) \ ? 0 : \text{TMP_SRC2}[47:32];\)

\(\text{DEST[95:80]} \leftarrow (\text{TMP_SRC2}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[95:80]};\)

\(\text{TMP[111:96]} \leftarrow (\text{TMP_SRC2}[95:64] < 0) \ ? 0 : \text{TMP_SRC2}[79:64];\)

\(\text{DEST[111:96]} \leftarrow (\text{TMP_SRC2}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[111:96]};\)

\(\text{TMP[127:112]} \leftarrow (\text{TMP_SRC2}[127:96] < 0) \ ? 0 : \text{TMP_SRC2}[111:96];\)

\(\text{DEST[127:112]} \leftarrow (\text{TMP_SRC2}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[127:112]};\)

IF \(VL > 256\)

\(\text{TMP[143:128]} \leftarrow (\text{SRC1}[159:128] < 0) \ ? 0 : \text{SRC1}[143:128];\)

\(\text{DEST[143:128]} \leftarrow (\text{SRC1}[159:128] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[143:128]};\)

\(\text{TMP[159:144]} \leftarrow (\text{SRC1}[191:160] < 0) \ ? 0 : \text{SRC1}[175:160];\)

\(\text{DEST[159:144]} \leftarrow (\text{SRC1}[191:160] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP[159:144]};\)
```assembly
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];

FI;
IF VL >= 512
    TMP[271:256] ← (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
    TMP[287:272] ← (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
    TMP[303:288] ← (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
    TMP[335:320] ← (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
    TMP[351:336] ← (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
    TMP[415:400] ← (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
    TMP[431:416] ← (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
    TMP[447:432] ← (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
    TMP[463:448] ← (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
    TMP[491:476] ← (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
    TMP[511:492] ← (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:j] ← TMP_DEST[i+15:j]
        ELSE
            IF *merging-masking*
                ; merging-masking
```
THEN *DEST[i+15:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+15:i] ← 0
FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSDW__m512i _mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i __mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_maskz_packus_epi32( __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i __mm256_mask_packus_epi32( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i _mm256_maskz_packus_epi32( __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i __mm_mask_packus_epi32( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i _mm128_maskz_packus_epi32( __mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i __mm256_packus_epi32(__m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.
### PACKUSWB—Pack with Unsigned Saturation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 67 /r</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Converts 4 signed word integers from ( mm ) and 4 signed word integers from ( mm/m64 ) into 8 unsigned byte integers in ( mm ) using unsigned saturation.</td>
</tr>
<tr>
<td>PACKUSWB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 67 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Converts 8 signed word integers from ( xmm1 ) and 8 signed word integers from ( xmm2/m128 ) into 16 unsigned byte integers in ( xmm1 ) using unsigned saturation.</td>
</tr>
<tr>
<td>PACKUSWB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG 67 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Converts 8 signed word integers from ( xmm2 ) and 8 signed word integers from ( xmm3/m128 ) into 16 unsigned byte integers in ( xmm1 ) using unsigned saturation.</td>
</tr>
<tr>
<td>VPACKUSWB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG 67 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Converts 16 signed word integers from ( ymm2 ) and 16 signed word integers from ( ymm3/m256 ) into 32 unsigned byte integers in ( ymm1 ) using unsigned saturation.</td>
</tr>
<tr>
<td>VPACKUSWB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG 67 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Converts signed word integers from ( xmm2 ) and signed word integers from ( xmm3/m128 ) into unsigned byte integers in ( xmm1 ) using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKUSWB xmm1(k1)[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG 67 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Converts signed word integers from ( ymm2 ) and signed word integers from ( ymm3/m256 ) into unsigned byte integers in ( ymm1 ) using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKUSWB ymm1(k1)[z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG 67 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Converts signed word integers from ( zmm2 ) and signed word integers from ( zmm3/m512 ) into unsigned byte integers in ( zmm1 ) using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKUSWB zmm1(k1)[z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Notes:

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts 4, 8, 16 or 32 signed word integers from the destination operand (first operand) and 4, 8, 16 or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.
VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation**

**PACKUSwB (with 64-bit operands)**

\[
\begin{align*}
\text{DEST}[7:0] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[15:0]; \\
\text{DEST}[15:8] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[31:16]; \\
\text{DEST}[23:16] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[47:32]; \\
\text{DEST}[31:24] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[63:48]; \\
\text{DEST}[39:32] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[15:0]; \\
\text{DEST}[47:40] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[31:16]; \\
\text{DEST}[55:48] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[47:32]; \\
\text{DEST}[63:56] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[63:48]; \\
\end{align*}
\]

**PACKUSwB (Legacy SSE instruction)**

\[
\begin{align*}
\text{DEST}[7:0] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[15:0]; \\
\text{DEST}[15:8] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[31:16]; \\
\text{DEST}[23:16] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[47:32]; \\
\text{DEST}[31:24] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[63:48]; \\
\text{DEST}[39:32] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[79:64]; \\
\text{DEST}[47:40] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[95:80]; \\
\text{DEST}[55:48] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[111:96]; \\
\text{DEST}[63:56] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ DEST}[127:112]; \\
\text{DEST}[71:64] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[15:0]; \\
\text{DEST}[79:72] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[31:16]; \\
\text{DEST}[87:80] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[47:32]; \\
\text{DEST}[95:88] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[63:48]; \\
\text{DEST}[103:96] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[79:64]; \\
\text{DEST}[111:104] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[95:80]; \\
\text{DEST}[119:112] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[111:96]; \\
\text{DEST}[127:120] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC}[127:112]; \\
\end{align*}
\]

**PACKUSwB (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[7:0] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[15:0]; \\
\text{DEST}[15:8] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[31:16]; \\
\text{DEST}[23:16] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[47:32]; \\
\text{DEST}[31:24] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[63:48]; \\
\text{DEST}[39:32] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[79:64]; \\
\text{DEST}[47:40] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[95:80]; \\
\text{DEST}[55:48] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[111:96]; \\
\text{DEST}[63:56] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC1}[127:112]; \\
\text{DEST}[71:64] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC2}[15:0]; \\
\text{DEST}[79:72] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC2}[31:16]; \\
\text{DEST}[87:80] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC2}[47:32]; \\
\text{DEST}[95:88] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC2}[63:48]; \\
\text{DEST}[103:96] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC2}[79:64]; \\
\text{DEST}[111:104] & \leftarrow \text{SaturateSignedWordToUnsignedByte} \text{ SRC2}[95:80]; \\
\end{align*}
\]
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] ← 0;

VPACKUSWB (VEX.256 encoded version)
DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);

VPACKUSWB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
TMP_DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
TMP_DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
TMP_DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
TMP_DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] ← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:128] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToUnsignedByte (SRC1[383:368]);
    TMP_DEST[327:320] ← SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToUnsignedByte (SRC2[383:368]);
    TMP_DEST[391:384] ← SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToUnsignedByte (SRC1[511:496]);
    TMP_DEST[455:448] ← SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToUnsignedByte (SRC2[479:464]);
TMP_DEST[503:496] ← SaturateSignedWordToUnsignedByte (SRC2[495:480]);
TMP_DEST[511:504] ← SaturateSignedWordToUnsignedByte (SRC2[511:496]);

Fi;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
        FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPACKUSWB__m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m256i _mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB__m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);
PACKUSWB: __m64 __mm_packs_pu16(__m64 m1, __m64 m2)
(V)PACKUSWB: __m128i __mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB: __m256i __mm256_packus_epi16(__m256i m1, __m256i m2);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F FC /r</td>
<td>A</td>
<td>V/V MMX</td>
<td>Add packed byte integers from mm/m64 and mm.</td>
<td></td>
</tr>
<tr>
<td>PADDB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F FD /r</td>
<td>A</td>
<td>V/V MMX</td>
<td>Add packed word integers from mm/m64 and mm.</td>
<td></td>
</tr>
<tr>
<td>PADDW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F FE /r</td>
<td>A</td>
<td>V/V MMX</td>
<td>Add packed doubleword integers from mm/m64 and mm.</td>
<td></td>
</tr>
<tr>
<td>PADDD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F D4 /r</td>
<td>A</td>
<td>V/V MMX</td>
<td>Add packed quadword integers from mm/m64 and mm.</td>
<td></td>
</tr>
<tr>
<td>PADDDQ mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FC /r</td>
<td>A</td>
<td>V/V SSE2</td>
<td>Add packed byte integers from xmm2/m128 and xmm1.</td>
<td></td>
</tr>
<tr>
<td>PADDB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FD /r</td>
<td>A</td>
<td>V/V SSE2</td>
<td>Add packed word integers from xmm2/m128 and xmm1.</td>
<td></td>
</tr>
<tr>
<td>PADDW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FE /r</td>
<td>A</td>
<td>V/V SSE2</td>
<td>Add packed doubleword integers from xmm2/m128 and xmm1.</td>
<td></td>
</tr>
<tr>
<td>PADDD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F D4 /r</td>
<td>A</td>
<td>V/V SSE2</td>
<td>Add packed quadword integers from xmm2/m128 and xmm1.</td>
<td></td>
</tr>
<tr>
<td>PADDDQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG FC /r</td>
<td>B</td>
<td>V/V AVX</td>
<td>Add packed byte integers from xmm2, xmm3/m128 and store in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG FD /r</td>
<td>B</td>
<td>V/V AVX</td>
<td>Add packed word integers from xmm2, xmm3/m128 and store in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG FE /r</td>
<td>B</td>
<td>V/V AVX</td>
<td>Add packed doubleword integers from xmm2, xmm3/m128 and store in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG D4 /r</td>
<td>B</td>
<td>V/V AVX</td>
<td>Add packed quadword integers from xmm2, xmm3/m128 and store in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG FC /r</td>
<td>B</td>
<td>V/V AVX2</td>
<td>Add packed byte integers from ymm2, ymm3/m256 and store in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG FD /r</td>
<td>B</td>
<td>V/V AVX2</td>
<td>Add packed word integers from ymm2, ymm3/m256 and store in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG FE /r</td>
<td>B</td>
<td>V/V AVX2</td>
<td>Add packed doubleword integers from ymm2, ymm3/m256 and store in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG D4 /r</td>
<td>B</td>
<td>V/V AVX2</td>
<td>Add packed quadword integers from ymm2, ymm3/m256 and store in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VPADDQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG FC /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td>Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPADDB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG FD /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td>Add packed word integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPADDW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W0 FE /r</td>
<td>D</td>
<td>V/V AVX512VL AVX512F</td>
<td>Add packed doubleword integers from xmm2, and xmm3/m128/m32bcst and store in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPADDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 D4 /r</td>
<td>D</td>
<td>V/V AVX512VL AVX512F</td>
<td>Add packed quadword integers from xmm2, and xmm3/m128/m64bcst and store in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPADDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG FC /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td>Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPADDW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG FD /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td>Add packed word integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPADDQ ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.256.66.0F.W0 FE /r VPADDD ymm1 (k1)[r], ymm2, ymm3/m256/m32bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed doubleword integers from ymm2, ymm3/m256/m32bcst and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 D4 /r VPADDQ ymm1 (k1)[r], ymm2, ymm3/m256/m64bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed quadword integers from ymm2, ymm3/m256/m64bcst and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG FC /r VPADDB zmm1 (k1)[r], zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add packed byte integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG FD /r VPADDW zmm1 (k1)[r], zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add packed word integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WE /r VPADDD zmm1 (k1)[r], zmm2, zmm3/m512/m32bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Add packed doubleword integers from zmm2, zmm3/m512/m32bcst and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 D4 /r VPADDQ zmm1 (k1)[r], zmm2, zmm3/m512/m64bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Add packed quadword integers from zmm2, zmm3/m512/m64bcst and store in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too...
large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the
destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or
signed (two’s complement notation) packed integers; however, it does not set bits in the EFLAGS register to indi-
cate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of
values operated on.

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second source operand is a
ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a
32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the
writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a
ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM
register updated according to the writemask.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a
YMM register or a 256-bit memory location. The destination operand is a YMM register. the upper bits (MAXVL-1:256) of the
destination are cleared.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM
register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of
the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM
register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the
upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

**PADDB (with 64-bit operands)**

\[
\text{DEST}[7:0] \leftarrow \text{DEST}[7:0] + \text{SRC}[7:0];
\]

(* Repeat add operation for 2nd through 7th byte *)

\[
\text{DEST}[63:56] \leftarrow \text{DEST}[63:56] + \text{SRC}[63:56];
\]

**PADDW (with 64-bit operands)**

\[
\text{DEST}[15:0] \leftarrow \text{DEST}[15:0] + \text{SRC}[15:0];
\]

(* Repeat add operation for 2nd and 3th word *)

\[
\text{DEST}[63:48] \leftarrow \text{DEST}[63:48] + \text{SRC}[63:48];
\]

**PADDD (with 64-bit operands)**

\[
\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] + \text{SRC}[31:0];
\]

\[
\text{DEST}[63:32] \leftarrow \text{DEST}[63:32] + \text{SRC}[63:32];
\]

**PADDQ (with 64-Bit operands)**

\[
\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] + \text{SRC}[63:0];
\]

**PADDB (Legacy SSE instruction)**

\[
\text{DEST}[7:0] \leftarrow \text{DEST}[7:0] + \text{SRC}[7:0];
\]

(* Repeat add operation for 2nd through 15th byte *)

\[
\text{DEST}[127:120] \leftarrow \text{DEST}[127:120] + \text{SRC}[127:120];
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \text{ Unmodified}
\]

**PADDW (Legacy SSE instruction)**

\[
\text{DEST}[15:0] \leftarrow \text{DEST}[15:0] + \text{SRC}[15:0];
\]

(* Repeat add operation for 2nd through 7th word *)

\[
\text{DEST}[127:112] \leftarrow \text{DEST}[127:112] + \text{SRC}[127:112];
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \text{ Unmodified}
\]
PADDD (Legacy SSE instruction)
DEST[31:0] ← DEST[31:0] + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[MAXVL-1:128] (Unmodified)

PADDDQ (Legacy SSE instruction)
DEST[63:0] ← DEST[63:0] + SRC[63:0];
DEST[127:64] ← DEST[127:64] + SRC[127:64];
DEST[MAXVL-1:128] (Unmodified)

VPADDB (VEX.128 encoded instruction)
DEST[7:0] ← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120] ← SRC1[127:120] + SRC2[127:120];
DEST[MAXVL-1:128] ← 0;

VPADDW (VEX.128 encoded instruction)
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[MAXVL-1:128] ← 0;

VPADDD (VEX.128 encoded instruction)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[MAXVL-1:128] ← 0;

VPADDQ (VEX.128 encoded instruction)
DEST[63:0] ← SRC1[63:0] + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[MAXVL-1:128] ← 0;

VPADDB (VEX.256 encoded instruction)
DEST[7:0] ← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)

VPADDW (VEX.256 encoded instruction)
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)

VPADDD (VEX.256 encoded instruction)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)

VPADDQ (VEX.256 encoded instruction)
DEST[63:0] ← SRC1[63:0] + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
VPADDB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC1[i+7:i] + SRC2[i+7:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPADDW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC1[i+15:i] + SRC2[i+15:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPADDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
        ELSE DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
VPADDQ (EVEX encoded versions)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)
\(i \leftarrow j \times 64\)
IF \(k_1[j]\) OR *no writemask*
THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN \(\text{DEST}[i+63:j] \leftarrow \text{SRC1}[i+63:j] + \text{SRC2}[63:0]\)
ELSE \(\text{DEST}[i+63:j] \leftarrow \text{SRC1}[i+63:j] + \text{SRC2}[i+63:j]\)
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN \(\text{DEST}[i+63:j]\) remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
\(\text{DEST}[i+63:j] \leftarrow 0\)
FI
FI;
ENDFOR;
\(\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\)

Intel C/C++ Compiler Intrinsic Equivalents

VPADDB\_m512i\_mm512\_add\_epi8 (\_m512i a, \_m512i b)
VPADDD\_m512i\_mm512\_add\_epi8 (\_m512i a, \_m512i b)
VPADDB\_m512i\_mm512\_mask\_add\_epi8 (\_m512i s, \_mmask64 m, \_m512i a, \_m512i b)
VPADDD\_m512i\_mm512\_mask\_add\_epi8 (\_m512i s, \_mmask64 m, \_m512i a, \_m512i b)
VPADDB\_m512i\_mm512\_maskz\_add\_epi16 (\_mmask32 m, \_m512i a, \_m512i b)
VPADDD\_m512i\_mm512\_maskz\_add\_epi16 (\_mmask32 m, \_m512i a, \_m512i b)
VPADDB\_m256i\_mm256\_mask\_add\_epi8 (\_m256i s, \_mmask32 m, \_m256i a, \_m256i b)
VPADDD\_m256i\_mm256\_mask\_add\_epi8 (\_m256i s, \_mmask32 m, \_m256i a, \_m256i b)
VPADDB\_m256i\_mm256\_maskz\_add\_epi16 (\_mmask16 m, \_m256i a, \_m256i b)
VPADDD\_m256i\_mm256\_maskz\_add\_epi16 (\_mmask16 m, \_m256i a, \_m256i b)
VPADDB\_m128i\_mm128\_mask\_add\_epi8 (\_m128i s, \_mmask16 m, \_m128i a, \_m128i b)
VPADDD\_m128i\_mm128\_mask\_add\_epi8 (\_m128i s, \_mmask16 m, \_m128i a, \_m128i b)
VPADDD\_m128i\_mm128\_mask\_add\_epi16 (\_mmask8 m, \_m128i a, \_m128i b)
VPADDD\_m128i\_mm128\_maskz\_add\_epi16 (\_mmask8 m, \_m128i a, \_m128i b)
VPADDB\_m128i\_mm128\_mask\_add\_epi32 (\_m128i s, \_mmask8 k, \_m128i a, \_m128i b)
VPADDD\_m128i\_mm128\_mask\_add\_epi32 (\_m128i s, \_mmask8 k, \_m128i a, \_m128i b)
VPADDB\_m128i\_mm128\_maskz\_add\_epi32 (\_mmask4 k, \_m128i a, \_m128i b)
VPADDD\_m128i\_mm128\_maskz\_add\_epi32 (\_mmask4 k, \_m128i a, \_m128i b)
VPADDB\_m256i\_mm256\_mask\_add\_epi32 (\_m256i s, \_mmask4 k, \_m256i a, \_m256i b)
VPADDD\_m256i\_mm256\_mask\_add\_epi32 (\_m256i s, \_mmask4 k, \_m256i a, \_m256i b)
VPADDB\_m256i\_mm256\_maskz\_add\_epi32 (\_mmask2 k, \_m256i a, \_m256i b)
VPADDD\_m256i\_mm256\_maskz\_add\_epi32 (\_mmask2 k, \_m256i a, \_m256i b)
VPADDB\_m512i\_mm512\_mask\_add\_epi64 (\_m512i s, \_mmask2 k, \_m512i a, \_m512i b)
VPADDD\_m512i\_mm512\_mask\_add\_epi64 (\_m512i s, \_mmask2 k, \_m512i a, \_m512i b)
VPADDB\_m512i\_mm512\_maskz\_add\_epi64 (\_mmask1 k, \_m512i a, \_m512i b)
VPADDD\_m512i\_mm512\_maskz\_add\_epi64 (\_mmask1 k, \_m512i a, \_m512i b)
VPADDB\_m512i\_mm512\_mask\_add\_epi64 (\_m512i s, \_mmask1 k, \_m512i a, \_m512i b)
VPADDD\_m512i\_mm512\_mask\_add\_epi64 (\_m512i s, \_mmask1 k, \_m512i a, \_m512i b)
VPADDB\_m512i\_mm512\_maskz\_add\_epi64 (\_mmask\_m512i a, \_m512i b)
VPADDD\_m512i\_mm512\_maskz\_add\_epi64 (\_mmask\_m512i a, \_m512i b)
VPADDB\_m256i\_mm256\_mask\_add\_epi64 (\_m256i s, \_mmask\_m256i a, \_m256i b)
VPADDD\_m256i\_mm256\_mask\_add\_epi64 (\_m256i s, \_mmask\_m256i a, \_m256i b)
VPADDB\_m128i\_mm128\_mask\_add\_epi64 (\_m128i s, \_mmask\_m128i a, \_m128i b)
VPADDD\_m128i\_mm128\_mask\_add\_epi64 (\_m128i s, \_mmask\_m128i a, \_m128i b)
PADDB\_m128i\_mm_add\_epi8 (\_m128i a, \_m128i b)
PADDW\_m128i\_mm_add\_epi16 (\_m128i a, \_m128i b)
PADDQ\_m128i\_mm_add\_epi64 (\_m128i a, \_m128i b)
VPADDB __m256i _mm256_add_epi8 ( __m256ia, __m256i b );  
VPADDW __m256i _mm256_add_epi16 ( __m256i a, __m256i b);  
VPADD __m256i _mm256_add_epi32 ( __m256i a, __m256i b);  
VPADDQ __m256i _mm256_add_epi64 ( __m256i a, __m256i b);  
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)  
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)  
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)  
PADDQ __m64 _mm_add_si64(__m64 m1, __m64 m2)

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 4.  
EVEX-encoded VPADD/Q, see Exceptions Type E4.  
EVEX-encoded VPADDB/W, see Exceptions Type E4.nb.
PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EC /r</td>
<td>A</td>
<td>V/V MMX</td>
<td></td>
<td>Add packed signed byte integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>PADDSB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F EC /r</td>
<td>A</td>
<td>V/V SSE2</td>
<td></td>
<td>Add packed signed byte integers from xmm2/m128 and xmm1 saturate the results.</td>
</tr>
<tr>
<td>PADDSB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F ED /r</td>
<td>A</td>
<td>V/V MMX</td>
<td></td>
<td>Add packed signed word integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>PADDSW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F ED /r</td>
<td>A</td>
<td>V/V SSE2</td>
<td></td>
<td>Add packed signed word integers from xmm2/m128 and xmm1 and saturate the results.</td>
</tr>
<tr>
<td>PADDSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG EC /r</td>
<td>B</td>
<td>V/V AVX</td>
<td></td>
<td>Add packed signed byte integers from xmm3/m128 and xmm2 saturate the results.</td>
</tr>
<tr>
<td>VPADDSB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:Wig EC /r</td>
<td>B</td>
<td>V/V AVX2</td>
<td></td>
<td>Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.</td>
</tr>
<tr>
<td>VPADDSW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG EC /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td></td>
<td>Add packed signed byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPADDSB xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:Wig EC /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td></td>
<td>Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPADDSW ymm1 {k1}[z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG EC /r</td>
<td>C</td>
<td>V/V AVX512BW</td>
<td></td>
<td>Add packed signed byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPADDSB zmm1 {k1}[z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG ED /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td></td>
<td>Add packed signed word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPADDSW xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:Wig ED /r</td>
<td>C</td>
<td>V/V AVX512VL AVX512BW</td>
<td></td>
<td>Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPADDSW ymm1 [k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG ED /r</td>
<td>C</td>
<td>V/V AVX512BW</td>
<td></td>
<td>Add packed signed word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPADDSW zmm1 [k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
**Instruction Set Reference, M-U**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

**(V)PADDSB** performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

**(V)PADDSW** performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation**

**PADDSB (with 64-bit operands)**

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0]); \\
\text{(Repeat add operation for 2nd through 7th bytes)} \\
\text{DEST}[63:56] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[63:56] + \text{SRC}[63:56]);
\]

**PADDSB (with 128-bit operands)**

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0]); \\
\text{(Repeat add operation for 2nd through 14th bytes)} \\
\text{DEST}[127:120] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[111:120] + \text{SRC}[127:120]);
\]

**VPADDSB (VEX.128 encoded version)**

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToSignedByte}(\text{SRC1}[7:0] + \text{SRC2}[7:0]); \\
\text{(Repeat subtract operation for 2nd through 14th bytes)} \\
\text{DEST}[127:120] \leftarrow \text{SaturateToSignedByte}(\text{SRC1}[111:120] + \text{SRC2}[127:120]); \\
\text{DEST}[\text{MAXVL-1:128}] \leftarrow 0
\]

**VPADDSB (VEX.256 encoded version)**

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToSignedByte}(\text{SRC1}[7:0] + \text{SRC2}[7:0]); \\
\text{(Repeat add operation for 2nd through 31st bytes)} \\
\text{DEST}[255:248] \leftarrow \text{SaturateToSignedByte}(\text{SRC1}[255:248] + \text{SRC2}[255:248]);
\]
VPADDSB (EVEX encoded versions)

KL, VL = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

PADDSSW (with 64-bit operands)

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 7th words *)

PADDSSW (with 128-bit operands)

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)

VPADDSSW (VEX.128 encoded version)

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] ← 0

VPADDSSW (VEX.256 encoded version)

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)

VPADDSSW (EVEX encoded versions)

KL, VL = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalents

PADDSB: __m64 _mm_adds_pi8(__m64 m1, __m64 m2)
(V)PADDSB: __m128i _mm_adds_epi8 ( __m128i a, __m128i b)
VPADDSB: __m256i _mm256_adds_epi8 ( __m256i a, __m256i b)
PADDSW: __m64 _mm_adds_pi16(__m64 m1, __m64 m2)
(V)PADDSW: __m128i _mm_adds_epi16 ( __m128i a, __m128i b)
VPADDSW: __m256i _mm256_adds_epi16 ( __m256i a, __m256i b)

VPADDSB __m512i _mm512_adds_epi8 ( __m512i a, __m512i b)
VPADDSW __m512i _mm512_adds_epi16 ( __m512i a, __m512i b)
VPADDSB __m512i _mm512_mask_adds_epi8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_mask_adds_epi16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB __m512i _mm512_maskz_adds_epi8 ( __mmask64 m, __m512i a, __m512i b)
VPADDSW __m512i _mm512_maskz_adds_epi16 ( __mmask32 m, __m512i a, __m512i b)

VPADDSB __m256i _mm256_mask_adds_epi8 ( __m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_mask_adds_epi16 ( __m256i s, __mmask8 m, __m256i a, __m256i b)
VPADDSB __m256i _mm256_maskz_adds_epi8 ( __mmask16 m, __m256i a, __m256i b)
VPADDSW __m256i _mm256_maskz_adds_epi16 ( __mmask8 m, __m256i a, __m256i b)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F DC /r1 PADDUSB mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Add packed unsigned byte integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F DC /r PADDUSB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results.</td>
</tr>
<tr>
<td>NP 0F DD /r1 PADDUSW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Add packed unsigned word integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F DD /r PADDUSW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results.</td>
</tr>
<tr>
<td>VEX.NDS.128.660F.WIG DC /r VPADDUSB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed unsigned byte integers from xmm3/m128 to xmm2 and saturate the results.</td>
</tr>
<tr>
<td>VEX.NDS.128.660F.WIG DD /r VPADDUSW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results.</td>
</tr>
<tr>
<td>VEX.NDS.256.660F.WIG DC /r VPADDUSB ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.660F.WIG DD /r VPADDUSW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.660F.WIG DC /r VPADDUSB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.660F.WIG DC /r VPADDUSB ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.660F.WIG DC /r VPADDUSB zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add packed unsigned byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.660F.WIG DD /r VPADDUSW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.660F.WIG DD /r VPADDUSW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
</tbody>
</table>
**PADDUSB/PADDUSW**—Add Packed Unsigned Integers with Unsigned Saturation

**Instruction Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(P)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(P)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is a ZMM/YMM/XMM register.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

**128-bit Legacy SSE version:** The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation**

**PADDUSB (with 64-bit operands)**

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0]);
\]

(* Repeat add operation for 2nd through 7th bytes *)

\[
\text{DEST}[63:56] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[63:56] + \text{SRC}[63:56])
\]

**PADDUSB (with 128-bit operands)**

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0]);
\]

(* Repeat add operation for 2nd through 14th bytes *)

\[
\text{DEST}[127:120] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[127:120] + \text{SRC}[127:120]);
\]

**NOTES:**

VPADDUSB (VEX.128 encoded version)
DEST[7:0] ← SaturateToUnsignedByte(SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte(SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] ← 0

VPADDUSB (VEX.256 encoded version)
DEST[7:0] ← SaturateToUnsignedByte(SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)

PADDUSW (with 64-bit operands)
DEST[15:0] ← SaturateToUnsignedWord(Dest[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 3rd words *)

PADDUSW (with 128-bit operands)
DEST[15:0] ← SaturateToUnsignedWord(Dest[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)

VPADDUSW (VEX.128 encoded version)
DEST[15:0] ← SaturateToUnsignedWord(SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] ← 0

VPADDUSW (VEX.256 encoded version)
DEST[15:0] ← SaturateToUnsignedWord(SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)

VPADDUSB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
i ← j * 8
IF k1[j] OR "no writemask"
THEN DEST[i+7:i] ← SaturateToUnsignedByte(SRC1[i+7:i] + SRC2[i+7:i])
ELSE
IF "merging-masking" ; merging-masking
THEN "DEST[i+7:i] remains unchanged"
ELSE "zeroing-masking" ; zeroing-masking
DEST[i+7:i] = 0
FI
Fi;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
VPADDUSw (EVEX encoded versions)  
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1  
i ← j * 16  
IF k1[j] OR *no writemask*  
THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])  
ELSE  
IF *merging-masking*  
THEN *DEST[i+15:i] remains unchanged*  
ELSE *zeroing-masking*  
DEST[i+15:i] = 0  
FI  
ENDFOR;  
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB: __m64 _mm_adds_pu8(__m64 m1, __m64 m2)  
PADDUSW: __m64 _mm_adds_pu16(__m64 m1, __m64 m2)  
(V)PADDUSB: __m128i _mm_adds_epu8 ( __m128i a, __m128i b)  
(V)PADDUSW: __m128i _mm_adds_epu16 ( __m128i a, __m128i b)  
VPADDUSB: __m256i _mm256_adds_epu8 ( __m256i a, __m256i b)  
VPADDUSW: __m256i _mm256_adds_epu16 ( __m256i a, __m256i b)  
VPADDUSB__m512i _mm512_adds_epu8 ( __m512i a, __m512i b)  
VPADDUSW__m512i _mm512_adds_epu16 ( __m512i a, __m512i b)  
VPADDUSB__m512i _mm512_mask_adds_epu8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)  
VPADDUSW__m512i _mm512_mask_adds_epu16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)  
VPADDUSB__m512i _mm512_maskz_adds_epu8 ( __mmask64 m, __m512i a, __m512i b)  
VPADDUSW__m512i _mm512_maskz_adds_epu16 ( __mmask32 m, __m512i a, __m512i b)  
VPADDUSB__m256i _mm256_mask_adds_epu8 ( __mmask32 m, __m256i a, __m256i b)  
VPADDUSW__m256i _mm256_mask_adds_epu16 ( __mmask16 m, __m256i a, __m256i b)  
VPADDUSB__m256i _mm256_maskz_adds_epu8 ( __mmask32 m, __m256i a, __m256i b)  
VPADDUSW__m256i _mm256_maskz_adds_epu16 ( __mmask16 m, __m256i a, __m256i b)  
VPADDUSB__m128i _mm_mask_adds_epu8 ( __m128i s, __mmask16 m, __m128i a, __m128i b)  
VPADDUSW__m128i _mm_mask_adds_epu16 ( __m128i s, __mmask8 m, __m128i a, __m128i b)  
VPADDUSB__m128i _mm_maskz_adds_epu8 ( __mmask16 m, __m128i a, __m128i b)  
VPADDUSW__m128i _mm_maskz_adds_epu16 ( __mmask8 m, __m128i a, __m128i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.  
EVEX-encoded instruction, see Exceptions Type E4.nb.
## PALIGNR — Packed Align Right

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 3A 0F /r ib²</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into mm1.</td>
</tr>
<tr>
<td>66 0F 3A 0F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.WIG 0F /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Concatenate xmm2 and xmm3/m128, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F3A.WIG 0F /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant value in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.WIG 0F /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Concatenatexmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.WIG 0F /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant value in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.WIG 0F /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant value in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.</td>
</tr>
</tbody>
</table>

### Notes:

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX,
XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e., 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Figure 4-7. 256-bit VPALIGN Instruction Operation

Operation

\[
\text{PALIGNR (with 64-bit operands)}
\]

\[
temp1[127:0] = \text{CONCATENATE(DEST, SRC)} >> (\text{imm8*8})
\]

\[
\text{DEST}[63:0] = temp1[63:0]
\]
PALIGNR (with 128-bit operands)

\[ \text{temp1}[255:0] \leftarrow ((\text{DEST}[127:0] \ll 128) \lor \text{SRC}[127:0]) \gg (\text{imm8} \times 8); \]
\[ \text{DEST}[127:0] \leftarrow \text{temp1}[127:0]; \]
\[ \text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)} \]

VPALIGNR (VEX.128 encoded version)

\[ \text{temp1}[255:0] \leftarrow ((\text{SRC1}[127:0] \ll 128) \lor \text{SRC2}[127:0]) \gg (\text{imm8}[7:0] \times 8); \]
\[ \text{DEST}[127:0] \leftarrow \text{temp1}[127:0]; \]
\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

VPALIGNR (VEX.256 encoded version)

\[ \text{temp1}[255:0] \leftarrow ((\text{SRC1}[127:0] \ll 128) \lor \text{SRC2}[127:0]) \gg (\text{imm8}[7:0] \times 8); \]
\[ \text{DEST}[127:0] \leftarrow \text{temp1}[127:0]; \]
\[ \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \]

VPALIGNR (EVEX encoded versions)

\((KL, VL) = (16, 128), (32, 256), (64, 512)\)

\[ \text{FOR } l \leftarrow 0 \text{ TO } VL-1 \text{ with increments of 128} \]
\[ \text{temp1}[255:0] \leftarrow ((\text{SRC1}[l+127:l] \ll 128) \lor \text{SRC2}[l+127:l]) \gg (\text{imm8}[7:0] \times 8); \]
\[ \text{TMP \_DEST}[l+127:l] \leftarrow \text{temp1}[127:0]; \]
ENDFOR;

\[ \text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]
\[ i \leftarrow j \times 8 \]
\[ \text{IF } k[1][j] \text{ OR *no writemask*} \]
\[ \quad \text{THEN DEST}[i+7:i] \leftarrow \text{TMP \_DEST}[i+7:i]; \]
\[ \quad \text{ELSE} \]
\[ \quad \quad \text{IF *merging-masking* ; merging-masking} \]
\[ \quad \quad \quad \text{THEN *DEST}[i+7:i] \text{ remains unchanged*} \]
\[ \quad \quad \quad \text{ELSE *zeroing-masking* ; zeroing-masking} \]
\[ \quad \quad \quad \quad \text{DEST}[i+7:i] = 0 \]
\[ \quad \text{FI} \]
\[ \quad \text{FI}; \]
ENDFOR;
\[ \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \]

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR: \_m64 \_mm\_alignr\_pi8 (\_m64 a, \_m64 b, int n)

(V)PALIGNR: \_m128i \_mm\_alignr\_epi8 (\_m128i a, \_m128i b, int n)

VPALIGNR: \_m256i \_mm256\_alignr\_epi8 (\_m256i a, \_m256i b, const int n)

VPALIGNR \_m512i \_mm512\_alignr\_epi8 (\_m512i a, \_m512i b, const int n)

VPALIGNR \_m512i \_mm512\_mask\_alignr\_epi8 (\_m512i s, \_mm\_mask64 m, \_m512i a, \_m512i b, const int n)

VPALIGNR \_m512i \_mm512\_maskz\_alignr\_epi8 (\_m512i s, \_mm\_mask64 m, \_m512i a, \_m512i b, const int n)

VPALIGNR \_m256i \_mm256\_mask\_alignr\_epi8 (\_m256i s, \_mm\_mask32 m, \_m256i a, \_m256i b, const int n)

VPALIGNR \_m256i \_mm256\_maskz\_alignr\_epi8 (\_m256i s, \_mm\_mask32 m, \_m256i a, \_m256i b, const int n)

VPALIGNR \_m128i \_mm\_mask\_alignr\_epi8 (\_m128i s, \_mm\_mask16 m, \_m128i a, \_m128i b, const int n)

VPALIGNR \_m128i \_mm\_maskz\_alignr\_epi8 (\_m128i s, \_mm\_mask16 m, \_m128i a, \_m128i b, const int n)

SIMD Floating-Point Exceptions

None.
**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
PAND—Logical AND

PAND—Logical AND

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**Operation**

**PAND (64-bit operand)**
DEST ← DEST AND SRC

**PAND (128-bit Legacy SSE version)**
DEST ← DEST AND SRC
DEST[MAXVL-1:128] (Unmodified)

**VPAND (VEX.128 encoded version)**
DEST ← SRC1 AND SRC2
DEST[MAXVL-1:128] ← 0

**VPAND (VEX.256 encoded instruction)**
DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])
DEST[MAXVL-1:256] ← 0

**VPANDDD (EVEX encoded versions)**
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]
        ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPANDQ (EVEX encoded versions)

KL, VL = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

VPANDD __m512i _mm512_and_epi32( __m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_and_epi64( __m512i a, __m512i b);
VPANDQ __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_maskz_and_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64( __mmask8 k, __m256i a, __m256i b);

PAND: __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND: __m128i _mm_and_si128 ( __m128i a, __m128i b)
VPAND: __m256i _mm256_and_si256 ( __m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
PANDN—Logical AND NOT

### Instruction Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMrm/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

**PANDN (64-bit operand)**

\[ \text{DEST} \leftarrow \text{NOT(DEST) AND SRC} \]

**PANDN (128-bit Legacy SSE version)**

\[ \text{DEST} \leftarrow \text{NOT(DEST) AND SRC} \]
\[ \text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)} \]

**VPANDN (VEX.128 encoded version)**

\[ \text{DEST} \leftarrow \text{NOT(SRC1) AND SRC2} \]
\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

**VPANDN (VEX.256 encoded instruction)**

\[ \text{DEST}[255:0] \leftarrow (\text{NOT SRC1}[255:0] \text{ AND SRC2}[255:0]) \]
\[ \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \]

**VPANDND (EVEX encoded versions)**

\( (KL, VL) = (4, 128), (8, 256), (16, 512) \)

\[ \text{FOR } j \leftarrow 0 \text{ TO KL-1} \]
\[ \quad i \leftarrow j \times 32 \]
\[ \quad \text{IF } k1[j] \text{ OR } *\text{no writemask}* \]
\[ \quad \text{THEN} \]
\[ \quad \quad \text{IF } (\text{EVEX.b} = 1) \text{ AND (SRC2 } *\text{is memory}*) \]
\[ \quad \quad \quad \text{THEN } \text{DEST}[i+31:j] \leftarrow (\text{NOT SRC1}[i+31:j] \text{ AND SRC2}[31:0]) \]
\[ \quad \quad \quad \text{ELSE } \text{DEST}[i+31:j] \leftarrow (\text{NOT SRC1}[i+31:j] \text{ AND SRC2}[i+31:j]) \]
\[ \quad \quad \text{FI;} \]
\[ \quad \quad \text{ELSE} \]
\[ \quad \quad \text{IF } *\text{merging-masking}* \quad ; \text{merging-masking} \]
\[ \quad \quad \quad \text{THEN } *\text{DEST}[i+31:j] \text{ remains unchanged}*_ \]
\[ \quad \quad \quad \text{ELSE} \quad ; \text{zeroing-masking} \]
\[ \quad \quad \quad \quad \text{DEST}[i+31:j] \leftarrow 0 \]
\[ \quad \quad \text{FI} \]
\[ \quad \text{FI} \]
\[ \text{ENDFOR} \]
\[ \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \]
VPANDNQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[63:0])
                ELSE DEST[i+63:i] ← (NOT SRC1[i+63:i]) AND SRC2[i+63:j])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:j] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents
VPANDND __m512i _mm512_andnot_epi32( __m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32( __m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_andnot_epi64( __m256i a, __m256i b);
VPANDND __m256i _mm256_mask_andnot_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm128_andnot_epi128( __m128i a, __m128i b);
VPANDND __m128i _mm128_mask_andnot_epi128( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm128_maskz_andnot_epi128( __mmask8 k, __m128i a, __m128i b);
VPANDN: __m64 _mm_andnot_si64( __m64 m1, __m64 m2)
(V)PANDN: __m128i _mm_andnot_si128( __m128i a, __m128i b)
VPANDN: __m256i _mm256_andnot_si256( __m256i a, __m256i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
PAUSE—Spin Loop Hint

Description
Improves the performance of spin-wait loops. When executing a “spin-wait loop,” processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
Execute_Next_Instruction(DELAY);

Numeric Exceptions
None.

Exceptions (All Operating Modes)
#UD If the LOCK prefix is used.
## PAVGB/PAVGW—Average Packed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E0 /r(^1)</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.</td>
</tr>
<tr>
<td>PAVGB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E0, /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.</td>
</tr>
<tr>
<td>PAVGB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F E3 /r(^1)</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Average packed unsigned word integers from mm2/m64 and mm1 with rounding.</td>
</tr>
<tr>
<td>PAVGW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E3 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.</td>
</tr>
<tr>
<td>PAVGW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG E0 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding.</td>
</tr>
<tr>
<td>VPAVGB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG E3 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding.</td>
</tr>
<tr>
<td>VPAVGW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG E0 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1.</td>
</tr>
<tr>
<td>VPAVGB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG E3 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Average packed unsigned word integers from ymm2, and ymm3/m256 with rounding and store to ymm1.</td>
</tr>
<tr>
<td>VPAVGW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG E0 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned byte integers from xmm2, and xmm3/m128 with rounding and store to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPVAVGB ymm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG E0 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPVAVGB ymm1 [k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG E0 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Average packed unsigned byte integers from zmm2, and zmm3/m512 with rounding and store to zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPVAVGB zmm1 [k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG E3 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned word integers from xmm2, xmm3/m128 with rounding to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPVAVGB xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG E3 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPVAVGB ymm1 [k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG E3 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Average packed unsigned word integers from zmm2, zmm3/m512 with rounding to zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPVAVGB zmm1 [k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:


---

INSTRUCTION SET REFERENCE, M-U

4-230 Vol. 2B

PAVGB/PAVGW—Average Packed Integers
Perform a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

**Operation**

**PAVGB (with 64-bit operands)**

\[
\text{DEST}[7:0] \leftarrow (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) >> 1; (* \text{Temp sum before shifting is 9 bits } *)
\]

\[
(*) \text{Repeat operation performed for bytes 2 through 6 } *
\]

\[
\text{DEST}[63:56] \leftarrow (\text{SRC}[63:56] + \text{DEST}[63:56] + 1) >> 1;
\]

**PAVGW (with 64-bit operands)**

\[
\text{DEST}[15:0] \leftarrow (\text{SRC}[15:0] + \text{DEST}[15:0] + 1) >> 1; (* \text{Temp sum before shifting is 17 bits } *)
\]

\[
(*) \text{Repeat operation performed for words 2 and 3 } *
\]

\[
\text{DEST}[63:48] \leftarrow (\text{SRC}[63:48] + \text{DEST}[63:48] + 1) >> 1;
\]

**PAVGB (with 128-bit operands)**

\[
\text{DEST}[7:0] \leftarrow (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) >> 1; (* \text{Temp sum before shifting is 9 bits } *)
\]

\[
(*) \text{Repeat operation performed for bytes 2 through 14 } *
\]

\[
\text{DEST}[127:120] \leftarrow (\text{SRC}[127:120] + \text{DEST}[127:120] + 1) >> 1;
\]

**PAVGW (with 128-bit operands)**

\[
\text{DEST}[15:0] \leftarrow (\text{SRC}[15:0] + \text{DEST}[15:0] + 1) >> 1; (* \text{Temp sum before shifting is 17 bits } *)
\]

\[
(*) \text{Repeat operation performed for words 2 through 6 } *
\]

\[
\text{DEST}[127:112] \leftarrow (\text{SRC}[127:112] + \text{DEST}[127:112] + 1) >> 1;
\]
VPAVGB (VEX.128 encoded version)
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[MAXVL-1:128] ← 0

VPAVGW (VEX.128 encoded version)
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[MAXVL-1:128] ← 0

VPAVGB (VEX.256 encoded instruction)
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 31)

VPAVGW (VEX.256 encoded instruction)
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 15)

VPAVGB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
i ← j * 8
IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] = 0
        FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPAVGW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
i ← j * 16
IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] = 0
        FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGB __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGW __m256i _mm256_avg_epu8(__m256i a, __m256i b);
VPAVGW __m256i _mm256_avg_epu16(__m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGW __m128i _mm128_avg_epu8(__m128i a, __m128i b);
VPAVGW __m128i _mm128_avg_epu16(__m128i a, __m128i b);
VPAVGW __m128i _mm128_mask_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm128_mask_avg_epu16(__mmask8 m, __m128i a, __m128i b);
VPAVGW __m128i _mm128_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm128_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);
PAPVGB: __m64 _mm_avg_pu8 (__m64 a, __m64 b)
PAPGw: __m64 _mm_avg_pu16 (__m64 a, __m64 b)
(V)PAPVGB: __m128i _mm_avg_epu8 (__m128i a, __m128i b)
(V)PAPVGB: __m128i _mm_avg_epu16 (__m128i a, __m128i b)
PAPVGW: __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
PAPVGW: __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
PBLENDVB — Variable Blend Packed Bytes

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>&lt;XMM0&gt;</td>
<td>NA</td>
</tr>
<tr>
<td>RVMR</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8[7:4]</td>
</tr>
</tbody>
</table>

### Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, XMM0. The mask bits are the most significant bit in each byte element of the XMM0 register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source operand is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

### Operation

**PBLENDVB (128-bit Legacy SSE version)**

\[
\text{MASK} \leftarrow \text{XMM0} \\
\text{IF (MASK}[7] = 1) \text{ THEN } \text{DEST}[7:0] \leftarrow \text{SRC}[7:0]; \\
\text{ELSE } \text{DEST}[7:0] \leftarrow \text{DEST}[7:0]; \\
\text{IF (MASK}[15] = 1) \text{ THEN } \text{DEST}[15:8] \leftarrow \text{SRC}[15:8];
\]
ELSE DEST[15:8] ← DEST[15:8];
ELSE DEST[23:16] ← DEST[23:16];
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ← DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ← DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120]);
DEST[MAXVL-1:128] (Unmodified)

VPBLENDVB (VEX.128 encoded version)
MASK ← SRC3
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
ELSE DEST[15:8] ← SRC1[15:8];
ELSE DEST[23:16] ← SRC1[23:16];
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC2[95:88];
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC2[103:96];
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC2[111:104];
ELSE DEST[111:104] ← SRC1[111:104];
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC2[127:120];
ELSE DEST[127:120] ← SRC1[127:120];
DEST[MAXVL-1:128] ← 0

VPBLENDVB (VEX.256 encoded version)
MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
ELSE DEST[15:8] ← SRC1[15:8];
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24];
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40];
ELSE DEST[47:40] ← SRC1[47:40];
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC2[63:56];
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC2[71:64];
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72];
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80];
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88];
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96];
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104];
ELSE DEST[111:104] ← SRC1[111:104];
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120];
ELSE DEST[127:120] ← SRC1[127:120];
IF (MASK[143] == 1) THEN DEST[143:136] ← SRC2[143:136];
ELSE DEST[143:136] ← SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] ← SRC2[151:144];
ELSE DEST[151:144] ← SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] ← SRC2[159:152];
ELSE DEST[159:152] ← SRC1[159:152];
ELSE DEST[167:160] ← SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] ← SRC2[175:168]
ELSE DEST[175:168] ← SRC1[175:168];
ELSE DEST[183:176] ← SRC1[183:176];
ELSE DEST[191:184] ← SRC1[191:184];
ELSE DEST[199:192] ← SRC1[199:192];
ELSE DEST[207:200] ← SRC1[207:200];
ELSE DEST[215:208] ← SRC1[215:208];
ELSE DEST[223:216] ← SRC1[223:216];
ELSE DEST[231:224] ← SRC1[231:224];
ELSE DEST[239:232] ← SRC1[239:232];
ELSE DEST[247:240] ← SRC1[247:240];
ELSE DEST[255:248] ← SRC1[255:248];

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDVB: __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);
VPBLENDVB: __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 4; additionally
#UD If VEX.W = 1.
PBLENDW — Blend Packed Words

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word element in the destination operand is unchanged.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

**PBLENDW (128-bit Legacy SSE version)**

IF (imm8[0] = 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
ELSE DEST[31:16] ← DEST[31:16]
ELSE DEST[63:48] ← DEST[63:48]
ELSE DEST[79:64] ← DEST[79:64]
ELSE DEST[95:80] ← DEST[95:80]
ELSE DEST[111:96] ← DEST[111:96]

**VPBLENDW (VEX.128 encoded version)**

IF (imm8[0] = 1) THEN DEST[15:0] <- SRC2[15:0]
ELSE DEST[15:0] <- SRC1[15:0]
ELSE DEST[31:16] <- SRC1[31:16]
ELSE DEST[63:48] <- SRC1[63:48]
ELSE DEST[79:64] <- SRC1[79:64]
ELSE DEST[95:80] <- SRC1[95:80]
ELSE DEST[111:96] <- SRC1[111:96]

DEST[MAXVL-1:128] <- 0

**VPBLENDW (VEX.256 encoded version)**

IF (imm8[0] == 1) THEN DEST[15:0] <- SRC2[15:0]
ELSE DEST[15:0] <- SRC1[15:0]
ELSE DEST[31:16] <- SRC1[31:16]
ELSE DEST[63:48] <- SRC1[63:48]
ELSE DEST[79:64] <- SRC1[79:64]
ELSE DEST[95:80] <- SRC1[95:80]
ELSE DEST[111:96] <- SRC1[111:96]

DEST[143:128] <- SRC2[143:128]
ELSE DEST[143:128] <- SRC1[143:128]
IF (imm8[0] == 1) THEN DEST[159:144] <- SRC2[159:144]
ELSE DEST[159:144] <- SRC1[159:144]
ELSE DEST[175:160] <- SRC1[175:160]
ELSE DEST[207:192] <- SRC1[207:192]
ELSE DEST[239:224] <- SRC1[239:224]
ELSE DEST[255:240] <- SRC1[255:240]
**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PBLENDW:  __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);
VPBLENDW:  __m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD  If VEX.L = 1 and AVX2 = 0.
PCLMULQDQ — Carry-Less Multiplication Quadword

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 44  lr ib</td>
<td>RMI</td>
<td>V/V</td>
<td>PCLMULQDQ</td>
<td>Carry-less multiplication of one quadword of xmm1 by one quadword of xmm2/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm1 and xmm2/m128 should be used.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F3A.IG 44  lr ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>Both PCLMULQDQ and AVX flags</td>
<td>Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-13, other bits of the immediate byte are ignored.

Table 4-13. PCLMULQDQ Quadword Selection of Immediate Byte

<table>
<thead>
<tr>
<th>Imm[4]</th>
<th>Imm[0]</th>
<th>PCLMULQDQ Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>CL_MUL( SRC2[63:0], SRC1[63:0])</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CL_MUL( SRC2[63:0], SRC1[127:64])</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CL_MUL( SRC2[127:64], SRC1[63:0])</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CL_MUL( SRC2[127:64], SRC1[127:64])</td>
</tr>
</tbody>
</table>

NOTES:
1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simply programming and emit the required encoding for Imm8.

Table 4-14. Pseudo-Op and PCLMULQDQ Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>Imm8 Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCLMULLQLQDQ xmm1, xmm2</td>
<td>0000_0000B</td>
</tr>
<tr>
<td>PCLMULHQLQDQ xmm1, xmm2</td>
<td>0000_0001B</td>
</tr>
<tr>
<td>PCLMULLQHQDQ xmm1, xmm2</td>
<td>0001_0000B</td>
</tr>
<tr>
<td>PCLMULHQQDQ xmm1, xmm2</td>
<td>0001_0001B</td>
</tr>
</tbody>
</table>
**Operation**

**PCLMULQDQ**

IF (Imm8[0] = 0)

THEN

\[ \text{TEMP1} \leftarrow \text{SRC1} \ [63:0]; \]
ELSE

\[ \text{TEMP1} \leftarrow \text{SRC1} \ [127:64]; \]

FI

IF (Imm8[4] = 0)

THEN

\[ \text{TEMP2} \leftarrow \text{SRC2} \ [63:0]; \]
ELSE

\[ \text{TEMP2} \leftarrow \text{SRC2} \ [127:64]; \]

FI

For \( i = 0 \) to 63 {

\[ \text{TmpB} [\ i \] \leftarrow (\text{TEMP1} [\ 0 \] and \ \text{TEMP2} [\ i ]); \]

For \( j = 1 \) to \( i \) {

\[ \text{TmpB} [\ i \] \leftarrow \text{TmpB} [\ i \] \ xor (\text{TEMP1} [\ j \] and \ \text{TEMP2} [\ i - j ]); \]

}\n
\[ \text{DEST}[\ i \] \leftarrow \text{TmpB}[\ i ]; \]

}\n
For \( i = 64 \) to 126 {

\[ \text{TmpB} [\ i \] \leftarrow 0; \]

For \( j = i - 63 \) to 63 {

\[ \text{TmpB} [\ i \] \leftarrow \text{TmpB} [\ i \] \ xor (\text{TEMP1} [\ j \] and \ \text{TEMP2} [\ i - j ]); \]

}\n
\[ \text{DEST}[\ i \] \leftarrow \text{TmpB}[\ i ]; \]

}\n
\[ \text{DEST}[127] \leftarrow 0; \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow \text{Unmodified} \]

\[ \text{VPCLMULQDQ} \]

IF (Imm8[0] = 0)

THEN

\[ \text{TEMP1} \leftarrow \text{SRC1} \ [63:0]; \]
ELSE

\[ \text{TEMP1} \leftarrow \text{SRC1} \ [127:64]; \]

FI

IF (Imm8[4] = 0)

THEN

\[ \text{TEMP2} \leftarrow \text{SRC2} \ [63:0]; \]
ELSE

\[ \text{TEMP2} \leftarrow \text{SRC2} \ [127:64]; \]

FI

For \( i = 0 \) to 63 {

\[ \text{TmpB} [\ i \] \leftarrow (\text{TEMP1} [\ 0 \] and \ \text{TEMP2} [\ i ]); \]

For \( j = 1 \) to \( i \) {

\[ \text{TmpB} [\ i \] \leftarrow \text{TmpB} [\ i \] \ xor (\text{TEMP1} [\ j \] and \ \text{TEMP2} [\ i - j ]); \]

}\n
\[ \text{DEST}[\ i \] \leftarrow \text{TmpB}[\ i ]; \]

}\n
For \( i = 64 \) to 126 {

\[ \text{TmpB} [\ i \] \leftarrow 0; \]

For \( j = i - 63 \) to 63 {

\[ \text{TmpB} [\ i \] \leftarrow \text{TmpB} [\ i \] \ xor (\text{TEMP1} [\ j \] and \ \text{TEMP2} [\ i - j ]); \]

}\n
\[ \text{DEST}[\ i \] \leftarrow \text{TmpB}[\ i ]; \]
TmpB[i] ← TmpB[i] xor (TEMP1[j] and TEMP2[i - j])
}
DEST[i] ← TmpB[i];
}
DEST[MAXVL-1:127] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PCLMULQDQ: _mm_clmulepi64_si128 (__m128i, __m128i, const int)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4, additionally

#UD If VEX.L = 1.
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 74 /r*&lt;sup&gt;1&lt;/sup&gt; PCMPEQB mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed bytes in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 74 /r PCMPEQB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed bytes in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>NP 0F 75 /r*&lt;sup&gt;1&lt;/sup&gt; PCMPEQW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed words in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 75 /r PCMPEQW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed words in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>NP 0F 76 /r*&lt;sup&gt;1&lt;/sup&gt; PCMPEQD mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed doublewords in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 76 /r PCMPEQD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed doublewords in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed bytes in xmm3/m128 and xmm2 for equality.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed words in xmm3/m128 and xmm2 for equality.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed doublewords in xmm3/m128 and xmm2 for equality.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed bytes in ymm3/m256 and ymm2 for equality.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW ymm1, ymm2, ymm3 /m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed words in ymm3/m256 and ymm2 for equality.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD ymm1, ymm2, ymm3 /m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed doublewords in ymm3/m256 and ymm2 for equality.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W0 76 /r VPCMPEQB k1 {k2}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Equal between int32 vector xmm2 and int32 vector xmm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
</tbody>
</table>
Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.
VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

**PCMPEQB (with 64-bit operands)**

IF DEST[7:0] = SRC[7:0]
THEN DEST[7:0] ← FFH;  
ELSE DEST[7:0] ← 0; FI;

(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)

IF DEST[63:56] = SRC[63:56]
THEN DEST[63:56] ← FFH;
ELSE DEST[63:56] ← 0; FI;

**COMPARE_BYTES_EQUAL (SRC1, SRC2)**

IF SRC1[7:0] = SRC2[7:0]
THEN DEST[7:0] ← FFH;
ELSE DEST[7:0] ← 0; FI;

(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)

IF SRC1[127:120] = SRC2[127:120]
THEN DEST[127:120] ← FFH;
ELSE DEST[127:120] ← 0; FI;

**COMPARE_WORDS_EQUAL (SRC1, SRC2)**

IF SRC1[15:0] = SRC2[15:0]
THEN DEST[15:0] ← FFFFH;
ELSE DEST[15:0] ← 0; FI;

(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)

IF SRC1[127:112] = SRC2[127:112]
THEN DEST[127:112] ← FFFFH;
ELSE DEST[127:112] ← 0; FI;

**COMPARE_DWORDS_EQUAL (SRC1, SRC2)**

IF SRC1[31:0] = SRC2[31:0]
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 0; FI;

(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)

IF SRC1[127:96] = SRC2[127:96]
THEN DEST[127:96] ← FFFFFFFFH;
ELSE DEST[127:96] ← 0; FI;

**PCMPEQB (with 128-bit operands)**

DEST[127:0] ← COMPARE_BYTES_EQUAL(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
VPCMPEQB (VEX.128 encoded version)
DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0

VPCMPEQB (VEX.256 encoded version)
DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPEQB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k2[j] OR "no writemask"
    THEN
      /* signed comparison */
      CMP ← SRC1[i+7:i] == SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0; /* zeroing-masking only */
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

PCMPEQW (with 64-bit operands)
IF DEST[15:0] = SRC[15:0]
  THEN DEST[15:0] ← FFFFH;
ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] = SRC[63:48]
  THEN DEST[63:48] ← FFFFH;
ELSE DEST[63:48] ← 0; FI;

PCMPEQW (with 128-bit operands)
DEST[127:0] ← COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPEQW (VEX.128 encoded version)
DEST[127:0] ← COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0

VPCMPEQW (VEX.256 encoded version)
DEST[127:0] ← COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
VPCMPEQW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP ← SRC1[i+15:i] == SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

PCMPEQD (with 64-bit operands)
  IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;

PCMPEQD (with 128-bit operands)
DEST[127:0] ← COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPEQD (VEX.128 encoded version)
DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0

VPCMPEQD (VEX.256 encoded version)
DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPEQD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP ← SRC1[i+31:i] = SRC2[31:0];
        ELSE CMP ← SRC1[i+31:i] = SRC2[i+31:i];
      FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

4-248 Vol. 2B

PCMPEQW/PCMPEQD— Compare Packed Data for Equal
Intel C/C++ Compiler Intrinsic Equivalents

VPCMPEQB __mmask64 __m512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 __m512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 __mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 __mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 __mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 __mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 __mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 __mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 __mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 __mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 __mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 __mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 __mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 __mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 __mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 __mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 __mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 __mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPEQD, see Exceptions Type E4.
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb.
### Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.
Operation

PCMPEQQ (with 128-bit operands)

IF (DEST[63:0] = SRC[63:0])
   THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
   ELSE DEST[63:0] ← 0; FI;
IF (DEST[127:64] = SRC[127:64])
   THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
   ELSE DEST[127:64] ← 0; FI;
DEST[MAXVL-1:128] (Unmodified)

COMPARE_QWORDS_EQUAL (SRC1, SRC2)

IF SRC1[63:0] = SRC2[63:0]
   THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
   ELSE DEST[63:0] ← 0; FI;
IF SRC1[127:64] = SRC2[127:64]
   THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
   ELSE DEST[127:64] ← 0; FI;
DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1, SRC2)

VPCMPEQQ (VEX.128 encoded version)

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPCMPEQQ (VEX.256 encoded version)

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_EQUAL(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPEQQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
   i ← j * 64
   IF k2[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN CMP ← SRC1[i+63:i] = SRC2[63:0];
            ELSE CMP ← SRC1[i+63:i] = SRC2[i+63:i];
            FI;
         IF CMP = TRUE
            THEN DEST[j] ← 1;
            ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0 ; zeroing-masking only
   FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPEQQ __mmask8 __mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
VPCMPEQQ __mmask8 __mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 __mm256_cmpeq_epi64_mask(__m256i a, __m256i b);
VPCMPEQQ __mmask8 __mm256_mask_cmpeq_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 __mm_cmpeq_epi64_mask(__m128i a, __m128i b);
VPCMPEQQ __mmask8 __mm_mask_cmpeq_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ: __m128i __mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ: __m256i __mm256_cmpeq_epi64(__m256i a, __m256i b);

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPEQQ, see Exceptions Type E4.
PCMPESTRI — Packed Compare Explicit Length Strings, Return Index

**Description**

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to the count register (ECX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
- SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
- OFlag – IntRes2[0]
- AFlag – Reset
- PFlag – Reset

**Effective Operand Size**

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Length 1</th>
<th>Length 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>ECX</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>ECX</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>ECX</td>
</tr>
<tr>
<td>64 bit + REX.W</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>RAX</td>
<td>RDX</td>
<td>ECX</td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent For Returning Index**

```c
int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);
```
Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpstrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and #UD if VEX.L = 1.
If VEX.vvvv ≠ 1111B.
**PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 60 /r imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A 60 /ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The instruction compares data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
- SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
- OFlag – IntRes2[0]
- AFlag – Reset
- PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.
Effective Operand Size

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand1</th>
<th>Operand 2</th>
<th>Length1</th>
<th>Length2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>XMM0</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>XMM0</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>XMM0</td>
</tr>
<tr>
<td>64 bit + REX.W</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>RAX</td>
<td>RDX</td>
<td>XMM0</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

`__m128i __m128i_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);`

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

`int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);`
`int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);`
`int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);`
`int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);`
`int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD

If VEX.L = 1.
If VEX.vvvv ≠ 1111B.
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 64 /r(^1)</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed signed byte integers in (mm) and (mm/m64) for greater than.</td>
</tr>
<tr>
<td>PCMPGTB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 64 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed byte integers in (xmm1) and (xmm2/m128) for greater than.</td>
</tr>
<tr>
<td>PCMPGTB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 65 /r(^1)</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed signed word integers in (mm) and (mm/m64) for greater than.</td>
</tr>
<tr>
<td>PCMPGTW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 65 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed word integers in (xmm1) and (xmm2/m128) for greater than.</td>
</tr>
<tr>
<td>PCMPGTW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 66 /r(^1)</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed signed doubleword integers in (mm) and (mm/m64) for greater than.</td>
</tr>
<tr>
<td>PCMPGTD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 66 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed doubleword integers in (xmm1) and (xmm2/m128) for greater than.</td>
</tr>
<tr>
<td>PCMPGTD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 64 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed byte integers in (xmm2) and (xmm3/m128) for greater than.</td>
</tr>
<tr>
<td>VPCMPGTB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 65 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed word integers in (xmm2) and (xmm3/m128) for greater than.</td>
</tr>
<tr>
<td>VPCMPGTW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 66 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed doubleword integers in (xmm2) and (xmm3/m128) for greater than.</td>
</tr>
<tr>
<td>VPCMPGTD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 64 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed byte integers in (ymm2) and (ymm3/m256) for greater than.</td>
</tr>
<tr>
<td>VPCMPGTB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 65 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed word integers in (ymm2) and (ymm3/m256) for greater than.</td>
</tr>
<tr>
<td>VPCMPGTW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 66 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed doubleword integers in (ymm2) and (ymm3/m256) for greater than.</td>
</tr>
<tr>
<td>VPCMPGTD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W0 66 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int32 vector (xmm2) and int32 vector (ymm3/m128/m32bcst), and set vector mask (k1) to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>VPCMPGTB k1 ([k2]), xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W0 66 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int32 vector (ymm2) and int32 vector (ymm3/m256/m32bcst), and set vector mask (k1) to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>VPCMPGTB k1 ([k2]), ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W0 66 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare Greater between int32 elements in (zmm2) and (zmm3/m512/m32bcst), and set destination (k1) according to the comparison results under writemask. (k2).</td>
</tr>
<tr>
<td>VPCMPGTB k1 ([k2]), zmm2, zmm3/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 64 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte integers in (xmm2) and (xmm3/m128) for greater than, and set vector mask (k1) to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>VPCMPGTB k1 ([k2]), xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG 64 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte integers in (ymm2) and (ymm3/m256) for greater than, and set vector mask (k1) to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>VPCMPGTB k1 ([k2]), ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding data element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.
EVEX encoded VPCMPGTD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (with 64-bit operands)
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
ELSE DEST[7:0] ← 0; Fl;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] > SRC[63:56]
    THEN DEST[63:56] ← FFH;
ELSE DEST[63:56] ← 0; Fl;

COMPARE_BYTES_GREATER (SRC1, SRC2)
IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] ← FFH;
ELSE DEST[7:0] ← 0; Fl;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] ← FFH;
ELSE DEST[127:120] ← 0; Fl;

COMPARE_WORDS_GREATER (SRC1, SRC2)
IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] ← FFFFH;
ELSE DEST[15:0] ← 0; Fl;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] ← FFFFH;
ELSE DEST[127:112] ← 0; Fl;

COMPARE_DWORDS_GREATER (SRC1, SRC2)
IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 0; Fl;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
ELSE DEST[127:96] ← 0; Fl;

PCMPGTB (with 128-bit operands)
DEST[127:0] ← COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPGTB (VEX.128 encoded version)
DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
VPCMPGTB (VEX.256 encoded version)
DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPGTB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP ← SRC1[i+7:i] > SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
      ELSE DEST[j] ← 0; FI;
    ELSE
      DEST[j] ← 0; zeroing-masking only; FI;
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

PCMPGTW (with 64-bit operands)
IF DEST[15:0] > SRC[15:0]
  THEN DEST[15:0] ← FFFFH;
ELSE DEST[15:0] ← 0; FI;
/* Continue comparison of 2nd and 3rd words in DEST and SRC */
IF DEST[63:48] > SRC[63:48]
  THEN DEST[63:48] ← FFFFH;
ELSE DEST[63:48] ← 0; FI;

PCMPGTW (with 128-bit operands)
DEST[127:0] ← COMPARE_WORDS_GREATER(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPGTW (VEX.128 encoded version)
DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPCMPGTW (VEX.256 encoded version)
DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPGTW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP ← SRC1[i+15:i] > SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
      ELSE DEST[j] ← 0; FI;
    FI;
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

PCMPGTD (with 64-bit operands)
IF DEST[31:0] > SRC[31:0]
  THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 0; FI;
  THEN DEST[63:32] ← FFFFFFFFH;
ELSE DEST[63:32] ← 0; FI;

PCMPGTD (with 128-bit operands)
DEST[127:0] ← COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPGTD (VEX.128 encoded version)
DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0

VPCMPGTD (VEX.256 encoded version)
DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPGTD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
        ELSE CMP ← SRC1[i+31:i] > SRC2[i+31:i];
      FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalents

VPCMPGTB __mmask64 __mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 __mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 __mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 __mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 __mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 __mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTB __mmask16 __mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTB __mmask16 __mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTB __mmask8 __mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTB __mmask8 __mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTB __mmask8 __mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTB __mmask8 __mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 __mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 __mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 __mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 __mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 __mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 __mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 __mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 __mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 __mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 __mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 __mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 __mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPGTD, see Exceptions Type E4.
EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.
PCMPGTQ — Compare Packed Data for Greater Than

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 37 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE_2</td>
<td>Compare packed qwords in xmm2/m128 and xmm1 for greater than.</td>
</tr>
<tr>
<td>PCMPGTQ xmm1,xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 37 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed qwords in xmm2 and xmm3/m128 for greater than.</td>
</tr>
<tr>
<td>VPCMPGTQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 37 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed qwords in ymm2 and ymm3/m256 for greater than.</td>
</tr>
<tr>
<td>VPCMPGTQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 37 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>VPCMPGTQ k1 [k2], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 37 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>VPCMPGTQ k1 [k2], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 37 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>VPCMPGTQ k1 [k2], zmm2, zmm3/m512/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM/reg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM/reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM/reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTQ/D/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.
Operation

**COMPARE_QWORDS_GREATER (SRC1, SRC2)**

IF SRC[63:0] > SRC[63:0]
  THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ←0; FI;
IF SRC[127:64] > SRC[127:64]
  THEN DEST[127:64] ←FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] ←0; FI;

**VPCMPGTQ (VEX.128 encoded version)**

DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] ←0

**VPCMPGTQ (VEX.256 encoded version)**

DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ←COMPARE_QWORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ←0

**VPCMPGTQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k[2][j] OR "no writemask"
    THEN
      /* signed comparison */
      IF (EVEX.b = 1) AND (SRC2 "is memory")
        THEN CMP ← SRC[i+63:i] > SRC[63:0];
        ELSE CMP ← SRC[i+63:i] > SRC[i+63:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
    ELSE DEST[j] ← 0; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask(__m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm256_cmpgt_epi64_mask(__m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm_cmpgt_epi64_mask(__m128i a, __m128i b);
VPCMPGTQ __mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPGTQ: __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
PCMPGTQ: __m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPGTQ, see Exceptions Type E4.
PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 63 /r imm8</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.</td>
</tr>
<tr>
<td>VEX.128.66 0F3A.WIG 63 /r ib</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The instruction compares data from two strings based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates an index stored to ECX.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
- SFlag – Set if any byte/word of xmm1 is null, reset otherwise
- OFlag – IntRes2[0]
- AFlag – Reset
- PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Effective Operand Size

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>ECX</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>ECX</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>ECX</td>
</tr>
</tbody>
</table>

### Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

```c
int  _mm_cmpistri (__m128i a, __m128i b, const int mode);
```
Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);
int _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int _mm_cmpistro (__m128i a, __m128i b, const int mode);
int _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD If VEX.L = 1.
If VEX.vvvv ≠ 1111B.
**PCMPISTRM — Packed Compare Implicit Length Strings, Return Mask**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 62 /r imm8</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in XMM0.</td>
</tr>
<tr>
<td>PCMPISTRM xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 62 /r ib</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in XMM0.</td>
</tr>
<tr>
<td>VPCMPISTRM xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”) generating a mask stored to XMM0.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
- SFlag – Set if any byte/word of xmm1 is null, reset otherwise
- OFlag - IntRes2[0]
- AFlag – Reset
- PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Effective Operand Size

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>XMM0</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>XMM0</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>XMM0</td>
</tr>
</tbody>
</table>

### Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

```
__m128i __mm_cmpistrm (__m128i a, __m128i b, const int mode);
```
Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int     _mm_cmpistra (__m128i a, __m128i b, const int mode);
int     _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int     _mm_cmpistro (__m128i a, __m128i b, const int mode);
int     _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int     _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD If VEX.L = 1.
If VEX.vvvv ≠ 1111B.
PDEP — Parallel Bits Deposit

Operation

\[
\begin{align*}
\text{TEMP} & \leftarrow \text{SRC1}; \\
\text{MASK} & \leftarrow \text{SRC2}; \\
\text{DEST} & \leftarrow 0; \\
m & \leftarrow 0, k & \leftarrow 0; \\
\text{DO WHILE } & m < \text{OperandSize} \\
& \quad \text{IF MASK}[m] = 1 \text{ THEN} \\
& \quad \quad \text{DEST}[m] \leftarrow \text{TEMP}[k]; \\
& \quad \quad k \leftarrow k + 1; \\
& \quad \text{FI} \\
& \quad m \leftarrow m + 1; \\
& \text{OD}
\end{align*}
\]

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in
64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt
to execute this instruction with VEX.L not equal to 0 will cause #UD.
Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
PDEP: unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);
PDEP: unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally
#UD If VEX.W = 1.
PEXT — Parallel Bits Extract

### Instruction Set Reference, M-U

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.LZ.F3.0F38.W0 F5/r PEXT r32a, r32b, r/m32</td>
<td>RVM</td>
<td>V/V</td>
<td>BMI2</td>
<td>Parallel extract of bits from r32b using mask in r/m32, result is written to r32a.</td>
</tr>
<tr>
<td>VEX.NDS.LZ.F3.0F38.W1 F5/r PEXT r64a, r64b, r/m64</td>
<td>RVM</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Parallel extract of bits from r64b using mask in r/m64, result is written to r64a.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

### Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0;
m ← 0, k ← 0;
DO WHILE m< OperandSize
    IF MASK[m] = 1 THEN
        DEST[k] ← TEMP[m];
        k ← k + 1;
    FI
```

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.
m ← m+ 1;

OD

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
PEXT: unsigned __int32 _pext_u32(unsigned __int32 src, unsigned __int32 mask);
PEXT: unsigned __int64 _pext_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally
#UD If VEX.W = 1.
## PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 14 /r ib PEXTRB reg/m8, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r32 or r64 are zeroed.</td>
</tr>
<tr>
<td>66 0F 3A 16 /r ib PEXTRD r/m32, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r/m32.</td>
</tr>
<tr>
<td>66 REX.W 0F 3A 16 /r ib PEXTRQ r/m64, xmm2, imm8</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE4_1</td>
<td>Extract a qword integer value from xmm2 at the source qword offset specified by imm8 into r/m64.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 14 /r ib VPEXTRB reg/m8, xmm2, imm8</td>
<td>A</td>
<td>V^1/V</td>
<td>AVX</td>
<td>Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8</td>
<td>A</td>
<td>V/V^2</td>
<td>AVX</td>
<td>Extract a qword integer value from xmm2 at the source qword offset specified by imm8 into r64/m64.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1G 14 /r ib VPEXTRB reg/m8, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8</td>
<td>B</td>
<td>V/N.E.^2</td>
<td>AVX512DQ</td>
<td>Extract a qword integer value from xmm2 at the source qword offset specified by imm8 into r64/m64.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L"L must be
0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

**Operation**

CASE of

PEXTRB: SEL $\leftarrow$ COUNT[3:0];
    TEMP $\leftarrow$ (Src $\gg$ SEL*8) AND FFH;
    IF (DEST = Mem8)
        THEN
            Mem8 $\leftarrow$ TEMP[7:0];
        ELSE IF (64-Bit Mode and 64-bit register selected)
            THEN
                R64[7:0] $\leftarrow$ TEMP[7:0];
                r64[63:8] $\leftarrow$ ZERO_FILL;
            ELSE
                R32[7:0] $\leftarrow$ TEMP[7:0];
                r32[31:8] $\leftarrow$ ZERO_FILL;
        FI;
    PEXTRD: SEL $\leftarrow$ COUNT[1:0];
    TEMP $\leftarrow$ (Src $\gg$ SEL*32) AND FFFF_FFFFH;
    DEST $\leftarrow$ TEMP;
    PEXTRQ: SEL $\leftarrow$ COUNT[0];
    TEMP $\leftarrow$ (Src $\gg$ SEL*64); 
    DEST $\leftarrow$ TEMP;

EASC:

VPEXTRTD/VPEXTRQ

IF (64-Bit Mode and 64-bit dest operand)
    THEN
        Src_Offset $\leftarrow$ Imm8[0]
        r64/m64 $\leftarrow$ (Src $\gg$ Src_Offset * 64)
    ELSE
        Src_Offset $\leftarrow$ Imm8[1:0]
        r32/m32 $\leftarrow$ ((Src $\gg$ Src_Offset *32) AND 0FFFFFFFH);
    FI

VPEXTRB (dest=m8)

SRC_Offset $\leftarrow$ Imm8[3:0]
Mem8 $\leftarrow$ (Src $\gg$ Src_Offset*8)

VPEXTRB (dest=reg)

IF (64-Bit Mode)
    THEN
        SRC_Offset $\leftarrow$ Imm8[3:0]
        DEST[7:0] $\leftarrow$ ((Src $\gg$ Src_Offset*8) AND OFH)
        DEST[63:8] $\leftarrow$ ZERO_FILL;
    ELSE
        SRC_Offset $\leftarrow$. Imm8[3:0];
        DEST[7:0] $\leftarrow$ ((Src $\gg$ Src_Offset*8) AND OFH);
        DEST[31:8] $\leftarrow$ ZERO_FILL;
    FI
Intel C/C++ Compiler Intrinsic Equivalent
PEXTRB: int _mm_extract_epi8 (__m128i src, const int ndx);
PEXTRD: int _mm_extract_epi32 (__m128i src, const int ndx);
PEXTRQ: __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 5;
EVEX-encoded instruction, see Exceptions Type E9NF.
#UD If VEX.L = 1 or EVEX.L’L > 0.
If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
## PEXTRW—Extract Word

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C5 /r ib1 PEXTRW reg, mm, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Extract the word specified by imm8 from mm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.</td>
</tr>
<tr>
<td>66 0F C5 /r ib PEXTRW reg, xmm, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Extract the word specified by imm8 from xmm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.</td>
</tr>
<tr>
<td>66 0F 3A 15 /r ib PEXTRW reg/m16, xmm, imm8</td>
<td>B</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Extract the word specified by imm8 from xmm and copy it to lowest 16 bits of reg or m16. Zero-extend the result in the destination, r32 or r64.</td>
</tr>
<tr>
<td>VEX.128.66.0F:w0 C5 /r ib VPEXTRW reg, xmm1, imm8</td>
<td>A</td>
<td>V^2/V</td>
<td>AVX</td>
<td>Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A:w0 15 /r ib VPEXTRW reg/m16, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:w1G C5 /r ib VPEXTRW reg, xmm1, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512B_W</td>
<td>Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A:w1G 15 /r ib VPEXTRW reg/m16, xmm2, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512B_W</td>
<td>Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
</tbody>
</table>

### NOTES:
2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).
In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

**Operation**

IF (DEST = Mem16)
    THEN
        SEL ← COUNT[2:0];
        TEMP ← (Src >> SEL*16) AND FFFFH;
        Mem16 ← TEMP[15:0];
    ELSE IF (64-Bit Mode and destination is a general-purpose register)
        THEN
            FOR (PEXTRW instruction with 64-bit source operand)
                { SEL ← COUNT[1:0];
                  TEMP ← (Src >> (SEL * 16)) AND FFFFH;
                  r64[15:0] ← TEMP[15:0];
                  r64[63:16] ← ZERO_FILL;
                }
            FOR (PEXTRW instruction with 128-bit source operand)
                { SEL ← COUNT[2:0];
                  TEMP ← (Src >> (SEL * 16)) AND FFFFH;
                  r64[15:0] ← TEMP[15:0];
                  r64[63:16] ← ZERO_FILL;
                }
            ELSE
                FOR (PEXTRW instruction with 64-bit source operand)
                    { SEL ← COUNT[1:0];
                      TEMP ← (Src >> (SEL * 16)) AND FFFFH;
                      r32[15:0] ← TEMP[15:0];
                      r32[31:16] ← ZERO_FILL;
                    }
                FOR (PEXTRW instruction with 128-bit source operand)
                    { SEL ← COUNT[2:0];
                      TEMP ← (Src >> (SEL * 16)) AND FFFFH;
                      r32[15:0] ← TEMP[15:0];
                      r32[31:16] ← ZERO_FILL;
                    }
        ELSE
            FOR (PEXTRW instruction with 64-bit source operand)
                { SEL ← COUNT[1:0];
                  TEMP ← (Src >> (SEL * 16)) AND FFFFH;
                  r32[15:0] ← TEMP[15:0];
                  r32[31:16] ← ZERO_FILL;
                }
            FOR (PEXTRW instruction with 128-bit source operand)
                { SEL ← COUNT[2:0];
                  TEMP ← (Src >> (SEL * 16)) AND FFFFH;
                  r32[15:0] ← TEMP[15:0];
                  r32[31:16] ← ZERO_FILL;
                }
        FI;
    FI;

VPEXTRW (dest=m16)
SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> Src_Offset*16)
VPEXTRw ( dest=reg)  
IF (64-Bit Mode )  
THEN  
  SRC_Offset ← Imm8[2:0]  
  DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)  
  DEST[63:16] ← ZERO_FILL;  
ELSE  
  SRC_Offset ← Imm8[2:0]  
  DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)  
  DEST[31:16] ← ZERO_FILL;  
FI  

Intel C/C++ Compiler Intrinsic Equivalent  
PEXTRw: int _mm_extract_pi16 (__m64 a, int n)  
PEXTRw: int _mm_extract_epi16 ( __m128i a, int imm)  

Flags Affected  
None.  

Numeric Exceptions  
None.  

Other Exceptions  
Non-EVEX-encoded instruction, see Exceptions Type 5;  
EVEX-encoded instruction, see Exceptions Type E9NF.  
#UD  
  If VEX.L = 1 or EVEX.L’L > 0.  
  If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
PHADDW/PHADDD — Packed Horizontal Add

**Description**

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Notes:


---

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 01 r1 PHADDW mm1, mm2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 16-bit integers horizontally, pack to mm1.</td>
</tr>
<tr>
<td>66 0F 38 01 r PHADDW xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 16-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>NP 0F 38 02 r PHADDD mm1, mm2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 32-bit integers horizontally, pack to mm1.</td>
</tr>
<tr>
<td>66 0F 38 02 r PHADDD xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 32-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 01 r PHADDD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add 32-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 02 r PHADDD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add 32-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 01 r PHADDD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add 32-bit signed integers horizontally, pack to ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 02 r PHADDD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add 32-bit signed integers horizontally, pack to ymm1.</td>
</tr>
</tbody>
</table>
VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.
VPHADDW (VEX.256 encoded version)
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]

椎 4-282 Vol. 2B

Intel C/C++ Compiler Intrinsic Equivalents
PHADDW: __m64 _mm_hadd_pi16 (__m64 a, __m64 b)
PHADDD: __m64 _mm_hadd_p i32 (__m64 a, __m64 b)
(V)PHADDW: __m128i _mm_hadd_epi16 (__m128i a, __m128i b)
(V)PHADDW: __m128i _mm_hadd_epi32 (__m128i a, __m128i b)
PHADDW: __m256i _mm256_hadd_epi16 (__m256i a, __m256i b)
PHADDW: __m256i _mm256_hadd_epi32 (__m256i a, __m256i b)
SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 4; additionally
#UD If VEX.L = 1.
PHADDSW — Packed Horizontal Add and Saturate

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/En</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 03 /r</td>
<td>RM V/V</td>
<td>SSSE3</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to mm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 03 /r</td>
<td>RM V/V</td>
<td>SSSE3</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 03 /r</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 03 /r</td>
<td>RVM V/V</td>
<td>AVX2</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to ymm1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

(V)PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand) When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

**PHADDSW (with 64-bit operands)**

\[
\begin{align*}
\text{mm1}[15:0] &= \text{SaturateToSignedWord}((\text{mm1}[31:16] + \text{mm1}[15:0])); \\
\text{mm1}[31:16] &= \text{SaturateToSignedWord}((\text{mm1}[63:48] + \text{mm1}[47:32])); \\
\text{mm1}[47:32] &= \text{SaturateToSignedWord}((\text{mm2/m64}[31:16] + \text{mm2/m64}[15:0])); \\
\text{mm1}[63:48] &= \text{SaturateToSignedWord}((\text{mm2/m64}[63:48] + \text{mm2/m64}[47:32])); \\
\end{align*}
\]
**PHADDSW (with 128-bit operands)**

```
xmm1[15-0] = SaturateToSignedWord(xmm1[31-16] + xmm1[15-0]);
xmm1[31-16] = SaturateToSignedWord(xmm1[63-48] + xmm1[47-32]);
xmm1[47-32] = SaturateToSignedWord(xmm1[95-80] + xmm1[79-64]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[31-16] + xmm2/m128[15-0]);
xmm1[95-80] = SaturateToSignedWord(xmm2/m128[63-48] + xmm2/m128[47-32]);
xmm1[111-96] = SaturateToSignedWord(xmm2/m128[95-80] + xmm2/m128[79-64]);
```

**VPHADDSW (VEX.128 encoded version)**

```
DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[MAXVL-1:128] = 0
```

**VPHADDSW (VEX.256 encoded version)**

```
DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[239:224] = SaturateToSignedWord(SRC2[191-160] + SRC2[159-128])
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
PHADDSW: __m64 _mm_hadds_pi16 (__m64 a, __m64 b)
(V)PHADDSW: __m128i _mm_hadds_epi16 (__m128i a, __m128i b)
VPHADDSW: __m256i _mm256_hadds_e pi16 (__m256i a, __m256i b)
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD IF VEX.L = 1.
# PHMINPOSUW — Packed Horizontal Word Minimum

## Opcode/Description

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 41 rPHMINPOSUW xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 41 rVPHMINPOSUW xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1.</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

## Description

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding XMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination XMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

## Operation

**PHMINPOSUW (128-bit Legacy SSE version)**

- `INDEX ← 0;`
- `MIN ← SRC[15:0]`
- IF (SRC[31:16] < MIN)
  - THEN `INDEX ← 1; MIN ← SRC[31:16]; Fl;`
- IF (SRC[47:32] < MIN)
  - THEN `INDEX ← 2; MIN ← SRC[47:32]; Fl;`
- * Repeat operation for words 3 through 6
- IF (SRC[127:112] < MIN)
  - THEN `INDEX ← 7; MIN ← SRC[127:112]; Fl;`
- `DEST[15:0] ← MIN;`
- `DEST[18:16] ← INDEX;`
- `DEST[127:19] ← 000000000000000000000000000H;`
**VPHMINPOSUw (VEX.128 encoded version)**

INDEX ← 0
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX ← 1; MIN ← SRC[31:16]
IF (SRC[47:32] < MIN) THEN INDEX ← 2; MIN ← SRC[47:32]
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN) THEN INDEX ← 7; MIN ← SRC[127:112]
DEST[15:0] ← MIN
DEST[18:16] ← INDEX
DEST[127:19] ← 0000000000000000000000000000H
DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

`VPHMINPOSUw: __m128i _mm_minpos_epu16(__m128i packed_words);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.L = 1.
   If VEX.vvvv ≠ 1111B.
PHSUBW/PHSUBD — Packed Horizontal Subtract

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (r, w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

NOTES:

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

**Operation**

**PHSUBW (with 64-bit operands)**

\[
\begin{align*}
mm1[15-0] &= mm1[15-0] - mm1[31-16]; \\
mm1[31-16] &= mm1[47-32] - mm1[63-48]; \\
mm1[47-32] &= mm2/m64[15-0] - mm2/m64[31-16]; \\
mm1[63-48] &= mm2/m64[47-32] - mm2/m64[63-48];
\end{align*}
\]

**PHSUBW (with 128-bit operands)**

\[
\begin{align*}
xmm1[15-0] &= xmm1[15-0] - xmm1[31-16]; \\
xmm1[31-16] &= xmm1[47-32] - xmm1[63-48]; \\
xmm1[47-32] &= xmm1[79-64] - xmm1[95-80]; \\
xmm1[63-48] &= xmm1[111-96] - xmm1[127-112]; \\
xmm1[79-64] &= xmm2/m128[15-0] - xmm2/m128[31-16]; \\
xmm1[95-80] &= xmm2/m128[47-32] - xmm2/m128[63-48]; \\
xmm1[111-96] &= xmm2/m128[79-64] - xmm2/m128[95-80]; \\
\end{align*}
\]

**VPHSUBW (VEX.128 encoded version)**

\[
\begin{align*}
&\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] - \text{SRC1}[31:16] \\
&\text{DEST}[31:16] \leftarrow \text{SRC1}[47:32] - \text{SRC1}[63:48] \\
&\text{DEST}[47:32] \leftarrow \text{SRC1}[79:64] - \text{SRC1}[95:80] \\
&\text{DEST}[63:48] \leftarrow \text{SRC1}[111:96] - \text{SRC1}[127:112] \\
&\text{DEST}[79:64] \leftarrow \text{SRC2}[15:0] - \text{SRC2}[31:16] \\
&\text{DEST}[95:80] \leftarrow \text{SRC2}[47:32] - \text{SRC2}[63:48] \\
&\text{DEST}[111:96] \leftarrow \text{SRC2}[79:64] - \text{SRC2}[95:80] \\
&\text{DEST}[127:112] \leftarrow \text{SRC2}[111:96] - \text{SRC2}[127:112] \\
&\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\end{align*}
\]

**VPHSUBW (VEX.256 encoded version)**

\[
\begin{align*}
&\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] - \text{SRC1}[31:16] \\
&\text{DEST}[31:16] \leftarrow \text{SRC1}[47:32] - \text{SRC1}[63:48] \\
&\text{DEST}[47:32] \leftarrow \text{SRC1}[79:64] - \text{SRC1}[95:80] \\
&\text{DEST}[63:48] \leftarrow \text{SRC1}[111:96] - \text{SRC1}[127:112] \\
&\text{DEST}[79:64] \leftarrow \text{SRC2}[15:0] - \text{SRC2}[31:16] \\
&\text{DEST}[95:80] \leftarrow \text{SRC2}[47:32] - \text{SRC2}[63:48] \\
&\text{DEST}[111:96] \leftarrow \text{SRC2}[79:64] - \text{SRC2}[95:80] \\
&\text{DEST}[127:112] \leftarrow \text{SRC2}[111:96] - \text{SRC2}[127:112] \\
&\text{DEST}[143:128] \leftarrow \text{SRC1}[143:128] - \text{SRC1}[159:144] \\
&\text{DEST}[159:144] \leftarrow \text{SRC1}[175:160] - \text{SRC1}[191:176] \\
&\text{DEST}[175:160] \leftarrow \text{SRC1}[207:192] - \text{SRC1}[223:208] \\
&\text{DEST}[191:176] \leftarrow \text{SRC1}[239:224] - \text{SRC1}[255:240] \\
&\text{DEST}[207:192] \leftarrow \text{SRC2}[143:128] - \text{SRC2}[159:144] \\
&\text{DEST}[223:208] \leftarrow \text{SRC2}[175:160] - \text{SRC2}[191:176] \\
&\text{DEST}[239:224] \leftarrow \text{SRC2}[207:192] - \text{SRC2}[223:208] \\
&\text{DEST}[255:240] \leftarrow \text{SRC2}[239:224] - \text{SRC2}[255:240]
\end{align*}
\]

**PHSUBD (with 64-bit operands)**

\[
\begin{align*}
mm1[31-0] &= mm1[31-0] - mm1[63-32]; \\
mm1[63-32] &= mm2/m64[31-0] - mm2/m64[63-32];
\end{align*}
\]
PHSUBD (with 128-bit operands)

\[
\begin{align*}
\text{xmm1}[31-0] &= \text{xmm1}[31-0] - \text{xmm1}[63-32]; \\
\text{xmm1}[63-32] &= \text{xmm1}[95-64] - \text{xmm1}[127-96]; \\
\text{xmm1}[95-64] &= \text{xmm2}_2[31-0] - \text{xmm2}_2[63-32]; \\
\text{xmm1}[127-96] &= \text{xmm2}_2[95-64] - \text{xmm2}_2[127-96]; \\
\end{align*}
\]

VPHSUBD (VEX.128 encoded version)

\[
\begin{align*}
\text{DEST}[31-0] &\leftarrow \text{SRC1}[31-0] - \text{SRC1}[63-32] \\
\text{DEST}[63-32] &\leftarrow \text{SRC1}[95-64] - \text{SRC1}[127-96] \\
\text{DEST}[95-64] &\leftarrow \text{SRC2}[31-0] - \text{SRC2}[63-32] \\
\text{DEST}[127-96] &\leftarrow \text{SRC2}[95-64] - \text{SRC2}[127-96] \\
\text{DEST}[\text{MAXVL}-1:128] &\leftarrow 0 \\
\end{align*}
\]

VPHSUBD (VEX.256 encoded version)

\[
\begin{align*}
\text{DEST}[31:0] &\leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32] \\
\text{DEST}[63:32] &\leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96] \\
\text{DEST}[95:64] &\leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32] \\
\text{DEST}[127:96] &\leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96] \\
\text{DEST}[159:128] &\leftarrow \text{SRC1}[159:128] - \text{SRC1}[191:160] \\
\text{DEST}[191:160] &\leftarrow \text{SRC1}[223:192] - \text{SRC1}[255:224] \\
\end{align*}
\]

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW: \text{__m64 } _mm_hsub_pi16 (\text{__m64 a, __m64 b})

PHSUBD: \text{__m64 } _mm_hsub_pi32 (\text{__m64 a, __m64 b})

(V)PHSUBW: \text{__m128i } _mm_hsub_epi16 (\text{__m128i a, __m128i b})

(V)PHSUBD: \text{__m128i } _mm_hsub_epi32 (\text{__m128i a, __m128i b})

VPHSUBW: \text{__m256i } _mm256_hsub_epi16 (\text{__m256i a, __m256i b})

VPHSUBD: \text{__m256i } _mm256_hsub_epi32 (\text{__m256i a, __m256i b})

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.
**PHSUBSW — Packed Horizontal Subtract and Saturate**

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Description</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract 16-bit signed integer horizontally, pack saturated integers to mm1.</td>
<td>NP 0F 38 07 Ir</td>
<td>R</td>
<td>SSSE3</td>
<td>(\text{mm1}[15-0] = \text{Sat} \text{arateToSignedWord}(\text{mm1}[15-0] - \text{mm1}[31-16]);) (\text{mm1}[31-16] = \text{Sat} \text{arateToSignedWord}(\text{mm1}[47-32] - \text{mm1}[63-48]);) (\text{mm1}[47-32] = \text{Sat} \text{arateToSignedWord}(\text{mm2/m64}[15-0] - \text{mm2/m64}[31-16]);) (\text{mm1}[63-48] = \text{Sat} \text{arateToSignedWord}(\text{mm2/m64}[47-32] - \text{mm2/m64}[63-48]);)</td>
</tr>
<tr>
<td>66 0F 38 07 Ir</td>
<td>R</td>
<td>SSSE3</td>
<td>Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

PHSUBSW (with 128-bit operands)

\[
\begin{align*}
\text{xmm1}[15-0] &= \text{SaturateToSignedWord}(\text{xmm1}[15-0] - \text{xmm1}[31-16]) \\
\text{xmm1}[31-16] &= \text{SaturateToSignedWord}(\text{xmm1}[47-32] - \text{xmm1}[63-48]) \\
\text{xmm1}[47-32] &= \text{SaturateToSignedWord}(\text{xmm1}[79-64] - \text{xmm1}[95-80]) \\
\text{xmm1}[63-48] &= \text{SaturateToSignedWord}(\text{xmm1}[111-96] - \text{xmm1}[127-112]) \\
\text{xmm1}[79-64] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[15-0] - \text{xmm2/m128}[31-16]) \\
\text{xmm1}[95-80] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[47-32] - \text{xmm2/m128}[63-48]) \\
\text{xmm1}[111-96] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[79-64] - \text{xmm2/m128}[95-80]) \\
\text{xmm1}[127-112] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[111-96] - \text{xmm2/m128}[127-112])
\end{align*}
\]

VPHSUBSW (VEX.128 encoded version)

\[
\begin{align*}
\text{DEST}[15:0] &= \text{SaturateToSignedWord}(\text{SRC1}[15:0] - \text{SRC1}[31:16]) \\
\text{DEST}[31:16] &= \text{SaturateToSignedWord}(\text{SRC1}[47:32] - \text{SRC1}[63:48]) \\
\text{DEST}[47:32] &= \text{SaturateToSignedWord}(\text{SRC1}[79:64] - \text{SRC1}[95:80]) \\
\text{DEST}[63:48] &= \text{SaturateToSignedWord}(\text{SRC1}[111:96] - \text{SRC1}[127:112]) \\
\text{DEST}[79:64] &= \text{SaturateToSignedWord}(\text{SRC2}[15:0] - \text{SRC2}[31:16]) \\
\text{DEST}[95:80] &= \text{SaturateToSignedWord}(\text{SRC2}[47:32] - \text{SRC2}[63:48]) \\
\text{DEST}[111:96] &= \text{SaturateToSignedWord}(\text{SRC2}[79:64] - \text{SRC2}[95:80]) \\
\text{DEST}[127:112] &= \text{SaturateToSignedWord}(\text{SRC2}[111:96] - \text{SRC2}[127:112]) \\
\text{DEST}[MAXVL-1:128] &= 0
\end{align*}
\]

VPHSUBSW (VEX.256 encoded version)

\[
\begin{align*}
\text{DEST}[15:0] &= \text{SaturateToSignedWord}(\text{SRC1}[15:0] - \text{SRC1}[31:16]) \\
\text{DEST}[31:16] &= \text{SaturateToSignedWord}(\text{SRC1}[47:32] - \text{SRC1}[63:48]) \\
\text{DEST}[47:32] &= \text{SaturateToSignedWord}(\text{SRC1}[79:64] - \text{SRC1}[95:80]) \\
\text{DEST}[63:48] &= \text{SaturateToSignedWord}(\text{SRC1}[111:96] - \text{SRC1}[127:112]) \\
\text{DEST}[79:64] &= \text{SaturateToSignedWord}(\text{SRC2}[15:0] - \text{SRC2}[31:16]) \\
\text{DEST}[95:80] &= \text{SaturateToSignedWord}(\text{SRC2}[47:32] - \text{SRC2}[63:48]) \\
\text{DEST}[111:96] &= \text{SaturateToSignedWord}(\text{SRC2}[79:64] - \text{SRC2}[95:80]) \\
\text{DEST}[127:112] &= \text{SaturateToSignedWord}(\text{SRC2}[111:96] - \text{SRC2}[127:112]) \\
\text{DEST}[143:128] &= \text{SaturateToSignedWord}(\text{SRC1}[143:128] - \text{SRC1}[159:144]) \\
\text{DEST}[159:144] &= \text{SaturateToSignedWord}(\text{SRC1}[175:160] - \text{SRC1}[191:176]) \\
\text{DEST}[175:160] &= \text{SaturateToSignedWord}(\text{SRC1}[207:192] - \text{SRC1}[223:208]) \\
\text{DEST}[191:176] &= \text{SaturateToSignedWord}(\text{SRC1}[239:224] - \text{SRC1}[255:240]) \\
\text{DEST}[207:192] &= \text{SaturateToSignedWord}(\text{SRC2}[143:128] - \text{SRC2}[159:144]) \\
\text{DEST}[223:208] &= \text{SaturateToSignedWord}(\text{SRC2}[175:160] - \text{SRC2}[191:176]) \\
\text{DEST}[239:224] &= \text{SaturateToSignedWord}(\text{SRC2}[207:192] - \text{SRC2}[223:208]) \\
\text{DEST}[255:240] &= \text{SaturateToSignedWord}(\text{SRC2}[239:224] - \text{SRC2}[255:240])
\end{align*}
\]

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSw: \[
\text{__m64 } \text{__mm_hsubs_pi16 (__m64 a, __m64 b)}
\]

(V)PHSUBSw: \[
\text{__m128i } \text{__mm_hsubs_epi16 (__m128i a, __m128i b)}
\]

VPHSUBSw: \[
\text{__m256i } \text{__mm256_hsubs_epi16 (__m256i a, __m256i b)}
\]

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.
PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>66 0F 3A 20 /r ib PINSRB xmm1, r32/m8, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Insert a byte integer value from r32/m8 into xmm1 at the destination element in xmm1 specified by imm8.</td>
</tr>
<tr>
<td>A</td>
<td>66 0F 3A 22 /r ib PINSRD xmm1, r/m32, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Insert a dword integer value from r/m32 into the xmm1 at the destination element specified by imm8.</td>
</tr>
<tr>
<td>A</td>
<td>66 REX.W 0F 3A 22 /r ib PINSRQ xmm1, r/m64, imm8</td>
<td>A</td>
<td>V/N. E.</td>
<td>SSE4_1</td>
<td>Insert a qword integer value from r/m64 into the xmm1 at the destination element specified by imm8.</td>
</tr>
<tr>
<td>B</td>
<td>VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8</td>
<td>B</td>
<td>V’/V</td>
<td>AVX</td>
<td>Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.</td>
</tr>
<tr>
<td>B</td>
<td>VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r/m32, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.</td>
</tr>
<tr>
<td>B</td>
<td>VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r/m64, imm8</td>
<td>B</td>
<td>V/I^2</td>
<td>AVX</td>
<td>Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.</td>
</tr>
<tr>
<td>C</td>
<td>EVEX.NDS.128.66.0F3A.W1G 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.</td>
</tr>
<tr>
<td>C</td>
<td>EVEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRD xmm1, xmm2, r/m32, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.</td>
</tr>
<tr>
<td>C</td>
<td>EVEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8</td>
<td>C</td>
<td>V/N.E.^2</td>
<td>AVX512DQ</td>
<td>Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.</td>
</tr>
</tbody>
</table>

### Notes:
1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description
Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.
In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

**Operation**

**CASE OF**

**PINSRB:**
\[
\text{SEL} \leftarrow \text{COUNT}[3:0];
\text{MASK} \leftarrow (0\text{FFH} << (\text{SEL} \times 8));
\text{TEMP} \leftarrow (((\text{SRC}[7:0] << (\text{SEL} \times 8)) \text{ AND MASK});
\]

**PINSRD:**
\[
\text{SEL} \leftarrow \text{COUNT}[1:0];
\text{MASK} \leftarrow (0\text{FFFFFFFFH} << (\text{SEL} \times 32));
\text{TEMP} \leftarrow (((\text{SRC} << (\text{SEL} \times 32)) \text{ AND MASK});
\]

**PINSRQ:**
\[
\text{SEL} \leftarrow \text{COUNT}[0];
\text{MASK} \leftarrow (0\text{FFFFFFFFFFFFFFFFH} << (\text{SEL} \times 64));
\text{TEMP} \leftarrow (((\text{SRC} << (\text{SEL} \times 64)) \text{ AND MASK});
\]

**ESAC;**

\[
\text{DEST} \leftarrow ((\text{DEST AND NOT MASK}) \text{ OR TEMP});
\]

**VPINSRB (VEX/EVEX encoded version)**

\[
\text{SEL} \leftarrow \text{imm8}[3:0]
\]

\[
\text{DEST}[127:0] \leftarrow \text{write_b_element}(\text{SEL}, \text{SRC2}, \text{SRC1})
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\]

**VPINSRD (VEX/EVEX encoded version)**

\[
\text{SEL} \leftarrow \text{imm8}[1:0]
\]

\[
\text{DEST}[127:0] \leftarrow \text{write_d_element}(\text{SEL}, \text{SRC2}, \text{SRC1})
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\]

**VPINSRQ (VEX/EVEX encoded version)**

\[
\text{SEL} \leftarrow \text{imm8}[0]
\]

\[
\text{DEST}[127:0] \leftarrow \text{write_q_element}(\text{SEL}, \text{SRC2}, \text{SRC1})
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

**PINSRB:**
\[
\text{__m128i } _\text{mm_insert_epi8 (\text{__m128i } s1, int } s2, \text{ const int ndx});
\]

**PINSRD:**
\[
\text{__m128i } _\text{mm_insert_epi32 (\text{__m128i } s2, int } s, \text{ const int ndx});
\]

**PINSRQ:**
\[
\text{__m128i } _\text{mm_insert_epi64(\text{__m128i } s2, } \text{__int64 } s, \text{ const int ndx});
\]

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type 5;
EVEX-encoded instruction, see Exceptions Type E9NF.

#UD If VEX.L = 1 or EVEX.L’L > 0.
INSTRUCTION SET REFERENCE, M-U

PINSRW—Insert Word

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C4 /r ib¹ PINSRW mm, r32/m16, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Insert the low word from r32 or from m16 into mm at the word position specified by imm8.</td>
</tr>
<tr>
<td>66 0F C4 /r ib PINSRW xmm, r32/m16, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move the low word of r32 or from m16 into xmm at the word position specified by imm8.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8</td>
<td>B</td>
<td>V²/V</td>
<td>AVX</td>
<td>Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.</td>
</tr>
</tbody>
</table>

NOTES:
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L’L must be 0, otherwise the instruction will #UD.
Operation

PINSRW (with 64-bit source operand)

SEL ← COUNT AND 3H;

CASE (Determine word position) OF
  SEL ← 0: MASK ← 000000000000FFFFH;
  SEL ← 1: MASK ← 00000000FFFF0000H;
  SEL ← 2: MASK ← 0000FFFF00000000H;
  SEL ← 3: MASK ← FFFF000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);

PINSRW (with 128-bit source operand)

SEL ← COUNT AND 7H;

CASE (Determine word position) OF
  SEL ← 0: MASK ← 0000000000000000000000000000FFFFH;
  SEL ← 1: MASK ← 000000000000000000000000FFFF0000H;
  SEL ← 2: MASK ← 00000000000000000000FFFF00000000H;
  SEL ← 3: MASK ← 0000000000000000FFFF000000000000H;
  SEL ← 4: MASK ← 00000000FFFF00000000000000000000H;
  SEL ← 5: MASK ← 00000000FFFF00000000000000000000000H;
  SEL ← 6: MASK ← 0000FFFF00000000000000000000000000H;
  SEL ← 7: MASK ← FFFF000000000000000000000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);

VPINSRW (VEX/EVEX encoded version)

SEL ← imm8[2:0]

DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)

DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW: __m64 _mm_insert_pi16 (__m64 a, int d, int n)

PINSRW: __m128i _mm_insert_epi16 ( __m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;
EVEX-encoded instruction, see Exceptions Type E9NF.

#UD If VEX.L = 1 or EVEX.L'L > 0.
PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

**Op/En Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 04 /r³</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to mm1.</td>
</tr>
<tr>
<td>66 0F 38 04 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 04 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 04 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.WIG 04 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.WIG 04 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.WIG 04 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w) ModRM:r/m (r) NA NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w) VEX.vvvv (r) ModRM:r/m (r) NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w) VEX.vvvv (r) ModRM:r/m (r) NA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.
128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

**PMADDUBSW (with 64 bit operands)**

\[
\text{DEST}[15-0] = \text{SaturateToSignedWord}(\text{SRC}[15-8] \times \text{DEST}[15-8] + \text{SRC}[7-0] \times \text{DEST}[7-0]); \\
\text{DEST}[31-16] = \text{SaturateToSignedWord}(\text{SRC}[31-24] \times \text{DEST}[31-24] + \text{SRC}[23-16] \times \text{DEST}[23-16]); \\
\text{DEST}[47-32] = \text{SaturateToSignedWord}(\text{SRC}[47-40] \times \text{DEST}[47-40] + \text{SRC}[39-32] \times \text{DEST}[39-32]); \\
\text{DEST}[63-48] = \text{SaturateToSignedWord}(\text{SRC}[63-56] \times \text{DEST}[63-56] + \text{SRC}[55-48] \times \text{DEST}[55-48]);
\]

**PMADDUBSW (with 128 bit operands)**

\[
\text{DEST}[15-0] = \text{SaturateToSignedWord}(\text{SRC}[15-8] \times \text{DEST}[15-8] + \text{SRC}[7-0] \times \text{DEST}[7-0]); \\
// Repeat operation for 2nd through 7th word \\
\text{SRC1/DEST}[127-112] = \text{SaturateToSignedWord}(\text{SRC}[127-120] \times \text{DEST}[127-120] + \text{SRC}[119-112] \times \text{DEST}[119-112]);
\]

**VPMADDUBSW (VEX.128 encoded version)**

\[
\text{DEST}[15:0] \leftarrow \text{SaturateToSignedWord}(\text{SRC2}[15:8] \times \text{SRC1}[15:8] + \text{SRC2}[7:0] \times \text{SRC1}[7:0]) \\
// Repeat operation for 2nd through 7th word \\
\text{DEST}[127:112] \leftarrow \text{SaturateToSignedWord}(\text{SRC2}[127:120] \times \text{SRC1}[127:120] + \text{SRC2}[119:112] \times \text{SRC1}[119:112]) \\
\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\]

**VPMADDUBSW (VEX.256 encoded version)**

\[
\text{DEST}[15:0] \leftarrow \text{SaturateToSignedWord}(\text{SRC2}[15:8] \times \text{SRC1}[15:8] + \text{SRC2}[7:0] \times \text{SRC1}[7:0]) \\
// Repeat operation for 2nd through 15th word \\
\text{DEST}[255:240] \leftarrow \text{SaturateToSignedWord}(\text{SRC2}[255:248] \times \text{SRC1}[255:248] + \text{SRC2}[247:240] \times \text{SRC1}[247:240]) \\
\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0
\]

**VPMADDUBSW (EVEX encoded versions)**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
\quad i \leftarrow j \times 16 \\
\quad \text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\quad \text{THEN } \text{DEST}[i+15:i] \leftarrow \text{SaturateToSignedWord}(\text{SRC2}[i+15:i+8] \times \text{SRC1}[i+15:i+8] + \text{SRC2}[i+7:i] \times \text{SRC1}[i+7:i]) \\
\quad \text{ELSE} \\
\quad \quad \text{IF } \text{*merging-masking*} \\
\quad \quad \text{THEN } \text{DEST}[i+15:i] \text{ remains unchanged*} \\
\quad \quad \text{ELSE } \text{*zeroing-masking*} \\
\quad \quad \text{DEST}[i+15:i] = 0 \\
\quad \text{FI} \\
\text{FI; \ ENDFOR; \\
\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0
\]
**Intel C/C++ Compiler Intrinsic Equivalents**

- VPMADDUBSW __m512i __mm512_maddubs_epi16(__m512i a, __m512i b);
- VPMADDUBSW __m512i __mm512_mask_maddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
- VPMADDUBSW __m512i __mm512_maskz_maddubs_epi16(__mmask32 k, __m512i a, __m512i b);
- VPMADDUBSW __m256i __mm256_mask_maddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
- VPMADDUBSW __m256i __mm256_maskz_maddubs_epi16(__mmask16 k, __m256i a, __m256i b);
- VPMADDUBSW __m128i __mm_mask_maddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
- VPMADDUBSW __m128i __mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b);
- PMADDUBSW: __m64 __mm_maddubs_pi16 (__m64 a, __m64 b)
- (V)PMADDUBSW: __m128i __mm_maddubs_epi16 (__m128i a, __m128i b)
- VPMADDUBSW: __m256i __mm256_maddubs_epi16 (__m256i a, __m256i b)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
PMADDWD—Multiply and Add Packed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F F5 /r</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Multiply the packed words in ( mm ) by the packed words in ( mm/m64 ), add adjacent doubleword results, and store in ( mm ).</td>
</tr>
<tr>
<td>PMADDWD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F F5 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed word integers in ( xmm1 ) by the packed word integers in ( xmm2/m128 ), add adjacent doubleword results, and store in ( xmm1 ).</td>
</tr>
<tr>
<td>PMADDWD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG F5 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed word integers in ( xmm2 ) by the packed word integers in ( xmm3/m128 ), add adjacent doubleword results, and store in ( xmm1 ).</td>
</tr>
<tr>
<td>VPMADDWD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG F5 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed word integers in ( ymm2 ) by the packed word integers in ( ymm3/m256 ), add adjacent doubleword results, and store in ( ymm1 ).</td>
</tr>
<tr>
<td>VPMADDWD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG F5 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed word integers in ( xmm2 ) by the packed word integers in ( xmm3/m128 ), add adjacent doubleword results, and store in ( xmm1 ) under writemask k1.</td>
</tr>
<tr>
<td>VPMADDWD xmm1[k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG F5 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed word integers in ( ymm2 ) by the packed word integers in ( ymm3/m256 ), add adjacent doubleword results, and store in ( ymm1 ) under writemask k1.</td>
</tr>
<tr>
<td>VPMADDWD ymm1[k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG F5 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply the packed word integers in ( zmm2 ) by the packed word integers in ( zmm3/m512 ), add adjacent doubleword results, and store in ( zmm1 ) under writemask k1.</td>
</tr>
<tr>
<td>VPMADDWD zmm1[k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMtr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMtr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMtr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).
The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

---

**Operation**

**PMADDWD (with 64-bit operands)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{DEST}[15:0] \times \text{SRC}[15:0]) + (\text{DEST}[31:16] \times \text{SRC}[31:16]) \\
\text{DEST}[63:32] & \leftarrow (\text{DEST}[47:32] \times \text{SRC}[47:32]) + (\text{DEST}[63:48] \times \text{SRC}[63:48])
\end{align*}
\]

**PMADDWD (with 128-bit operands)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{DEST}[15:0] \times \text{SRC}[15:0]) + (\text{DEST}[31:16] \times \text{SRC}[31:16]) \\
\text{DEST}[63:32] & \leftarrow (\text{DEST}[47:32] \times \text{SRC}[47:32]) + (\text{DEST}[63:48] \times \text{SRC}[63:48]) \\
\text{DEST}[95:64] & \leftarrow (\text{DEST}[79:64] \times \text{SRC}[79:64]) + (\text{DEST}[95:80] \times \text{SRC}[95:80]) \\
\text{DEST}[127:96] & \leftarrow (\text{DEST}[111:96] \times \text{SRC}[111:96]) + (\text{DEST}[127:112] \times \text{SRC}[127:112])
\end{align*}
\]

**VPMADDWD (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{SRC}[15:0] \times \text{SRC2}[15:0]) + (\text{SRC}[31:16] \times \text{SRC2}[31:16]) \\
\text{DEST}[63:32] & \leftarrow (\text{SRC}[47:32] \times \text{SRC2}[47:32]) + (\text{SRC}[63:48] \times \text{SRC2}[63:48]) \\
\text{DEST}[95:64] & \leftarrow (\text{SRC}[79:64] \times \text{SRC2}[79:64]) + (\text{SRC}[95:80] \times \text{SRC2}[95:80]) \\
\text{DEST}[127:96] & \leftarrow (\text{SRC}[111:96] \times \text{SRC2}[111:96]) + (\text{SRC}[127:112] \times \text{SRC2}[127:112]) \\
\text{DEST}[\text{MAXVL}-1:128] & = 0
\end{align*}
\]
VPMADDWD (VEX.256 encoded version)
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[MAXVL-1:256] ← 0

VPMADDWD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← (SRC2[i+31:i+16] * SRC1[i+31:i+16]) + (SRC2[i+15:i] * SRC1[i+15:i])
        ELSE IF *merging-masking* ; merging-masking
            THEN DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+31:i] = 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VPMADDWD __m512i _mm512_madd_epi16( __m512i a, __m512i b);
VPMADDWD __m512i __m512i_mask_madd_epi16( __m512i a, __m512i s, __mmask32 k, __m512i m512i a, __m512i b);
VPMADDWD __m512i __m512i_maskz_madd_epi16( __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m256i __m256i_mask_madd_epi16( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m256i __m256i_maskz_madd_epi16( __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m128i __m128i_mask_madd_epi16( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i __m128i_maskz_madd_epi16( __mmask8 k, __m128i a, __m128i b);
PMADDWD: __m64 __m64_madd_pi16( __m64 m1, __m64 m2);
(V)PMADDWD: __m128i __m128i_mask_madd_epi16 ( __m128i a, __m128i b)
VPMADDWD: __m256i __m256i_mask_madd_epi16 ( __m256i a, __m256i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EE /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare signed word integers in mm2/m64 and mm1 and return maximum values.</td>
</tr>
<tr>
<td>PMAXSw mm1, mm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>PMAXSB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.</td>
</tr>
<tr>
<td>PMAXSD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 3C /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VPMAXSB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VPMAXSD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 3C /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>VPMAXSB ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 3D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 3C /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXSB xmm1[k1][z], xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VEX.NDS.512.66.0F38.WIG 3C /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXSB zmm1[k1][z], zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 3D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXSW xmm1[k1][z], ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VEX.NDS.512.66.0F38.WIG 3D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXSW zmm1[k1][z], zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 3D /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.
Operation

PMAXSW (64-bit operands)
IF DEST[15:0] > SRC[15:0] THEN
  DEST[15:0] ← DEST[15:0];
ELSE
  DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
  DEST[63:48] ← DEST[63:48];
ELSE
  DEST[63:48] ← SRC[63:48]; FI;

PMAXSB (128-bit Legacy SSE version)
IF DEST[7:0] > SRC[7:0] THEN
  DEST[7:0] ← DEST[7:0];
ELSE
  DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
  DEST[127:120] ← DEST[127:120];
ELSE
  DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)

VPMAXSB (VEX.128 encoded version)
IF SRC1[7:0] > SRC2[7:0] THEN
  DEST[7:0] ← SRC1[7:0];
ELSE
  DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
  DEST[127:120] ← SRC1[127:120];
ELSE
  DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0

VPMAXSB (VEX.256 encoded version)
IF SRC1[7:0] > SRC2[7:0] THEN
  DEST[7:0] ← SRC1[7:0];
ELSE
  DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
  DEST[255:248] ← SRC1[255:248];
ELSE
  DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0
VPMAXSB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking* THEN *DEST[i+7:i] remains unchanged* ELSE ; zeroing-masking
                DEST[i+7:i] ← 0
            FI;
        FI;
    ENDFOR;
    DEST[MAXVL-1:VL] ← 0

VPMAXSW (128-bit Legacy SSE version)

IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0];
ENDIF;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112];
ENDIF;
DEST[MAXVL-1:128] ← 0

VPMAXSW (VEX.128 encoded version)

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0];
ENDIF;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112];
ENDIF;
DEST[MAXVL-1:128] ← 0

VPMAXSW (VEX.256 encoded version)

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0];
ENDIF;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240];
ENDIF;
DEST[MAXVL-1:256] ← 0
VPMAXSW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask* THEN
    IF SRC1[i+15:j] > SRC2[i+15:j]
      THEN DEST[i+15:j] ← SRC1[i+15:j];
      ELSE DEST[i+15:j] ← SRC2[i+15:j];
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+15:j] ← 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

P MAXSD (128-bit Legacy SSE version)

IF DEST[31:0] > SRC[31:0] THEN
  DEST[31:0] ← DEST[31:0];
ELSE
  DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
  DEST[127:96] ← DEST[127:96];
ELSE
  DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)

VPMAXSD (VEX.128 encoded version)

IF SRC1[31:0] > SRC2[31:0] THEN
  DEST[31:0] ← SRC1[31:0];
ELSE
  DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
  DEST[255:224] ← SRC1[255:224];
ELSE
  DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0

VPMAXSD (VEX.256 encoded version)

IF SRC1[31:0] > SRC2[31:0] THEN
  DEST[31:0] ← SRC1[31:0];
ELSE
  DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
  DEST[255:224] ← SRC1[255:224];
ELSE
  DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
VPMAXSD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j $\leftarrow$ 0 TO KL-1
  i $\leftarrow$ j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[i+31:i] > SRC2[31:0]
          THEN DEST[i+31:i] $\leftarrow$ SRC1[i+31:i];
          ELSE DEST[i+31:i] $\leftarrow$ SRC2[31:0];
        FI;
      ELSE
        IF SRC1[i+31:i] > SRC2[i+31:i]
          THEN DEST[i+31:i] $\leftarrow$ SRC1[i+31:i];
          ELSE DEST[i+31:i] $\leftarrow$ SRC2[i+31:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] $\leftarrow$ 0 ; zeroing-masking
        FI
    FI;
  ENDFOR
DEST[MAXVL-1:VL] $\leftarrow$ 0

VPMAXSQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j $\leftarrow$ 0 TO KL-1
  i $\leftarrow$ j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[i+63:i] > SRC2[63:0]
          THEN DEST[i+63:i] $\leftarrow$ SRC1[i+63:i];
          ELSE DEST[i+63:i] $\leftarrow$ SRC2[63:0];
        FI;
      ELSE
        IF SRC1[i+63:i] > SRC2[i+63:i]
          THEN DEST[i+63:i] $\leftarrow$ SRC1[i+63:i];
          ELSE DEST[i+63:i] $\leftarrow$ SRC2[i+63:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        THEN DEST[i+63:i] $\leftarrow$ 0
      FI
    FI;
  ENDFOR;
DEST[MAXVL-1:VL] $\leftarrow$ 0
Intel C/C++ Compiler Intrinsic Equivalent

VPMAXSB __m512i _mm512_max_epi8( __m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8( __m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi8( __mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16( __m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16( __mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8( __m256i s, __mmask64 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8( __mmask64 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16( __mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8( __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16( __m128i s, __mmask4 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16( __mmask4 k, __m128i a, __m128i b);
VPMAXSB __m256i _mm256_mask_max_epi32( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi32( __mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64( __mmask8 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi32( __m128i s, __mmask4 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi32( __mmask4 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64( __m128i s, __mmask2 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epi64( __mmask2 k, __m128i a, __m128i b);
VPMAXSB __m512i _mm512_mask_max_epi32( __m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi32( __mmask64 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64( __m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64( __mmask32 k, __m512i a, __m512i b);
(V)PMAXSB __m128i _mm_max_epi8( __m128i a, __m128i b);
(V)PMAXSW __m128i _mm_max_epi16( __m128i a, __m128i b);
(V)PMAXSD __m128i _mm_max_epi32( __m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8( __m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16( __m256i a, __m256i b);
PMAWSW __m64 _mm_max_epi16( __m64 a, __m64 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPMAXSD/Q, see Exceptions Type E4.
EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb.
## PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32/128 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F DE /r [\text{PMAXUB} \text{mm1, mm2/m64}]</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.</td>
</tr>
<tr>
<td>66 0F DE /r [\text{PMAXUB} \text{xmm1, xmm2/m128}]</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>66 0F 38 3E/r [\text{PMAXUW} \text{xmm1, xmm2/m128}]</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F DE /r [\text{VPMAXUB} \text{xmm1, xmm2/m3/m128}]</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38 3E/r [\text{VPMAXUW} \text{xmm1, xmm2/m3/m128}]</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F DE /r [\text{VPMAXUB} \text{ymm1, ymm2, ymm3/m256}]</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38 3E/r [\text{VPMAXUW} \text{ymm1, ymm2, ymm3/m256}]</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG DE /r [\text{VPMAXUB} \text{xmm1[k1]{z}, xmm2, xmm3/m128}]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG DE /r [\text{VPMAXUB} \text{ymm1[k1]{z}, ymm2, ymm3/m256}]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG DE /r [\text{VPMAXUB} \text{zmm1[k1]{z}, zmm2, zmm3/m512}]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.WIG 3E/r [\text{VPMAXUW} \text{xmm1[k1]{z}, xmm2, xmm3/m128}]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.WIG 3E/r [\text{VPMAXUW} \text{ymm1[k1]{z}, ymm2, ymm3/m256}]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.WIG 3E/r [\text{VPMAXUW} \text{zmm1[k1]{z}, zmm2, zmm3/m512}]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

### Notes:

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation**

**PMAXUB (64-bit operands)**

```plaintext
IF DEST[7:0] > SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0];
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] > SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56];
```

**PMAXUB (128-bit Legacy SSE version)**

```plaintext
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0];
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120];
DEST[MAXVL-1:128] (Unmodified)
```

**VPMAXUB (VEX.128 encoded version)**

```plaintext
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0];
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120];
DEST[MAXVL-1:128] ← 0
```
VPMAXUB (VEX.256 encoded version)
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI; 
DEST[MAXVL-1:128] ← 0
VPMAXUB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i] THEN
            DEST[i+7:i] ← SRC1[i+7:i];
        ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking* THEN *DEST[i+7:i] remains unchanged*
            ELSE *zeroing-masking
                DEST[i+7:i] ← 0
            ENDIF
    FI
ENDDFOR;
DEST[MAXVL-1:VL] ← 0
PMAXUW (128-bit Legacy SSE version)
IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
VPMAXUW (VEX.128 encoded version)
IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0
VPMAXUW (VEX.256 encoded version)

IF SRC1[15:0] > SRC2[15:0] THEN
   DEST[15:0] ← SRC1[15:0];
ELSE
   DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
   DEST[255:240] ← SRC1[255:240];
ELSE
   DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:128] ← 0

VPMAXUW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask* THEN
      IF SRC1[i+15:i] > SRC2[i+15:i]
         THEN DEST[i+15:i] ← SRC1[i+15:i];
         ELSE DEST[i+15:i] ← SRC2[i+15:i];
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+15:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+15:i] ← 0
      FI
   FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUB __m512i __mm512_max_epu8( __m512i a, __m512i b);
VPMAXUB __m512i __mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i __mm512_maskz_max_epu8( __mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i __mm512_max_epu16( __m512i a, __m512i b);
VPMAXUW __m512i __mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i __mm512_maskz_max_epu16( __mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i __mm256_max_epu8( __m256i a, __m256i b);
VPMAXUB __m256i __mm256_mask_max_epu8(__m256i s, __mmask64 k, __m256i a, __m256i b);
VPMAXUB __m256i __mm256_maskz_max_epu8( __mmask64 k, __m256i a, __m256i b);
VPMAXUW __m256i __mm256_max_epu16( __m256i a, __m256i b);
VPMAXUW __m256i __mm256_mask_max_epu16(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i __mm256_maskz_max_epu16( __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m128i __mm128_max_epu8 ( __m128i a, __m128i b);
VPMAXUB __m128i __mm128_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i __mm128_maskz_max_epu8( __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m64i __mm64_max_epu8( __m64i a, __m64i b);
**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
**PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 3F /r PMAXUD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.Wig 3F /r VPMAXUD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 3F /r VPMAXUD xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 3F /r VPMAXUD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 3F /r VPMAXUD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 3F /r VPMAXUQ xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 3F /r VPMAXUQ ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 3F /r VPMAXUQ zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.
Operation

**PMAXUD (128-bit Legacy SSE version)**

IF DEST[31:0] > SRC[31:0] THEN
   DEST[31:0] ← DEST[31:0];
ELSE
   DEST[31:0] ← SRC[31:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

   DEST[127:96] ← DEST[127:96];
ELSE
   DEST[127:96] ← SRC[127:96]; FI;

**VPMAXUD (VEX.128 encoded version)**

IF SRC1[31:0] > SRC2[31:0] THEN
   DEST[31:0] ← SRC1[31:0];
ELSE
   DEST[31:0] ← SRC2[31:0]; FI;

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

   DEST[127:96] ← SRC1[127:96];
ELSE
   DEST[127:96] ← SRC2[127:96]; FI;

**VPMAXUD (VEX.256 encoded version)**

IF SRC1[31:0] > SRC2[31:0] THEN
   DEST[31:0] ← SRC1[31:0];
ELSE
   DEST[31:0] ← SRC2[31:0]; FI;

(* Repeat operation for 2nd through 7th dwords in source and destination operands *)

   DEST[255:224] ← SRC1[255:224];
ELSE
   DEST[255:224] ← SRC2[255:224]; FI;

DEST[MAXVL-1:256] ← 0
VPMAXUD (EVEX encoded versions)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j \times 32\]

IF \(k1[\text{]}](\text{)} \text{ OR *no writemask*} \) THEN

IF \((\text{EVEX.b} = 1) \text{ AND} \text{ (SRC2 *is memory*)}\)

THEN

IF \(\text{SRC1}[i+31:1] > \text{SRC2}[31:0]\)

THEN \(\text{DEST}[i+31:1] \leftarrow \text{SRC1}[i+31:1]\);

ELSE \(\text{DEST}[i+31:1] \leftarrow \text{SRC2}[31:0]\);

FI;

ELSE

IF \(\text{SRC1}[i+31:1] > \text{SRC2}[i+31:1]\)

THEN \(\text{DEST}[i+31:1] \leftarrow \text{SRC1}[i+31:1]\);

ELSE \(\text{DEST}[i+31:1] \leftarrow \text{SRC2}[i+31:1]\);

FI;

FI;

ELSE

IF \(*\text{merging-masking*}; \text{merging-masking}\)

THEN \(*\text{DEST}[i+31:1] \text{ remains unchanged*}\)

ELSE \(*\text{zeroing-masking}\)

THEN \(\text{DEST}[i+31:1] \leftarrow 0\)

FI

FI;

ENDFOR;

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

VPMAXUQ (EVEX encoded versions)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j \times 64\]

IF \(k1[\text{]}](\text{)} \text{ OR *no writemask*} \) THEN

IF \((\text{EVEX.b} = 1) \text{ AND} \text{ (SRC2 *is memory*)}\)

THEN

IF \(\text{SRC1}[i+63:1] > \text{SRC2}[63:0]\)

THEN \(\text{DEST}[i+63:1] \leftarrow \text{SRC1}[i+63:1]\);

ELSE \(\text{DEST}[i+63:1] \leftarrow \text{SRC2}[63:0]\);

FI;

ELSE

IF \(\text{SRC1}[i+31:1] > \text{SRC2}[i+31:1]\)

THEN \(\text{DEST}[i+63:1] \leftarrow \text{SRC1}[i+31:1]\);

ELSE \(\text{DEST}[i+63:1] \leftarrow \text{SRC2}[i+31:1]\);

FI;

FI;

ELSE

IF \(*\text{merging-masking*}; \text{merging-masking}\)

THEN \(*\text{DEST}[i+63:1] \text{ remains unchanged*}\)

ELSE \(*\text{zeroing-masking}\)

THEN \(\text{DEST}[i+63:1] \leftarrow 0\)

FI

FI;

ENDFOR;

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUD __m512i _mm512_max_epu32(__m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_max_epu64(__m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
VPMAXUD __m256i _mm256_mask_max_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32(__mmask16 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b);
VPMAXUD __m128i _mm_mask_max_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_mask_max_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm512_max_epu32(__m128i a, __m128i b);
VPMAXUD __m256i _mm256_max_epu32(__m256i a, __m256i b);
VPMAXUQ __m128i _mm512_max_epu64(__m128i a, __m128i b);
VPMAXUQ __m256i _mm256_max_epu64(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
**PMINSB/PMINSW—Minimum of Packed Signed Integers**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EA /r³</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare signed word integers in mm2/m64 and mm1 and return minimum values.</td>
</tr>
<tr>
<td>PMINSW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 38 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4.1</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>PMINSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F EA /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>PMINSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38 38 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VPMINSW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F EA /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed word integers in xmm3/m128 and xmm1 and return packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VPMINSW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38 38 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.</td>
</tr>
<tr>
<td>VPMINSW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F EA /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.</td>
</tr>
<tr>
<td>VPMINSW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.WIG 38 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINSW xmm1{k1}[k]{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.WIG 38 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed signed byte integers in ymm1 and ymm2/m256 and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINSW ymm1{k1}[k]{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.WIG 38 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINSW zmm1{k1}[k]{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG EA /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed signed word integers in xmm1 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINSW xmm1{k1}[k]{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG EA /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed signed word integers in ymm1 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINSW ymm1{k1}[k]{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG EA /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Compare packed signed word integers in zmm1 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINSW zmm1{k1}[k]{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation
PMINSW (64-bit operands)
IF DEST[15:0] < SRC[15:0] THEN
   DEST[15:0] ← DEST[15:0];
ELSE
   DEST[15:0] ← SRC[15:0]; Fl;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
   DEST[63:48] ← DEST[63:48];
ELSE
   DEST[63:48] ← SRC[63:48]; Fl;

PMINSB (128-bit Legacy SSE version)
IF DEST[7:0] < SRC[7:0] THEN
   DEST[7:0] ← DEST[7:0];
ELSE
   DEST[15:0] ← SRC[7:0]; Fl;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
   DEST[127:120] ← DEST[127:120];
ELSE
   DEST[127:120] ← SRC[127:120]; Fl;
DEST[MAXVL-1:128] (Unmodified)

VPMINSB (VEX.128 encoded version)
IF SRC1[7:0] < SRC2[7:0] THEN
   DEST[7:0] ← SRC1[7:0];
ELSE
   DEST[7:0] ← SRC2[7:0]; Fl;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
   DEST[127:120] ← SRC1[127:120];
ELSE
   DEST[127:120] ← SRC2[127:120]; Fl;
DEST[MAXVL-1:128] ← 0
VPMINSB (VEX.256 encoded version)
IF SRC1[7:0] < SRC2[7:0] THEN
  DEST[7:0] ← SRC1[7:0];
ELSE
  DEST[15:0] ← SRC2[7:0]; Fl;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
  DEST[255:248] ← SRC1[255:248];
ELSE
  DEST[255:248] ← SRC2[255:248]; Fl;
DEST[MAXVL-1:256] ← 0

VPMINSB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask* THEN
    IF SRC1[i+7:i] < SRC2[i+7:i]
      THEN DEST[i+7:i] ← SRC1[i+7:i];
      ELSE DEST[i+7:i] ← SRC2[i+7:i];
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+7:i] ← 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0

PMINSW (128-bit Legacy SSE version)
IF DEST[15:0] < SRC[15:0] THEN
  DEST[15:0] ← DEST[15:0];
ELSE
  DEST[15:0] ← SRC[15:0]; Fl;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
  DEST[127:112] ← DEST[127:112];
ELSE
  DEST[127:112] ← SRC[127:112]; Fl;
DEST[MAXVL-1:128] (Unmodified)

VPMINSW (VEX.128 encoded version)
IF SRC1[15:0] < SRC2[15:0] THEN
  DEST[15:0] ← SRC1[15:0];
ELSE
  DEST[15:0] ← SRC2[15:0]; Fl;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
  DEST[127:112] ← SRC1[127:112];
ELSE
  DEST[127:112] ← SRC2[127:112]; Fl;
DEST[MAXVL-1:128] ← 0
VPMINSW (VEX.256 encoded version)

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0

VPMINSW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
    IF SRC1[i+15:i] < SRC2[i+15:i]
        THEN DEST[i+15:i] ← SRC1[i+15:i];
        ELSE DEST[i+15:i] ← SRC2[i+15:i];
    FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSB __m512i _mm512_min_epi8( __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16( __m512i a, __m512i b)

PMINSW: __m64 _mm_min_pi16 ( __m64 a, __m64 b)
SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
#MF (64-bit operations only) If there is a pending x87 FPU exception.
**PMINSD/PMINSQ—Minimum of Packed Signed Integers**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 39 /r PMINSD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.
EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation**

**PMINSD (128-bit Legacy SSE version)**

IF \( \text{DEST}[31:0] < \text{SRC}[31:0] \) THEN
\( \text{DEST}[31:0] \leftarrow \text{DEST}[31:0] \);
ELSE
\( \text{DEST}[31:0] \leftarrow \text{SRC}[31:0] \); FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

IF \( \text{DEST}[127:96] < \text{SRC}[127:96] \) THEN
\( \text{DEST}[127:96] \leftarrow \text{DEST}[127:96] \);
ELSE
\( \text{DEST}[127:96] \leftarrow \text{SRC}[127:96] \); FI;

\( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)

**VPMINSD (VEX.128 encoded version)**

IF \( \text{SRC1}[31:0] < \text{SRC2}[31:0] \) THEN
\( \text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \);
ELSE
\( \text{DEST}[31:0] \leftarrow \text{SRC2}[31:0] \); FI;

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

IF \( \text{SRC1}[127:96] < \text{SRC2}[127:96] \) THEN
\( \text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \);
ELSE
\( \text{DEST}[127:96] \leftarrow \text{SRC2}[127:96] \); FI;

\( \text{DEST}[\text{MAXVL}-1:128] \) \( \leftarrow 0 \)

**VPMINSD (VEX.256 encoded version)**

IF \( \text{SRC1}[31:0] < \text{SRC2}[31:0] \) THEN
\( \text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \);
ELSE
\( \text{DEST}[31:0] \leftarrow \text{SRC2}[31:0] \); FI;

(* Repeat operation for 2nd through 7th dwords in source and destination operands *)

IF \( \text{SRC1}[255:224] < \text{SRC2}[255:224] \) THEN
\( \text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \);
ELSE
\( \text{DEST}[255:224] \leftarrow \text{SRC2}[255:224] \); FI;

\( \text{DEST}[\text{MAXVL}-1:256] \) \( \leftarrow 0 \)
VPMINSQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[i+63:i] < SRC2[i+63:i]
          THEN DEST[i+63:i] ← SRC1[i+63:i];
          ELSE DEST[i+63:i] ← SRC2[i+63:i];
        FI;
      ELSE
        IF SRC1[i+63:i] < SRC2[i+63:i]
          THEN DEST[i+63:i] ← SRC1[i+63:i];
          ELSE DEST[i+63:i] ← SRC2[i+63:i];
        FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

PMINS/PMINSQ—Minimum of Packed Signed Integers
Intel C/C++ Compiler Intrinsic Equivalent

VPMINSQ __m512i _mm512_min_epi64( __m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64( __mmask16 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
### PMINUB/PMINUW—Minimum of Packed Unsigned Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F DA /r^1</td>
<td>A V/V</td>
<td>SSE</td>
<td>Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values.</td>
<td></td>
</tr>
<tr>
<td>PMINUB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F DA /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>PMINUB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 3A/r</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>PMINUW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F DA /r</td>
<td>B V/V</td>
<td>AVX</td>
<td>Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38 3A/r</td>
<td>B V/V</td>
<td>AVX</td>
<td>Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F DA /r</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38 3A/r</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F DA /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUB xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F DA /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUB ymm1 (k1)[z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F DA /r</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUB zmm1 (k1)[z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38 3A/r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUW xmm1[k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38 3A/r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUW ymm1[k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38 3A/r</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPMINUW zmm1[k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be a YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation
PMINUB (for 64-bit operands)
IF DEST[7:0] < SRC[17:0] THEN
   DEST[7:0] ← DEST[7:0];
ELSE
   DEST[7:0] ← SRC[7:0]; Fl;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] < SRC[63:56] THEN
   DEST[63:56] ← DEST[63:56];
ELSE
   DEST[63:56] ← SRC[63:56]; Fl;

PMINUB instruction for 128-bit operands:
IF DEST[7:0] < SRC[7:0] THEN
   DEST[7:0] ← DEST[7:0];
ELSE
   DEST[15:0] ← SRC[7:0]; Fl;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
   DEST[127:120] ← DEST[127:120];
ELSE
   DEST[127:120] ← SRC[127:120]; Fl;
DEST[MAXVL-1:128] (Unmodified)

VPMINUB (VEX.128 encoded version)
IF SRC1[7:0] < SRC2[7:0] THEN
   DEST[7:0] ← SRC1[7:0];
ELSE
   DEST[7:0] ← SRC2[7:0]; Fl;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
   DEST[127:120] ← SRC1[127:120];
ELSE
   DEST[127:120] ← SRC2[127:120]; Fl;
DEST[MAXVL-1:128] ← 0
VPMINUB (VEX.256 encoded version)

IF SRC[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; Fl;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    DEST[255:248] ← SRC[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; Fl;
DEST[MAXVL-1:256] ← 0

VPMINUB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k[1][j] OR *no writemask* THEN
        IF SRC[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC[i+7:i];
        ELSE DEST[i+7:i] ← SRC2[i+7:i]; FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

PMINUW instruction for 128-bit operands:

IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; Fl;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; Fl;
DEST[MAXVL-1:128] (Unmodified)

VPMINUW (VEX.128 encoded version)

IF SRC[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; Fl;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
    DEST[127:112] ← SRC[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; Fl;
DEST[MAXVL-1:128] ← 0
VPMINUW (VEX.256 encoded version)

- IF SRC1[15:0] < SRC2[15:0] THEN
  - DEST[15:0] ← SRC1[15:0];
- ELSE
  - DEST[15:0] ← SRC2[15:0]; FL;

(* Repeat operation for 2nd through 15th words in source and destination operands *)

  - DEST[255:240] ← SRC1[255:240];
- ELSE
  - DEST[255:240] ← SRC2[255:240]; FL;

DEST[MAXVL-1:256] ← 0

VPMINUW (EVEX encoded versions)

- (KL, VL) = (8, 128), (16, 256), (32, 512)
- FOR j ← 0 TO KL-1
  - i ← j + 16
  - IF k1[j] OR *no writemask* THEN
    - IF SRC1[i+15:i] < SRC2[i+15:i]
      - THEN DEST[i+15:i] ← SRC1[i+15:i];
    - ELSE DEST[i+15:i] ← SRC2[i+15:i];
    - FI;
  - ELSE
    - IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    - ELSE ; zeroing-masking
      DEST[i+15:i] ← 0
    - FI
  - FI
ENDFOR;

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUB __m512i _mm512_min_epu8( __m512i a, __m512i b);
VPMINUB __m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i _mm512_maskz_min_epu8( __mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_min_epu16( __m512i a, __m512i b);
VPMINUW __m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_maskz_min_epu16( __mmask32 k, __m512i a, __m512i b);
VPMINUW __m256i _mm256_min_epu8(__m256i s, __m256i a, __m256i b);
VPMINUW __m256i _mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_maskz_min_epu8( __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i _mm256_min_epu16( __m256i a, __m256i b);
VPMINUW __m256i _mm256_min_epu16( __m256i a, __m256i b);
VPMINUW __m128i _mm128_min_epu8( __m128i s, __m128i a, __m128i b);
VPMINUW __m128i _mm128_min_epu16( __m128i s, __m128i a, __m128i b);
VPMINUW __m128i _mm128_min_epu16( __m128i s, __m128i a, __m128i b);
VPMINUW __m128i _mm128_min_epu16( __m128i s, __m128i a, __m128i b);
VPMINUW __m64i _mm64_min_epu8( __m64i a, __m64i b);
VPMINUW __m64i _mm64_min_epu16( __m64i a, __m64i b);
VPMINUW __m64i _mm64_min_epu16( __m64i a, __m64i b);
VPMINUW __m64i _mm64_min_epu16( __m64i a, __m64i b);
SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
**PMINUD/PMINUQ—Minimum of Packed Unsigned Integers**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 3B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 3B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 3B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.WO 3B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.WO 3B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.WO 3B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.WL 3B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.WL 3B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.WL 3B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM/reg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM/reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM/reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.
Operation

**PMINUD (128-bit Legacy SSE version)**

PMINUD instruction for 128-bit operands:

- IF DEST[31:0] < SRC[31:0] THEN
  - DEST[31:0] ← DEST[31:0];
- ELSE
  - DEST[31:0] ← SRC[31:0];

(* Repeat operation for 2nd through 7th words in source and destination operands *)

  - DEST[127:96] ← DEST[127:96];
- ELSE
  - DEST[127:96] ← SRC[127:96];

DEST[MAXVL-1:128] (Unmodified)

**VPMINUD (VEX.128 encoded version)**

VPMINUD instruction for 128-bit operands:

- IF SRC1[31:0] < SRC2[31:0] THEN
  - DEST[31:0] ← SRC1[31:0];
- ELSE
  - DEST[31:0] ← SRC2[31:0];

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

  - DEST[127:96] ← SRC1[127:96];
- ELSE
  - DEST[127:96] ← SRC2[127:96];

DEST[MAXVL-1:128] ← 0

**VPMINUD (VEX.256 encoded version)**

VPMINUD instruction for 128-bit operands:

- IF SRC1[31:0] < SRC2[31:0] THEN
  - DEST[31:0] ← SRC1[31:0];
- ELSE
  - DEST[31:0] ← SRC2[31:0];

(* Repeat operation for 2nd through 7th dwords in source and destination operands *)

  - DEST[255:224] ← SRC1[255:224];
- ELSE
  - DEST[255:224] ← SRC2[255:224];

DEST[MAXVL-1:256] ← 0
VPMINUD (EVEX encoded versions)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)
\(i \leftarrow j \times 32\)
IF \(k1[j]\) OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
IF \(SRC1[i+31:i] < SRC2[31:0]\)
THEN \(DEST[i+31:i] \leftarrow SRC1[i+31:i]\);
ELSE \(DEST[i+31:i] \leftarrow SRC2[31:0]\);
FI;
ELSE
IF \(SRC1[i+31:i] < SRC2[i+31:i]\)
THEN \(DEST[i+31:i] \leftarrow SRC1[i+31:i]\);
ELSE \(DEST[i+31:i] \leftarrow SRC2[i+31:i]\);
FI;
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+31:i] \(\leftarrow 0\)
FI
FI;
ENDFOR;

\(DEST[MAXVL-1:VL] \leftarrow 0\)

VPMINUQ (EVEX encoded versions)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)
\(i \leftarrow j \times 64\)
IF \(k1[j]\) OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
IF \(SRC1[i+63:i] < SRC2[63:0]\)
THEN \(DEST[i+63:i] \leftarrow SRC1[i+63:i]\);
ELSE \(DEST[i+63:i] \leftarrow SRC2[63:0]\);
FI;
ELSE
IF \(SRC1[i+63:i] < SRC2[i+63:i]\)
THEN \(DEST[i+63:i] \leftarrow SRC1[i+63:i]\);
ELSE \(DEST[i+63:i] \leftarrow SRC2[i+63:i]\);
FI;
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] \(\leftarrow 0\)
FI
FI;
ENDFOR;

\(DEST[MAXVL-1:VL] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VPMINUD __m512i _mm512_min_epu32(__m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu32(__mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64(__mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_min_epu32(__m256i a, __m256i b);
VPMINUD __m256i _mm256_mask_min_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu32(__mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64(__mmask8 k, __m256i a, __m256i b);
VPMINUQ __m128i _mm_mask_min_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu32(__mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
PMOVMSKB—Move Byte Mask

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D7 /r1 PMOVMSKB reg, mm</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Move a byte mask of mm to reg. The upper bits of r32 or r64 are zeroed</td>
</tr>
<tr>
<td>66 0F D7 /r PMOVMSKB reg, xmm</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move a byte mask of xmm to reg. The upper bits of r32 or r64 are zeroed</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG D7 /r VPMOVMSKB reg, xmm1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D7 /r VPMOVMSKB reg, ymm1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Move a 32-bit mask of ymm1 to reg. The upper bits of r64 are filled with zeros.</td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

Operation

PMOVMSKB (with 64-bit source operand and r32)

\[
\begin{align*}
    r32[0] &\leftarrow \text{SRC}[7]; \\
    r32[1] &\leftarrow \text{SRC}[15]; \\
    (* \text{Repeat operation for bytes 2 through 6} *) \\
    r32[7] &\leftarrow \text{SRC}[63]; \\
    r32[31:8] &\leftarrow \text{ZERO_FILL}; \\
\end{align*}
\]

(V)PMOVMSKB (with 128-bit source operand and r32)

\[
\begin{align*}
    r32[0] &\leftarrow \text{SRC}[7]; \\
    r32[1] &\leftarrow \text{SRC}[15]; \\
    (* \text{Repeat operation for bytes 2 through 14} *) \\
    r32[15] &\leftarrow \text{SRC}[127]; \\
    r32[31:16] &\leftarrow \text{ZERO_FILL}; \\
\end{align*}
\]
VPMOVMSKB (with 256-bit source operand and r32)
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 3rd through 31 *)
r32[31] ← SRC[255];

PMOVMSKB (with 64-bit source operand and r64)
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r64[7] ← SRC[63];
r64[63:8] ← ZERO_FILL;

(V)PMOVMSKB (with 128-bit source operand and r64)
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r64[15] ← SRC[127];
r64[63:16] ← ZERO_FILL;

VPMOVMSKB (with 256-bit source operand and r64)
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 31 *)
r64[31] ← SRC[255];
r64[63:32] ← ZERO_FILL;

Intel C/C++ Compiler Intrinsic Equivalent
PMOVMSKB: int _mm_movemask_pi8(__m64 a)
(V)PMOVMSKB: int _mm_movemask_epi8 (__m128i a)
VPMOVMSKB: int _mm256_movemask_epi8 (__m256i a)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
See Exceptions Type 7; additionally
#UD If VEX.vvvv ≠ 1111B.
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 20 /r PMOVSXBW xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 21 /r PMOVSXBD xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 22 /r PMOVSXBQ xmm1, xmm2/m16</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 23 /r PMOVSXWD xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 24 /r PMOVSXWQ xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 25 /r PMOVSXDQ xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m32</td>
<td>A V/V</td>
<td>AVX</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16</td>
<td>A V/V</td>
<td>AVX</td>
<td>Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m32</td>
<td>A V/V</td>
<td>AVX</td>
<td>Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m16</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m16 to 4 packed 64-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m32</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64</td>
<td>B V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256</td>
<td>B V/V</td>
<td>AVX512Bw</td>
<td>Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>Opcode/Instruction</td>
<td>Op / En</td>
<td>64/32 bit Mode Support</td>
<td>CPUID Feature Flag</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------</td>
<td>------------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 21 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 21 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 22 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 22 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 22 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 23 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 23 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 23 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 24 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 24 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 24 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 25 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 25 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 25 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

**Packed_Sign_Extend_BYTE_to_WORD(DEST, SRC)**

DEST[15:0] ← SignExtend(SRC[7:0]);
DEST[31:16] ← SignExtend(SRC[15:8]);
DEST[47:32] ← SignExtend(SRC[23:16]);
DEST[63:48] ← SignExtend(SRC[31:24]);
DEST[79:64] ← SignExtend(SRC[39:32]);
DEST[95:80] ← SignExtend(SRC[47:40]);
DEST[111:96] ← SignExtend(SRC[55:48]);
DEST[127:112] ← SignExtend(SRC[63:56]);

**Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)**

DEST[31:0] ← SignExtend(SRC[7:0]);
DEST[63:32] ← SignExtend(SRC[15:8]);
DEST[95:64] ← SignExtend(SRC[23:16]);
DEST[127:96] ← SignExtend(SRC[31:24]);

**Packed_Sign_Extend_BYTE_to_QWORD(DEST, SRC)**

DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);

**Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)**

DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);

**Packed_Sign_Extend_WORD_to_QWORD(DEST, SRC)**

DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);
Packed_Sign_Extend_DWORD_to_QWORD(DEST, SRC)
DEST[63:0] <- SignExtend(SRC[31:0]);
DEST[127:64] <- SignExtend(SRC[63:32]);

VPMOVXSXBw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
  Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
  Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
  Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j <- 0 TO KL-1
  i <- j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] <- TEMP_DEST[i+15:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:j] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] <- 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] <- 0

VPMOVXSXBd (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
  Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
  Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
  Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j <- 0 TO KL-1
  i <- j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] <- TEMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:j] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] <- 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] <- 0
VPMOVSBQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
   Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
   Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
   Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j  0 TO KL-1
   i  j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i]  TEMP_DEST[i+63:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+63:i]  0
   FI
FI;
ENDFOR
DEST[MAXVL-1:VL]  0

VPMOVSWD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
   Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
   Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
   Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j  0 TO KL-1
   i  j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i]  TEMP_DEST[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+31:i]  0
   FI
FI;
ENDFOR
DEST[MAXVL-1:VL]  0
VPMOVVSXWQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
   Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
   Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
   Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j  0 TO KL-1
   i  j * 64
   IF kl[j] OR *no writemask*
      THEN DEST[i+63:i]  TEMP_DEST[i+63:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+63:i]  0
   FI
   FI;
ENDFOR
DEST[MAXVL-1:VL]  0

VPMOVVSXDQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
   Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
   Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
   Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j  0 TO KL-1
   i  j * 64
   IF kl[j] OR *no writemask*
      THEN DEST[i+63:i]  TEMP_DEST[i+63:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+63:i]  0
   FI
   FI;
ENDFOR
DEST[MAXVL-1:VL]  0

VPMOVXSXBW (VEX.256 encoded version)
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:128]  0
VPMOVSXBD (VEX.256 encoded version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVSXBQ (VEX.256 encoded version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] ← 0

VPMOVSXWD (VEX.256 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVSXWQ (VEX.256 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVSXDQ (VEX.256 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVSXBW (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPMOVSXBQ (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPMOVSXWD (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPMOVSXWQ (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPMOVSXDQ (VEX.128 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
PMOVSB
Packed Move Word (dest[127:0], src[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSB
Packed Move Byte (dest[127:0], src[127:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMOVSB
_mm512 extracti8_i16(__m512 a);

PMOVSB
_mm512 casti8_i32(__m512 a, __mmask8 k);

PMOVSB
_mm512 casti8_i64(__m512 a, __mmask8 k);

PMOVSB
_mm512 casti8_i128k(__m512 a, __mmask64 k);

PMOVSB
_mm512 casti8_i256k(__m512 a, __mmask128 k);

PMOVSB
_mm512 casti8_i512k(__m512 a, __mmask256 k);

PMOVSB
_mm512 casti8_i1024k(__m512 a, __mmask512 k);

PMOVSB
_mm512 casti8_i2048k(__m512 a, __mmask1024 k);
PMOVSXWQ __m256i _mm256_cvtepi16_epi64(__m256i a);
PMOVSXWQ __m256i _mm256_mask_cvtepi16_epi64(__m256i a, __mmask8 k, __m256i b);
PMOVSXWQ __m256i _mm256_maskz_cvtepi16_epi64(__mmask8 k, __m256i a);
PMOVSXWB __m128i _mm_mask_cvtepi8_epi16(__m128i a, __mmask8 k, __m128i b);
PMOVSXWB __m128i _mm_maskz_cvtepi8_epi16(__mmask8 k, __m128i b);
PMOVSXBD __m128i _mm_mask_cvtepi8_epi32(__m128i a, __mmask8 k, __m128i b);
PMOVSXBD __m128i _mm_maskz_cvtepi8_epi32(__mmask8 k, __m128i b);
PMOVSXBQ __m128i _mm_mask_cvtepi8_epi64(__m128i a, __mmask8 k, __m128i b);
PMOVSXBQ __m128i _mm_maskz_cvtepi8_epi64(__mmask8 k, __m128i b);
PMOVSXBW __m128i _mm_ cvtepi8_epi16 ( __m128i a);
PMOVSXBD __m128i _mm_ cvtepi8_epi32 ( __m128i a);
PMOVSXBQ __m128i _mm_ cvtepi8_epi64 ( __m128i a);
PMOVSXWD __m128i _mm_ cvtepi16_epi32 ( __m128i a);
PMOVSXDQ __m128i _mm_ cvtepi32_epi64 ( __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 5.
EVEX-encoded instruction, see Exceptions Type E5.
#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.
### PMOVZX—Packed Move with Zero Extend

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0f 38 30 /r PMOVZX8bw xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0f 38 31 /r PMOVZX8BD xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0f 38 32 /r PMOVZX8BQ xmm1, xmm2/m16</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0f 38 33 /r PMOVZX8wD xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0f 38 34 /r PMOVZX8wQ xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 30 /r VPMOVZX8bw xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 31 /r VPMOVZX8BD xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 32 /r VPMOVZX8BQ xmm1, xmm2/m16</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 33 /r VPMOVZX8wD xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 34 /r VPMOVZX8wQ xmm1, xmm2/m16</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 35 /r VPMOVZX8DQ xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 30 /r VPMOVZX8BW ymm1, ymm2/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 31 /r VPMOVZX8BD ymm1, ymm2/m64</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 32 /r VPMOVZX8BQ ymm1, ymm2/m32</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 33 /r VPMOVZX8wD ymm1, ymm2/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 34 /r VPMOVZX8wQ ymm1, ymm2/m64</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 35 /r VPMOVZX8DQ ymm1, ymm2/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>Opcode/ Instruction</td>
<td>Op / En</td>
<td>64/32 bit Mode Support</td>
<td>CPUID Feature Flag</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
<td>------------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVEX.128.66.0F38 30.WIG /r VPMOVZXBW xmm1 {k1}{z}, xmm2/m64</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 {k1}{z}, xmm2/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW zmm1 {k1}{z}, ymm2/m256</td>
<td>B V/V</td>
<td>AVX512BW</td>
<td>Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 31 /r VPMOVZXBQ xmm1 {k1}{z}, xmm2/m32</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 31 /r VPMOVZXBQ ymm1 {k1}{z}, xmm2/m64</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 31 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/m128</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 32 /r VPMOVZXWD xmm1 {k1}{z}, xmm2/m16</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 2 packed 16-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 32 /r VPMOVZXWD ymm1 {k1}{z}, xmm2/m32</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 32 /r VPMOVZXWD zmm1 {k1}{z}, xmm2/m64</td>
<td>D V/V</td>
<td>AVX512F</td>
<td>Zero extend 8 packed 16-bit integers in xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 33 /r VPMOVZXwD xmm1 {k1}{z}, xmm2/m64</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 33 /r VPMOVZXwD ymm1 {k1}{z}, ymm2/m256</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 8 packed 16-bit integers in ymm2/m256 to 8 packed 32-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 33 /r VPMOVZXwD zmm1 {k1}{z}, ymm2/m512</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Zero extend 16 packed 16-bit integers in ymm2/m512 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 34 /r VPMOVZXwQ xmm1 {k1}{z}, xmm2/m32</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 34 /r VPMOVZXwQ ymm1 {k1}{z}, xmm2/m64</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 34 /r VPMOVZXwQ zmm1 {k1}{z}, xmm2/m128</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
PMOVZX—Packed Move with Zero Extend

INSTRUCTION SET REFERENCE, M-U

Vol. 2B  4-351

PMOVZX—Packed Move with Zero Extend

Op / Instruction

<table>
<thead>
<tr>
<th>Opcode/</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 35 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 35 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 35 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Zero extend 8 packed 32-bit integers in xmm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Half Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Quarter Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Eighth Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Legacy, VEX and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

**Packed_Zero_Extend_BYTE_to_WORD***(DEST, SRC)***

DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);

**Packed_Zero_Extend_BYTE_to_DWORD***(DEST, SRC)***

DEST[31:0] ← ZeroExtend(SRC[7:0]);
DEST[63:32] ← ZeroExtend(SRC[15:8]);
DEST[95:64] ← ZeroExtend(SRC[23:16]);
DEST[127:96] ← ZeroExtend(SRC[31:24]);
Packed_Zero_Extend_BYTE_to_QWORD(DEST, SRC)
DEST[63:0] ← ZeroExtend(SRC[7:0]);
DEST[127:64] ← ZeroExtend(SRC[15:8]);

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)
DEST[31:0] ← ZeroExtend(SRC[15:0]);
DEST[63:32] ← ZeroExtend(SRC[31:16]);
DEST[95:64] ← ZeroExtend(SRC[47:32]);
DEST[127:96] ← ZeroExtend(SRC[63:48]);

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)
DEST[63:0] ← ZeroExtend(SRC[15:0]);
DEST[127:64] ← ZeroExtend(SRC[31:16]);

Packed_Zero_Extend_DWORD_to_QWORD(DEST, SRC)
DEST[31:0] ← ZeroExtend(SRC[31:0]);
DEST[63:64] ← ZeroExtend(SRC[63:32]);

VPMOVZXBD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 32
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[191:128])
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask*
THEN DEST[i+15:i] ← TEMP_DEST[i+15:i]
ELSE
IF *merging-masking* ; merging-masking
THEN DEST[i+15:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
DEST[i+15:i] ← 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
IF k1[i] OR *no writemask*
  THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
       DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPMOVZXBQB (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPMOVZXWD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

PMOVZX—Packed Move with Zero Extend
DEST[i+31:i] ← 0
FI
FI;
ENDIF;
DEST[MAXVL-1:VL] ← 0

VPMOVZXWQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
  Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
  Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI
  FI;
ENDIF;
DEST[MAXVL-1:VL] ← 0

VPMOVZXWDQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
  Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
  Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI
  FI;
ENDIF;
DEST[MAXVL-1:VL] ← 0
VPMOVZXBw (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_toWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_toWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXBd (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_toDWord(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_toDWord(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVZXBq (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_toQWord(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_toQWord(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] ← 0

VPMOVZXwD (VEX.256 encoded version)
Packed_Zero_Extend_WORD_toDWord(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_toDWord(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXwQ (VEX.256 encoded version)
Packed_Zero_Extend_WORD_toQWord(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_toQWord(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVZXdQ (VEX.256 encoded version)
Packed_Zero_Extend_DWord_toQWord(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWord_toQWord(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXBw (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_toWORD()
DEST[MAXVL-1:128] ← 0

VPMOVZXBD (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_toDWord()
DEST[MAXVL-1:128] ← 0

VPMOVZXBQ (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_toQWord()
DEST[MAXVL-1:128] ← 0

VPMOVZXWD (VEX.128 encoded version)
Packed_Zero_Extend_WORD_toDWord()
DEST[MAXVL-1:128] ← 0

VPMOVZXWQ (VEX.128 encoded version)
Packed_Zero_Extend_WORD_toQWord()
DEST[MAXVL-1:128] ← 0

VPMOVZXDQ (VEX.128 encoded version)
Packed_Zero_Extend_DWord_toQWord()
DEST[MAXVL-1:128] ← 0
PMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZxbd
Packed_Zero_Extend_BYTE_to(DWORD)
DEST[MAXVL-1:128] (Unmodified)

PMOVZxbq
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZxwd
Packed_Zero_Extend_WORD_to(DWORD)
DEST[MAXVL-1:128] (Unmodified)

PMOVZxwq
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZxdq
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VPMOVZXBw __m512i_mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBw __m512i_mm512_mask_cvtepu8_epi16(__m512i a, __mmask32 k, __m256i b);
VPMOVZXBw __m512i_mm512_maskz_cvtepu8_epi16(__mmask32 k, __m256i b);
VPMOVZXBd __m512i_mm512_cvtepu8_epi32(__m128i a);
VPMOVZXBd __m512i_mm512_mask_cvtepu8_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXBd __m512i_mm512_maskz_cvtepu8_epi32(__mmask16 k, __m128i b);
VPMOVZXbq __m512i_mm512_cvtepu8_epi64(__m128i a);
VPMOVZXbq __m512i_mm512_mask_cvtepu8_epi64(__m512i a, __mmask8 k, __m128i b);
VPMOVZXbq __m512i_mm512_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);
VPMOVZXdq __m512i_mm512_cvtepu32_epi64(__m128i a);
VPMOVZXdq __m512i_mm512_mask_cvtepu32_epi64(__m512i a, __mmask8 k, __m128i b);
VPMOVZXdq __m512i_mm512_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);
VPMOVZXwd __m512i_mm512_cvtepu16_epi32(__m256i a);
VPMOVZXwd __m512i_mm512_mask_cvtepu16_epi32(__m512i a, __mmask8 k, __m128i b);
VPMOVZXwd __m512i_mm512_maskz_cvtepu16_epi32(__mmask8 k, __m128i b);
VPMOVZXwq __m512i_mm512_cvtepu16_epi64(__m256i a);
VPMOVZXwq __m512i_mm512_mask_cvtepu16_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXwq __m512i_mm512_maskz_cvtepu16_epi64(__mmask8 k, __m256i b);
VPMOVZXDQ __m256i __mm256_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m256i __mm256_maskz_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m256i __mm256_maskz_cvtepu16_epi32(__m256i a, __mmask16 k, __m128i b);
VPMOVZXWD __m256i __mm256_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m256i __mm256_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);
VPMOVZXWQ __m256i __mm256_maskz_cvtepu16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBW __m128i __mm_maskz_cvtepu8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBW __m128i __mm_maskz_cvtepu8_epi16(__mmask8 k, __m128i b);
VPMOVZXBW __m128i __mm_maskz_cvtepu8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBW __m128i __mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 5.
EVEX-encoded instruction, see Exceptions Type E5.

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.
PMULDQ—Multiply Packed Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 28 /r PMULDQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 28 /r VPMULDQ xmm1 (k1)[2], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 28 /r VPMULDQ ymm1 (k1)[2], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 28 /r VPMULDQ zmm1 (k1)[2], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand are the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.
EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM register. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

VPMULDQ (EVEX encoded versions)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
\quad i \leftarrow j \times 64 \\
\quad \text{IF } k1[j] \text{ OR *no writemask*} \\
\text{THEN} \\
\quad \text{IF } (EVEX.b = 1) \text{ AND SRC2 *is memory*} \\
\quad \quad \text{THEN } DEST[i+63:j] \leftarrow \text{SignExtend64}(SRC1[i+31:j]) \times \text{SignExtend64}(SRC2[31:0]) \\
\quad \quad \text{ELSE } DEST[i+63:j] \leftarrow \text{SignExtend64}(SRC1[i+31:j]) \times \text{SignExtend64}(SRC2[i+31:j]) \\
\quad \text{ELSE} \\
\quad \quad \text{IF *merging-masking*} \\
\quad \quad \quad \text{DEST[i+63:j] remains unchanged*} \\
\quad \quad \quad \text{ELSE *zeroing-masking*} \\
\quad \quad \quad \quad \text{DEST}[i+63:j] \leftarrow 0 \\
\quad \text{FI} \\
\text{FI}; \\
\text{ENDFOR} \\
\text{DEST[MAXVL-1:VL] \leftarrow 0}
\]

VPMULDQ (VEX.256 encoded version)

\[
\text{DEST}[63:0] \leftarrow \text{SignExtend64}(SRC1[31:0]) \times \text{SignExtend64}(SRC2[31:0]) \\
\text{DEST}[127:64] \leftarrow \text{SignExtend64}(SRC1[95:64]) \times \text{SignExtend64}(SRC2[95:64]) \\
\text{DEST}[191:128] \leftarrow \text{SignExtend64}(SRC1[159:128]) \times \text{SignExtend64}(SRC2[159:128]) \\
\text{DEST}[255:192] \leftarrow \text{SignExtend64}(SRC1[223:192]) \times \text{SignExtend64}(SRC2[223:192]) \\
\text{DEST[MAXVL-1:256] \leftarrow 0}
\]

VPMULDQ (VEX.128 encoded version)

\[
\text{DEST}[63:0] \leftarrow \text{SignExtend64}(SRC1[31:0]) \times \text{SignExtend64}(SRC2[31:0]) \\
\text{DEST}[127:64] \leftarrow \text{SignExtend64}(SRC1[95:64]) \times \text{SignExtend64}(SRC2[95:64]) \\
\text{DEST[MAXVL-1:128] \leftarrow 0}
\]

PMULDQ (128-bit Legacy SSE version)

\[
\text{DEST}[63:0] \leftarrow \text{SignExtend64}(DEST[31:0]) \times \text{SignExtend64}(SRC[31:0]) \\
\text{DEST}[127:64] \leftarrow \text{SignExtend64}(DEST[95:64]) \times \text{SignExtend64}(SRC[95:64]) \\
\text{DEST[MAXVL-1:128] (Unmodified)}
\]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\text{VPMULDQ } \_mm512\_mm512\_mul\_epi32(\_m512\_a, \_m512\_b); \\
\text{VPMULDQ } \_mm512\_mm512\_mask\_mul\_epi32(\_m512\_s, \_mmask8\_k, \_m512\_a, \_m512\_b); \\
\text{VPMULDQ } \_mm512\_mm512\_mask2\_mul\_epi32(\_mmask8\_k, \_m512\_a, \_m512\_b); \\
\text{VPMULDQ } \_mm256\_mm256\_mask\_mul\_epi32(\_mmask8\_k, \_m256\_a, \_m256\_b); \\
\text{VPMULDQ } \_mm256\_mm256\_mask2\_mul\_epi32(\_mmask8\_k, \_m256\_a, \_m256\_b); \\
\text{VPMULDQ } \_mm128\_mm128\_mask\_mul\_epi32(\_mmask8\_k, \_m128\_a, \_m128\_b); \\
\text{VPMULDQ } \_mm128\_mm128\_mask2\_mul\_epi32(\_mmask8\_k, \_m128\_a, \_m128\_b); \\
\text{(V)PMULDQ } \_mm128\_mm128\_mul\_epi32(\_m128\_a, \_m128\_b); \\
\text{VPMULDQ } \_mm256\_mm256\_mul\_epi32(\_m256\_a, \_m256\_b);
\]
SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
PMULHRSW — Packed Multiply High with Round and Scale

Description
PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.
128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

**PMULHRSW (with 64-bit operands)**

\[
\begin{align*}
tmp0[31:0] &= \text{INT32 } ((\text{DEST}[15:0] \times \text{SRC}[15:0]) \gg 14) + 1; \\
tmp1[31:0] &= \text{INT32 } ((\text{DEST}[31:16] \times \text{SRC}[31:16]) \gg 14) + 1; \\
tmp2[31:0] &= \text{INT32 } ((\text{DEST}[47:32] \times \text{SRC}[47:32]) \gg 14) + 1; \\
tmp3[31:0] &= \text{INT32 } ((\text{DEST}[63:48] \times \text{SRC}[63:48]) \gg 14) + 1; \\
\text{DEST}[15:0] &= \text{temp0}[16:1]; \\
\text{DEST}[31:16] &= \text{temp1}[16:1]; \\
\text{DEST}[47:32] &= \text{temp2}[16:1]; \\
\text{DEST}[63:48] &= \text{temp3}[16:1]; \\
\end{align*}
\]

PMULHRSW (with 128-bit operand)

\[
\begin{align*}
tmp0[31:0] &= \text{INT32 } ((\text{DEST}[15:0] \times \text{SRC}[15:0]) \gg 14) + 1; \\
tmp1[31:0] &= \text{INT32 } ((\text{DEST}[31:16] \times \text{SRC}[31:16]) \gg 14) + 1; \\
tmp2[31:0] &= \text{INT32 } ((\text{DEST}[47:32] \times \text{SRC}[47:32]) \gg 14) + 1; \\
tmp3[31:0] &= \text{INT32 } ((\text{DEST}[63:48] \times \text{SRC}[63:48]) \gg 14) + 1; \\
tmp4[31:0] &= \text{INT32 } ((\text{DEST}[79:64] \times \text{SRC}[79:64]) \gg 14) + 1; \\
tmp5[31:0] &= \text{INT32 } ((\text{DEST}[95:80] \times \text{SRC}[95:80]) \gg 14) + 1; \\
tmp6[31:0] &= \text{INT32 } ((\text{DEST}[111:96] \times \text{SRC}[111:96]) \gg 14) + 1; \\
tmp7[31:0] &= \text{INT32 } ((\text{DEST}[127:112] \times \text{SRC}[127:112]) \gg 14) + 1; \\
\text{DEST}[15:0] &= \text{temp0}[16:1]; \\
\text{DEST}[31:16] &= \text{temp1}[16:1]; \\
\text{DEST}[47:32] &= \text{temp2}[16:1]; \\
\text{DEST}[63:48] &= \text{temp3}[16:1]; \\
\text{DEST}[79:64] &= \text{temp4}[16:1]; \\
\text{DEST}[95:80] &= \text{temp5}[16:1]; \\
\text{DEST}[111:96] &= \text{temp6}[16:1]; \\
\text{DEST}[127:112] &= \text{temp7}[16:1]; \\
\end{align*}
\]

VPMULHRSW (VEX.128 encoded version)

\[
\begin{align*}
tmp0[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[15:0] \times \text{SRC2}[15:0]) \gg 14) + 1 \\
tmp1[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[31:16] \times \text{SRC2}[31:16]) \gg 14) + 1 \\
tmp2[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[47:32] \times \text{SRC2}[47:32]) \gg 14) + 1 \\
tmp3[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[63:48] \times \text{SRC2}[63:48]) \gg 14) + 1 \\
tmp4[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[79:64] \times \text{SRC2}[79:64]) \gg 14) + 1 \\
tmp5[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[95:80] \times \text{SRC2}[95:80]) \gg 14) + 1 \\
tmp6[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[111:96] \times \text{SRC2}[111:96]) \gg 14) + 1 \\
tmp7[31:0] &\leftarrow \text{INT32 } ((\text{SRC1}[127:112] \times \text{SRC2}[127:112]) \gg 14) + 1 \\
\text{DEST}[15:0] &\leftarrow \text{temp0}[16:1] \\
\text{DEST}[31:16] &\leftarrow \text{temp1}[16:1] \\
\text{DEST}[47:32] &\leftarrow \text{temp2}[16:1] \\
\end{align*}
\]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[MAXVL-1:128] ← 0

VPMULHRSW (VEX.256 encoded version)

temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1

temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1

temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1

temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1

temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1

temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1

Temp[6:31] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1

temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1

temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1

temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1

temp10[31:0] ← INT32 ((SRC1[175:160] * SRC2[175:160]) >>14) + 1

temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1

temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1

temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1

temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1

temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240]) >>14) + 1

DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[143:128] ← temp8[16:1]
DEST[159:144] ← temp9[16:1]
DEST[175:160] ← temp10[16:1]
DEST[191:176] ← temp11[16:1]
DEST[207:192] ← temp12[16:1]
DEST[223:208] ← temp13[16:1]
DEST[239:224] ← temp14[16:1]
DEST[255:240] ← temp15[16:1]
DEST[MAXVL-1:256] ← 0

VPMULHRSW (EVEX encoded version)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      temp[31:0] ← ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
      DEST[i+15:i] ← tmp[16:1]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*

PMULHRSW — Packed Multiply High with Round and Scale
ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+15:i] \leftarrow 0
  FI
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
PMULHRSW: __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW: __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW:__m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
PMULHUW—Multiply Packed Unsigned Integers and Store High Result

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E4 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Multiply the packed unsigned word integers in ( mn1 ) and ( mn2/m64 ), and store the high 16 bits of the results in ( mn1 ).</td>
</tr>
<tr>
<td>66 0F E4 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed unsigned word integers in ( xmm1 ) and ( xmm2/m128 ), and store the high 16 bits of the results in ( xmm1 ).</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG E4 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed unsigned word integers in ( xmm2 ) and ( xmm3/m128 ), and store the high 16 bits of the results in ( xmm1 ).</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG E4 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed unsigned word integers in ( ymm2 ) and ( ymm3/m256 ), and store the high 16 bits of the results in ( ymm1 ).</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW xmm1 ( k1 ), xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply the packed unsigned word integers in ( xmm2 ) and ( xmm3/m128 ), and store the high 16 bits of the results in ( xmm1 ) under writemask ( k1 ).</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG E4 /r VPMULHUW ymm1 ( k1 ), ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply the packed unsigned word integers in ( ymm2 ) and ( ymm3/m256 ), and store the high 16 bits of the results in ( ymm1 ) under writemask ( k1 ).</td>
</tr>
<tr>
<td>VEX.NDS.512.66.0F.WIG E4 /r VPMULHUW zmm1 ( k1 ), zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Multiply the packed unsigned word integers in ( zmm2 ) and ( zmm3/m512 ), and store the high 16 bits of the results in ( zmm1 ) under writemask ( k1 ).</td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM( r, w )</td>
<td>ModRM( r/m ( r )</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM( w )</td>
<td>VEX.( vvvv ( r )</td>
<td>ModRM( r/m ( r )</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM( w )</td>
<td>VEX.( vvvv ( r )</td>
<td>ModRM( r/m ( r )</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (\( XMM8-\text{XMM15} \)).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is a MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.
VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

![Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands](image)

**Operation**

**PMULHUW (with 64-bit operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] & \leftarrow \text{DEST}[15:0] \times \text{SRC}[15:0]; \quad (* \text{Unsigned multiplication} *) \\
\text{TEMP1}[31:0] & \leftarrow \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] & \leftarrow \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] & \leftarrow \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{DEST}[15:0] & \leftarrow \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] & \leftarrow \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] & \leftarrow \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] & \leftarrow \text{TEMP3}[31:16];
\end{align*}
\]

**PMULHUW (with 128-bit operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] & \leftarrow \text{DEST}[15:0] \times \text{SRC}[15:0]; \quad (* \text{Unsigned multiplication} *) \\
\text{TEMP1}[31:0] & \leftarrow \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] & \leftarrow \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] & \leftarrow \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{TEMP4}[31:0] & \leftarrow \text{DEST}[79:64] \times \text{SRC}[79:64]; \\
\text{TEMP5}[31:0] & \leftarrow \text{DEST}[95:80] \times \text{SRC}[95:80]; \\
\text{TEMP6}[31:0] & \leftarrow \text{DEST}[111:96] \times \text{SRC}[111:96]; \\
\text{TEMP7}[31:0] & \leftarrow \text{DEST}[127:112] \times \text{SRC}[127:112]; \\
\text{DEST}[15:0] & \leftarrow \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] & \leftarrow \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] & \leftarrow \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] & \leftarrow \text{TEMP3}[31:16]; \\
\text{DEST}[79:64] & \leftarrow \text{TEMP4}[31:16]; \\
\text{DEST}[95:80] & \leftarrow \text{TEMP5}[31:16]; \\
\text{DEST}[111:96] & \leftarrow \text{TEMP6}[31:16]; \\
\text{DEST}[127:112] & \leftarrow \text{TEMP7}[31:16];
\end{align*}
\]
VPMULHUW (VEX.128 encoded version)
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[MAXVL-1:128] ← 0

PMULHUW (VEX.256 encoded version)
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAXVL-1:256] ← 0
PMULHUw (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      temp[31:0] ← SRC1[i+15:j] * SRC2[i+15:j]
      DEST[i+15:j] ← tmp[31:16]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:j] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:j] ← 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16(__mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16(__mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm128_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i _mm128_maskz_mulhi_epu16(__mmask8 k, __m128i a, __m128i b);
VPMULHUW __m64i _mm64_mulhi_epi16(__m64 a, __m64 b)
(V)PMULHUW __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)
VPMULHUW __m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
### PMULHW—Multiply Packed Signed Integers and Store High Result

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E5 r1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Multiply the packed signed word integers in xmm1 register and xmm2/m64, and store the high 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>PMULHW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG E5 r1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG E5 r1</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG E5 r1</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG E5 r1</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG E5 r1</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMrm/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.
VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**Operation**

**PMULHW (with 64-bit operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] &\leftarrow \text{DEST}[15:0] \times \text{SRC}[15:0]; (*\text{Signed multiplication}*) \\
\text{TEMP1}[31:0] &\leftarrow \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] &\leftarrow \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] &\leftarrow \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{DEST}[15:0] &\leftarrow \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] &\leftarrow \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] &\leftarrow \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] &\leftarrow \text{TEMP3}[31:16];
\end{align*}
\]

**PMULHW (with 128-bit operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] &\leftarrow \text{DEST}[15:0] \times \text{SRC}[15:0]; (*\text{Signed multiplication}*) \\
\text{TEMP1}[31:0] &\leftarrow \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] &\leftarrow \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] &\leftarrow \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{TEMP4}[31:0] &\leftarrow \text{DEST}[79:64] \times \text{SRC}[79:64]; \\
\text{TEMP5}[31:0] &\leftarrow \text{DEST}[95:80] \times \text{SRC}[95:80]; \\
\text{TEMP6}[31:0] &\leftarrow \text{DEST}[111:96] \times \text{SRC}[111:96]; \\
\text{TEMP7}[31:0] &\leftarrow \text{DEST}[127:112] \times \text{SRC}[127:112]; \\
\text{DEST}[15:0] &\leftarrow \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] &\leftarrow \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] &\leftarrow \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] &\leftarrow \text{TEMP3}[31:16]; \\
\text{DEST}[79:64] &\leftarrow \text{TEMP4}[31:16]; \\
\text{DEST}[95:80] &\leftarrow \text{TEMP5}[31:16]; \\
\text{DEST}[111:96] &\leftarrow \text{TEMP6}[31:16]; \\
\text{DEST}[127:112] &\leftarrow \text{TEMP7}[31:16]; \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\end{align*}
\]

**VPMULHW (VEX.128 encoded version)**

\[
\begin{align*}
\text{TEMP0}[31:0] &\leftarrow \text{SRC1}[15:0] \times \text{SRC2}[15:0]; (*\text{Signed Multiplication}*) \\
\text{TEMP1}[31:0] &\leftarrow \text{SRC1}[31:16] \times \text{SRC2}[31:16]; \\
\text{TEMP2}[31:0] &\leftarrow \text{SRC1}[47:32] \times \text{SRC2}[47:32]; \\
\text{TEMP3}[31:0] &\leftarrow \text{SRC1}[63:48] \times \text{SRC2}[63:48]; \\
\text{TEMP4}[31:0] &\leftarrow \text{SRC1}[79:64] \times \text{SRC2}[79:64]; \\
\text{TEMP5}[31:0] &\leftarrow \text{SRC1}[95:80] \times \text{SRC2}[95:80]; \\
\text{TEMP6}[31:0] &\leftarrow \text{SRC1}[111:96] \times \text{SRC2}[111:96]; \\
\text{TEMP7}[31:0] &\leftarrow \text{SRC1}[127:112] \times \text{SRC2}[127:112]; \\
\text{DEST}[15:0] &\leftarrow \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] &\leftarrow \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] &\leftarrow \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] &\leftarrow \text{TEMP3}[31:16]; \\
\text{DEST}[79:64] &\leftarrow \text{TEMP4}[31:16]; \\
\text{DEST}[95:80] &\leftarrow \text{TEMP5}[31:16]; \\
\text{DEST}[111:96] &\leftarrow \text{TEMP6}[31:16]; \\
\text{DEST}[127:112] &\leftarrow \text{TEMP7}[31:16];
\end{align*}
\]
PMULHW (VEX.256 encoded version)
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
DEST[15:0] ← TEMPO[31:16]
DEST[15:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMPO[31:16]
DEST[191:176] ← TEMP10[31:16]
DEST[207:192] ← TEMP11[31:16]
DEST[223:208] ← TEMP12[31:16]
DEST[239:224] ← TEMP13[31:16]
DEST[255:240] ← TEMP14[31:16]
DEST[MAXVL-1:256] ← 0

PMULHw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← tmp[31:16]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VPMULHW __m512i _mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i _mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i _mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i _mm256_mulhi_epi16(__m256i a, __m256i b);
VPMULHW __m256i _mm256_mask_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i _mm256_maskz_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i _mm_mask_mulhi_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i _mm_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHW:__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
(V)PMULHW:__m128i _mm_mulhi_epi16 (__m128i a, __m128i b)
VPMULHW:__m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 40 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 40 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 40 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 40 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 40 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 40 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply the packed dword signed integers in zmm2 and zmm3/m128/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 40 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 40 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 40 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Multiply the packed qword signed integers in zmm2 and zmm3/m128/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.
EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

**VPMULLQ (EVEX encoded versions)**

\[
(\text{KL, VL}) = (2, 128), (4, 256), (8, 512)
\]

FOR \( j \leftarrow 0 \) TO \( \text{KL}-1 \)

\[
i \leftarrow j * 64
\]

IF \( k1[j] \) OR *no writemask* THEN

\[
\text{IF (EVEX.b == 1) AND (SRC2 is memory*)}
\]

\[
\text{THEN Temp}[127:0] \leftarrow \text{SRC1}[i+63:i] * \text{SRC2}[63:0]
\]

ELSE Temp[127:0] \( \leftarrow \text{SRC1}[i+63:i] * \text{SRC2}[i+63:i] \)

FI;

\[
\text{DEST}[i+63:i] \leftarrow \text{Temp}[63:0]
\]

ELSE

\[
\text{IF *merging-masking* ; merging-masking}
\]

\[
\text{THEN *DEST}[i+63:i] \text{ remains unchanged*}
\]

ELSE ; zeroing-masking

\[
\text{DEST}[i+63:i] \leftarrow 0
\]

FI

ENDFOR

\[
\text{DEST}[	ext{MAXVL}-1:VL] \leftarrow 0
\]

**VPMULLD (EVEX encoded versions)**

\[
(\text{KL, VL}) = (4, 128), (8, 256), (16, 512)
\]

FOR \( j \leftarrow 0 \) TO \( \text{KL}-1 \)

\[
i \leftarrow j * 32
\]

IF \( k1[j] \) OR *no writemask* THEN

\[
\text{IF (EVEX.b = 1) AND (SRC2 is memory*)}
\]

\[
\text{THEN Temp}[63:0] \leftarrow \text{SRC1}[i+31:i] * \text{SRC2}[31:0]
\]

ELSE Temp[63:0] \( \leftarrow \text{SRC1}[i+31:i] * \text{SRC2}[i+31:i] \)

FI;

\[
\text{DEST}[i+31:i] \leftarrow \text{Temp}[31:0]
\]

ELSE

\[
\text{IF *merging-masking* ; merging-masking}
\]

\[
\text{THEN *DEST}[i+31:i] \text{ remains unchanged*}
\]

ELSE ; zeroing-masking

\[
\text{DEST}[i+31:i] \leftarrow 0
\]

FI

ENDFOR

\[
\text{DEST}[	ext{MAXVL}-1:VL] \leftarrow 0
\]
VPMULLD (VEX.256 encoded version)
Temp0[63:0] <- SRC1[31:0] * SRC2[31:0]
Temp2[63:0] <- SRC1[95:64] * SRC2[95:64]

DEST[31:0] <- Temp0[31:0]
DEST[63:32] <- Temp1[31:0]
DEST[95:64] <- Temp2[31:0]
DEST[127:96] <- Temp3[31:0]
DEST[159:128] <- Temp4[31:0]
DEST[191:160] <- Temp5[31:0]
DEST[223:192] <- Temp6[31:0]
DEST[255:224] <- Temp7[31:0]
DEST[MAXVL-1:256] <- 0

VPMULLD (VEX.128 encoded version)
Temp0[63:0] <- SRC1[31:0] * SRC2[31:0]
Temp2[63:0] <- SRC1[95:64] * SRC2[95:64]

DEST[31:0] <- Temp0[31:0]
DEST[63:32] <- Temp1[31:0]
DEST[95:64] <- Temp2[31:0]
DEST[127:96] <- Temp3[31:0]
DEST[MAXVL-1:128] <- 0

PMULLD (128-bit Legacy SSE version)
Temp0[63:0] <- DEST[31:0] * SRC[31:0]
Temp2[63:0] <- DEST[95:64] * SRC[95:64]

DEST[31:0] <- Temp0[31:0]
DEST[63:32] <- Temp1[31:0]
DEST[95:64] <- Temp2[31:0]
DEST[127:96] <- Temp3[31:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask16 k, __m512i a, __m512i b);
VPMULLD __m256i _mm256_mullo_epi32(__m256i a, __m256i b);
VPMULLD __m256i _mm256_mask_mullo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLD __m128i _mm128_mullo_epi32(__m128i a, __m128i b);
VPMULLD __m128i _mm128_mask_mullo_epi32(__mmask8 k, __m128i a, __m128i b);
VPMULLD __m128i _mm128_maskz_mullo_epi32(__mmask8 k, __m128i a, __m128i b);
VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__mmask8 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask8 k, __m512i a, __m512i b);

PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result
VPMULLQ __m512i _mm512_maskz_mullo_epi64( __mmask8 k, __m512i a, __m512i b);
VPMULLQ __m256i _mm256_mullo_epi64( __m256i a, __m256i b);
VPMULLQ __m256i _mm256_mask_mullo_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m256i _mm256_maskz_mullo_epi64( __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m128i _mm_mullo_epi64( __m128i a, __m128i b);
VPMULLQ __m128i _mm_mask_mullo_epi64( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLQ __m128i _mm_maskz_mullo_epi64( __mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
**PMULLW—Multiply Packed Signed Integers and Store Low Result**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D5 r1 PMULLW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Multiply the packed signed word integers in mm1 register and mm2/m64, and store the low 16 bits of the results in mm1.</td>
</tr>
<tr>
<td>66 0F D5 r PMULLW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG D5 r VPMULLW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG D5 r VPMULLW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG D5 r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG D5 r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG D5 r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.
VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

```
<table>
<thead>
<tr>
<th>SRC</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEST</td>
<td>Y3</td>
<td>Y2</td>
<td>Y1</td>
<td>Y0</td>
</tr>
<tr>
<td>TEMP</td>
<td>Z3 = X3 * Y3</td>
<td>Z2 = X2 * Y2</td>
<td>Z1 = X1 * Y1</td>
<td>Z0 = X0 * Y0</td>
</tr>
<tr>
<td>DEST</td>
<td>Z3[15:0]</td>
<td>Z2[15:0]</td>
<td>Z1[15:0]</td>
<td>Z0[15:0]</td>
</tr>
</tbody>
</table>
```

**PMULLW (with 64-bit operands)**
- \( \text{TEMP0}[31:0] \leftarrow \text{DEST}[15:0] \ast \text{SRC}[15:0]; \) (*Signed multiplication*)
- \( \text{TEMP1}[31:0] \leftarrow \text{DEST}[31:16] \ast \text{SRC}[31:16]; \)
- \( \text{TEMP2}[31:0] \leftarrow \text{DEST}[47:32] \ast \text{SRC}[47:32]; \)
- \( \text{TEMP3}[31:0] \leftarrow \text{DEST}[63:48] \ast \text{SRC}[63:48]; \)
- \( \text{DEST}[15:0] \leftarrow \text{TEMP0}[15:0]; \)
- \( \text{DEST}[31:16] \leftarrow \text{TEMP1}[15:0]; \)
- \( \text{DEST}[47:32] \leftarrow \text{TEMP2}[15:0]; \)
- \( \text{DEST}[63:48] \leftarrow \text{TEMP3}[15:0]; \)

**PMULLW (with 128-bit operands)**
- \( \text{TEMP0}[31:0] \leftarrow \text{DEST}[15:0] \ast \text{SRC}[15:0]; \) (*Signed multiplication*)
- \( \text{TEMP1}[31:0] \leftarrow \text{DEST}[31:16] \ast \text{SRC}[31:16]; \)
- \( \text{TEMP2}[31:0] \leftarrow \text{DEST}[47:32] \ast \text{SRC}[47:32]; \)
- \( \text{TEMP3}[31:0] \leftarrow \text{DEST}[63:48] \ast \text{SRC}[63:48]; \)
- \( \text{TEMP4}[31:0] \leftarrow \text{DEST}[79:64] \ast \text{SRC}[79:64]; \)
- \( \text{TEMP5}[31:0] \leftarrow \text{DEST}[95:80] \ast \text{SRC}[95:80]; \)
- \( \text{TEMP6}[31:0] \leftarrow \text{DEST}[111:96] \ast \text{SRC}[111:96]; \)
- \( \text{TEMP7}[31:0] \leftarrow \text{DEST}[127:112] \ast \text{SRC}[127:112]; \)
- \( \text{DEST}[15:0] \leftarrow \text{TEMP0}[15:0]; \)
- \( \text{DEST}[31:16] \leftarrow \text{TEMP1}[15:0]; \)
- \( \text{DEST}[47:32] \leftarrow \text{TEMP2}[15:0]; \)
- \( \text{DEST}[63:48] \leftarrow \text{TEMP3}[15:0]; \)
- \( \text{DEST}[79:64] \leftarrow \text{TEMP4}[15:0]; \)
- \( \text{DEST}[95:80] \leftarrow \text{TEMP5}[15:0]; \)
- \( \text{DEST}[111:96] \leftarrow \text{TEMP6}[15:0]; \)
- \( \text{DEST}[127:112] \leftarrow \text{TEMP7}[15:0]; \)
- \( \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \)
VPMULLW (VEX.128 encoded version)
Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[MAXVL-1:128] ← 0

PMULLW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
      DEST[i+15:i] ← temp[15:0]
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mullo_epi16 ( __m256i a, __m256i b);
VPMULLW __m256i _mm256_mask_mullo_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16 ( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm128_mullo_epi16 ( __m128i a, __m128i b);
VPMULLW __m128i _mm128_mask_mullo_epi16( __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm128_maskz_mullo_epi16 ( __mmask8 k, __m128i a, __m128i b);
PMULLW: __m64 __mm_mullo_epi16(__m64 m1, __m64 m2)
VPMULLW: __m128i __mm_mullo_epi16 ( __m128i a, __m128i b)
VPMULLW: __m256i __mm256_mullo_epi16 ( __m256i a, __m256i b);

Flags Affected
None.
SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
PMULUDQ—Multiply Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F F4 /r†</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.</td>
</tr>
<tr>
<td>PMULUDQ mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F F4 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.</td>
</tr>
<tr>
<td>PMULUDQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG F4 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.</td>
</tr>
<tr>
<td>VPMULUDQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG F4 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply packed unsigned doubleword integers in ymm1 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.</td>
</tr>
<tr>
<td>VPMULUDQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 F4 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMULUDQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 F4 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMULUDQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 F4 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMULUDQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned
quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

**Operation**

**PMULUDQ (with 64-Bit operands)**

\[ \text{DEST}[63:0] \leftarrow \text{DEST}[31:0] \times \text{SRC}[31:0]; \]

**PMULUDQ (with 128-Bit operands)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{DEST}[31:0] \times \text{SRC}[31:0]; \\
\text{DEST}[127:64] & \leftarrow \text{DEST}[95:64] \times \text{SRC}[95:64];
\end{align*}
\]

**VPMULUDQ (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{SRC1}[31:0] \times \text{SRC2}[31:0]; \\
\text{DEST}[127:64] & \leftarrow \text{SRC1}[95:64] \times \text{SRC2}[95:64]; \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0
\end{align*}
\]

**VPMULUDQ (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[63:0] & \leftarrow \text{SRC1}[31:0] \times \text{SRC2}[31:0]; \\
\text{DEST}[127:64] & \leftarrow \text{SRC1}[95:64] \times \text{SRC2}[95:64]; \\
\text{DEST}[191:128] & \leftarrow \text{SRC1}[159:128] \times \text{SRC2}[159:128]; \\
\text{DEST}[255:192] & \leftarrow \text{SRC1}[223:192] \times \text{SRC2}[223:192]; \\
\text{DEST}[\text{MAXVL}-1:256] & \leftarrow 0
\end{align*}
\]
VPMULUDQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31:i])
        ELSE DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VPMULUDQ __m512i _mm512_mul_epu32 (__m512i a, __m512i b);
VPMULUDQ __m512i _mm512_mask_mul_epu32 (__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i _mm512_maskz_mul_epu32 (__mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i _mm256_mul_epu32 (__m256i a, __m256i b);
VPMULUDQ __m256i _mm256_mask_mul_epu32 (__mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i _mm256_maskz_mul_epu32 (__mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i _mm128_mul_epu32 (__m128i a, __m128i b);
PMULUDQ:__m64 _mm_mul_su32 (__m64 a, __m64 b)
(V)PMULUDQ:__m128i _mm_mul_epu32 ( __m128i a, __m128i b)
VPMULUDQ:__m256i _mm256_mul_epu32( __m256i a, __m256i b);

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
### POP—Pop a Value from the Stack

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8F /0</td>
<td>POP r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into m16; increment stack pointer.</td>
</tr>
<tr>
<td>8F /0</td>
<td>POP r/m32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into m32; increment stack pointer.</td>
</tr>
<tr>
<td>8F /0</td>
<td>POP r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack into m64; increment stack pointer. Cannot encode 32-bit operand size.</td>
</tr>
<tr>
<td>5B+  rw</td>
<td>POP r16</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into r16; increment stack pointer.</td>
</tr>
<tr>
<td>5B+  rd</td>
<td>POP r32</td>
<td>0</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into r32; increment stack pointer.</td>
</tr>
<tr>
<td>5B+  rd</td>
<td>POP r64</td>
<td>0</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into r64; increment stack pointer. Cannot encode 32-bit operand size.</td>
</tr>
<tr>
<td>1F</td>
<td>POP DS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop top of stack into DS; increment stack pointer.</td>
</tr>
<tr>
<td>07</td>
<td>POP ES</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop top of stack into ES; increment stack pointer.</td>
</tr>
<tr>
<td>17</td>
<td>POP SS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop top of stack into SS; increment stack pointer.</td>
</tr>
<tr>
<td>0F A1</td>
<td>POP FS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into FS; increment stack pointer by 16 bits.</td>
</tr>
<tr>
<td>0F A1</td>
<td>POP FS</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into FS; increment stack pointer by 32 bits.</td>
</tr>
<tr>
<td>0F A1</td>
<td>POP FS</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack into FS; increment stack pointer by 64 bits.</td>
</tr>
<tr>
<td>0F A9</td>
<td>POP GS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into GS; increment stack pointer by 16 bits.</td>
</tr>
<tr>
<td>0F A9</td>
<td>POP GS</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into GS; increment stack pointer by 32 bits.</td>
</tr>
<tr>
<td>0F A9</td>
<td>POP GS</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack into GS; increment stack pointer by 64 bits.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- **Address size.** The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).
The address size is used only when writing to a destination operand in memory.

- **Operand size.** The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).
  
The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- **Stack-address size.** Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.
  
The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

Loading the SS register with a POP instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (POP ESP) before an event can be delivered. See Section 6.8.3, "Masking Exceptions and Interrupts When Switching Stacks,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POps using 32-bit operands are not encodable and POps to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF StackAddrSize = 32
  THEN
    IF OperandSize = 32
      THEN
        DEST ← SS:ESP; (* Copy a doubleword *)
        ESP ← ESP + 4;
      ELSE (* OperandSize = 16*)
        DEST ← SS:ESP; (* Copy a word *)
        ESP ← ESP + 2;
    FI;
    ELSE IF StackAddrSize = 64
      THEN
        IF OperandSize = 64
          THEN
...
LOADING Instructions

DEST ← SS:RSP; (* Copy quadword *)
RSP ← RSP + 8;
ELSE (* OperandSize = 16*)
  DEST ← SS:RSP; (* Copy a word *)
  RSP ← RSP + 2;
FI;
FI;
ELSE StackAddrSize = 16
  THEN
    IF OperandSize = 16
    THEN
      DEST ← SS:SP; (* Copy a word *)
      SP ← SP + 2;
      ELSE (* OperandSize = 32 *)
      DEST ← SS:SP; (* Copy a doubleword *)
      SP ← SP + 4;
    FI;
  FI;
FI;

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

64-BIT_MODE
IF FS, or GS is loaded with non-NULL selector;
  THEN
    IF segment selector index is outside descriptor table limits
    OR segment is not a data or readable code segment
    OR ((segment is a data or nonconforming code segment)
      AND (both RPL and CPL > DPL))
    THEN #GP(selector);
    IF segment not marked present
    THEN #NP(selector);
    ELSE
      SegmentRegister ← segment selector;
      SegmentRegister ← segment descriptor;
    FI;
FI;
IF FS, or GS is loaded with a NULL selector;
  THEN
    SegmentRegister ← segment selector;
    SegmentRegister ← segment descriptor;
FI;

PRETECTED MODE OR COMPATIBILITY MODE;
IF SS is loaded;
  THEN
    IF segment selector is NULL
    THEN #GP(0);
    IF segment selector index is outside descriptor table limits
    or segment selector's RPL ≠ CPL
or segment is not a writable data segment
or DPL ≠ CPL
   THEN #GP(selector);
FI;
IF segment not marked present
   THEN #SS(selector);
ELSE
   SS ← segment selector;
   SS ← segment descriptor;
FI;
FI;
IF DS, ES, FS, or GS is loaded with non-NULL selector;
   THEN
   IF segment selector index is outside descriptor table limits
   or segment is not a data or readable code segment
   or ((segment is a data or nonconforming code segment)
   and (both RPL and CPL > DPL))
   THEN #GP(selector);
FI;
IF segment not marked present
   THEN #NP(selector);
ELSE
   SegmentRegister ← segment selector;
   SegmentRegister ← segment descriptor;
FI;
FI;
IF DS, ES, FS, or GS is loaded with a NULL selector
   THEN
   SegmentRegister ← segment selector;
   SegmentRegister ← segment descriptor;
FI;

Flags Affected
None.

Protected Mode Exceptions

#GP(0) If attempt is made to load SS register with NULL segment selector.
   If the destination operand is in a non-writable segment.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#GP(selector) If segment selector index is outside descriptor table limits.
   If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
   If the SS register is being loaded and the segment pointed to is a non-writable data segment.
   If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
   If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0) If the current top of stack is not within the stack segment.
    If a memory operand effective address is outside the SS segment limit.
#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.
#NP If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same as for protected mode exceptions.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
#SS(0) If the stack address is in a non-canonical form.
#GP(selector) If the descriptor is outside the descriptor table limit.
    If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
    If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code) If a page fault occurs.
#NP If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD If the LOCK prefix is used.
**POPA/POPAD—Pop All General-Purpose Registers**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>POPA</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop DI, SI, BP, BX, DX, CX, and AX.</td>
</tr>
<tr>
<td>61</td>
<td>POPAD</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment’s segment descriptor determines the operand-size attribute.)

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

**Operation**

IF 64-Bit Mode
    THEN
        #UD;
ELSE
    IF OperandSize = 32 (* Instruction = POPAD *)
        THEN
            EDI ← Pop();
            ESI ← Pop();
            EBP ← Pop();
            Increment ESP by 4; (* Skip next 4 bytes of stack *)
            EBX ← Pop();
            EDX ← Pop();
            ECX ← Pop();
            EAX ← Pop();
        ELSE (* OperandSize = 16, instruction = POPA *)
            DI ← Pop();
            SI ← Pop();
            BP ← Pop();
            Increment ESP by 2; (* Skip next 2 bytes of stack *)
            BX ← Pop();
            DX ← Pop();
            CX ← Pop();
            AX ← Pop();
        FI;
    FI;
Flags Affected
None.

Protected Mode Exceptions
#SS(0) If the starting or ending stack address is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#SS If the starting or ending stack address is not within the stack segment.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#SS(0) If the starting or ending stack address is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same as for protected mode exceptions.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
POPCNT — Return the Count of Number of Bits Set to 1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F B8 /r</td>
<td>POPCNT r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>POPCNT on r/m16</td>
</tr>
<tr>
<td>F3 0F B8 /r</td>
<td>POPCNT r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>POPCNT on r/m32</td>
</tr>
<tr>
<td>F3 REX.W 0F B8 /r</td>
<td>POPCNT r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>POPCNT on r/m64</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

Operation

Count = 0;
For (i=0; i < OperandSize; i++)
{  IF (SRC[i] = 1) // i'th bit
    THEN Count++ ; FI;
}
DEST ← Count;

Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared.

Intel C/C++ Compiler Intrinsic Equivalent

POPCNT: int _mm_popcnt_u32(unsigned int a);
POPCNT: int64_t _mm_popcnt_u64(unsigned __int64 a);

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
If LOCK prefix is used.
Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
If LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
If LOCK prefix is used.
POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9D</td>
<td>POPF</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into lower 16 bits of EFLAGS.</td>
</tr>
<tr>
<td>9D</td>
<td>POPFD</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into EFLAGS.</td>
</tr>
<tr>
<td>9D</td>
<td>POPFQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack and zero-extend into RFLAGS.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD/PUSHFQ instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See Table 4-15 and the key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF\(^{1}\), VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. POPF, however, is not affected by CR4.PVI.)

In 64-bit mode, the mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack. Reserved bits of RFLAGS (including the upper 32 bits of RFLAGS) are not affected.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the EFLAGS registers.

---

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.
**INSTRUCTION SET REFERENCE, M-U**

**Table 4-15. Effect of POPF/POPFD on the EFLAGS Register**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operand Size</th>
<th>CPL</th>
<th>IOPL</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Real-Address Mode</strong></td>
<td>16</td>
<td>0</td>
<td>0-3</td>
<td>N</td>
</tr>
<tr>
<td><strong>(CR0.PE = 0)</strong></td>
<td>32</td>
<td>0</td>
<td>0-3</td>
<td>N</td>
</tr>
<tr>
<td><strong>Protected, Compatibility,</strong></td>
<td>16</td>
<td>0</td>
<td>0-3</td>
<td>N</td>
</tr>
<tr>
<td>and 64-Bit Modes</td>
<td>16</td>
<td>1-3</td>
<td>&lt;CPL</td>
<td>N</td>
</tr>
<tr>
<td><strong>(CR0.PE = 1)</strong></td>
<td>32,64</td>
<td>0</td>
<td>0-3</td>
<td>N</td>
</tr>
<tr>
<td><strong>EFLAGS.VM = 0)</strong></td>
<td>32,64</td>
<td>1-3</td>
<td>&lt;CPL</td>
<td>N</td>
</tr>
<tr>
<td><strong>Virtual-8086</strong></td>
<td>16</td>
<td>3</td>
<td>0-2</td>
<td>X</td>
</tr>
<tr>
<td><strong>(CR0.PE = 1)</strong></td>
<td>32</td>
<td>3</td>
<td>0-2</td>
<td>X</td>
</tr>
<tr>
<td><strong>EFLAGS.VM = 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CR4.VME = 0)</strong></td>
<td>16</td>
<td>3</td>
<td>0-2</td>
<td>N/</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>3</td>
<td>0-2</td>
<td>N/</td>
</tr>
<tr>
<td><strong>VME</strong></td>
<td>16</td>
<td>3</td>
<td>0-2</td>
<td>N/</td>
</tr>
<tr>
<td><strong>(CR0.PE = 1)</strong></td>
<td>32</td>
<td>3</td>
<td>0-2</td>
<td>N/</td>
</tr>
<tr>
<td><strong>EFLAGS.VM = 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CR4.VME = 1)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack
3. #GP fault with no flag update if TF=1 in FLAGS value on stack

**Key**

<table>
<thead>
<tr>
<th>S</th>
<th>Updated from stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>SV</td>
<td>Updated from IF (bit 9) in FLAGS value on stack</td>
</tr>
<tr>
<td>N</td>
<td>No change in value</td>
</tr>
<tr>
<td>X</td>
<td>No EFLAGS update</td>
</tr>
<tr>
<td>0</td>
<td>Value is cleared</td>
</tr>
</tbody>
</table>

**Operation**

IF EFLAGS.VM = 0 (* Not in Virtual-8086 Mode *)

THEN IF CPL = 0 OR CR0.PE = 0

THEN

IF OperandSize = 32;

THEN

EFLAGS ← Pop(); (* 32-bit pop *)

(* All non-reserved flags except RF, VIP, VIF, and VM can be modified; VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)

ELSE IF (OperandSize = 64)

RFLAGS = Pop(); (* 64-bit pop *)

(* All non-reserved flags except RF, VIP, VIF, and VM can be modified; VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
ELSE (* OperandSize = 16 *)
    EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
    (* All non-reserved flags can be modified. *)
    FI;
ELSE (* CPL > 0 *)
    IF OperandSize = 32
        THEN
            IF CPL > IOPL
                THEN
                    EFLAGS ← Pop(); (* 32-bit pop *)
                    (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                    IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                ELSE
                    EFLAGS ← Pop(); (* 32-bit pop *)
                    (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                    IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                FI;
            ELSE IF (OperandSize = 64)
                IF CPL > IOPL
                    THEN
                        RFLAGS ← Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                        IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    ELSE
                        RFLAGS ← Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                        IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    FI;
                ELSE (* OperandSize = 16 *)
                    EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                    (* All non-reserved bits except IOPL can be modified; IOPL and all
                    reserved bits are unaffected. *)
                    FI;
            ELSE (* In virtual-8086 mode *)
                IF IOPL = 3
                    THEN
                        IF OperandSize = 32
                            THEN
                                EFLAGS ← Pop();
                                (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                                VIP, VIF, VM, IOPL and all reserved bits are unaffected. RF is cleared. *)
                            ELSE
                                EFLAGS[15:0] ← Pop(); FI;
                                (* All non-reserved bits except IOPL can be modified; IOPL and all
                                reserved bits are unaffected. *)
                            FI;
                    ELSE (* IOPL < 3 *)
                        IF (Operandsize = 32) OR (CR4.VME = 0)
                            THEN #GP(0); (* Trap to virtual-8086 monitor. *)
                        ELSE (* Operandsize = 16 and CR4.VME = 1 *)
                            tempFLAGS ← Pop();
                            IF (EFLAGS.VIP = 1 AND tempFLAGS[9] = 1) OR tempFLAGS[8] = 1
                                THEN #GP(0);
                            ELSE
POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

EFLAGS.VIF ← tempFLAGS[9];
EFLAGS[15:0] ← tempFLAGS;
(*) All non-reserved bits except IOPL and IF can be modified;
IOPL, IF, and all reserved bits are unaffected. *)

FI;
FI;
FI;

FI;

Flags Affected
All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0) If the top of stack is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS If the top of stack is not within the stack segment.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If IOPL < 3 and VME is not enabled.
If IOPL < 3 and the 32-bit operand size is used.
If IOPL < 3, EFLAGS.VIP = 1, and bit 9 (IF) is set in the FLAGS value on the stack.
If IOPL < 3 and bit 8 (TF) is set in the FLAGS value on the stack.
If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.

#SS(0) If the top of stack is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
### POR—Bitwise Logical OR

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| NP 0F EB /r
POR mm, mm/m64 | A | V/V | MMX | Bitwise OR of mm/m64 and mm. |
| 66 0F EB /r
POR xmm1, xmm2/m128 | A | V/V | SSE2 | Bitwise OR of xmm2/m128 and xmm1. |
| VEX.NDS.128.66.0F.WIG EB /r
VPOR xmm1, xmm2, xmm3/m128 | B | V/V | AVX | Bitwise OR of xmm2/m128 and xmm3. |
| VEX.NDS.256.66.0F.WIG EB /r
VPOR ymm1, ymm2, ymm3/m256 | B | V/V | AVX2 | Bitwise OR of ymm2/m256 and ymm3. |
| EVEX.NDS.128.66.0F.W0 EB /r
VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst | C | V/V | AVX512VL AVX512F | Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1. |
| EVEX.NDS.256.66.0F.W0 EB /r
VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst | C | V/V | AVX512VL AVX512F | Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1. |
| EVEX.NDS.512.66.0F.W0 EB /r
VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst | C | V/V | AVX512F | Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1. |
| EVEX.NDS.128.66.0F.W1 EB /r
VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst | C | V/V | AVX512VL AVX512F | Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1. |
| EVEX.NDS.256.66.0F.W1 EB /r
VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst | C | V/V | AVX512VL AVX512F | Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1. |
| EVEX.NDS.512.66.0F.W1 EB /r
VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst | C | V/V | AVX512F | Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1. |

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

Operation

POR (64-bit operand)
DEST ← DEST OR SRC

POR (128-bit Legacy SSE version)
DEST ← DEST OR SRC
DEST[MAXVL-1:128] ← 0

VPOR (VEX.128 encoded version)
DEST ← SRC1 OR SRC2
DEST[MAXVL-1:128] ← 0

VPOR (VEX.256 encoded version)
DEST ← SRC1 OR SRC2
DEST[MAXVL-1:256] ← 0

VPORD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask* THEN
   IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
      ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
   FI;
ELSE
   IF *merging-masking* ; merging-masking
      *DEST[i+31:i] remains unchanged*
   ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
   FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPORD __m512i __m512i_or_epi32(__m512i a, __m512i b);
VPORD __m512i __m512i_mask_or_epi32(__m512i s, __m512i mask16 k, __m512i a, __m512i b);
VPORD __m512i __m512i_maskz_or_epi32(__m512i mask16 k, __m512i a, __m512i b);
VPORD __m256i __m256i_or_epi32(__m256i a, __m256i b);
VPORD __m256i __m256i_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORD __m256i __m256i_maskz_or_epi32(__mmask8 k, __m256i a, __m256i b);
VPORD __m128i __m128i_or_epi32(__m128i a, __m128i b);
VPORD __m128i __m128i_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i __m128i_maskz_or_epi32(__mmask8 k, __m128i a, __m128i b);
VPORD __m64 __m64_or_epi64(__m64 m1, __m64 m2);
(V)POR: __m128i __m128i_or_epi64(__m128i a, __m128i b)
VPOR: __m256i __m256i_or_epi64(__m256i a, __m256i b)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
PREFETCHh—Prefetch Data Into Caches

**Description**

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- **T0** (temporal data)—prefetch data into all levels of the cache hierarchy.
- **T1** (temporal data with respect to first level cache misses)—prefetch data into level 2 cache and higher.
- **T2** (temporal data with respect to second level cache misses)—prefetch data into level 3 cache and higher, or an implementation-specific choice.
- **NTA** (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHh instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHh instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHh instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHh instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

FETCH (m8):

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 18 /1</td>
<td>PREFETCHT0 m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T0 hint.</td>
</tr>
<tr>
<td>0F 18 /2</td>
<td>PREFETCHT1 m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T1 hint.</td>
</tr>
<tr>
<td>0F 18 /3</td>
<td>PREFETCHT2 m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T2 hint.</td>
</tr>
<tr>
<td>0F 18 /0</td>
<td>PREFETCHNTA m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using NTA hint.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Intel C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch(char *p, int i)

The argument "*p" gives the address of the byte (and corresponding cache line) to be prefetched. The value "i" gives a constant (_MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, or _MM_HINT_NTA) that specifies the type of prefetch operation to be performed.

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

**Description**
Fetcheds the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates other cached copies in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHW instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**
FETCH_WITH_EXCLUSIVE_OwNERSHIP (m8);

**Flags Affected**
All flags are affected

**C/C++ Compiler Intrinsic Equivalent**
void __m_prefetchw( void * );

**Protected Mode Exceptions**

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
## PSADBW—Compute Sum of Absolute Differences

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 66 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the absolute differences of the packed unsigned byte integers from \textit{mm2} /m64 and \textit{mm1}; differences are then summed to produce an unsigned word integer result.</td>
</tr>
<tr>
<td>PSADBW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 66 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Computes the absolute differences of the packed unsigned byte integers from \textit{xmm2} /m128 and \textit{xmm1}; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.</td>
</tr>
<tr>
<td>PSADBW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG 66 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the absolute differences of the packed unsigned byte integers from \textit{xmm3} /m128 and \textit{xmm2}; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.</td>
</tr>
<tr>
<td>VPSADBW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG 66 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Computes the absolute differences of the packed unsigned byte integers from \textit{ymm3} /m256 and \textit{ymm2}; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.</td>
</tr>
<tr>
<td>VPSADBW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG 66 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Computes the absolute differences of the packed unsigned byte integers from \textit{xmm3} /m128 and \textit{xmm2}; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.</td>
</tr>
<tr>
<td>VPSADBW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td>AVX512BW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG 66 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Computes the absolute differences of the packed unsigned byte integers from \textit{ymm3} /m256 and \textit{ymm2}; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.</td>
</tr>
<tr>
<td>VPSADBW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td>AVX512BW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG 66 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Computes the absolute differences of the packed unsigned byte integers from \textit{zmm3} /m512 and \textit{zmm2}; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.</td>
</tr>
<tr>
<td>VPSADBW zmm1, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**


### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

![Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands](image-url)
Operation

**VPSADBW (EVEX encoded versions)**

VL = 128, 256, 512

TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])

(* Repeat operation for bytes 1 through 15 *)

TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])

DEST[15:0] ← SUM(TEMP0:TEMP7)

DEST[63:16] ← 000000000000H

DEST[79:64] ← SUM(TEMP8:TEMP15)

DEST[127:80] ← 000000000000H

IF VL >= 256

(* Repeat operation for bytes 16 through 31*)

TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])

DEST[143:128] ← SUM(TEMP16:TEMP23)

DEST[191:144] ← 000000000000H

DEST[207:192] ← SUM(TEMP24:TEMP31)

DEST[223:208] ← 000000000000H

Fi;

IF VL >= 512

(* Repeat operation for bytes 32 through 63*)

TEMP63 ← ABS(SRC1[511:504] - SRC2[511:504])

DEST[271:256] ← SUM(TEMP0:TEMP7)

DEST[319:272] ← 000000000000H

DEST[335:320] ← SUM(TEMP8:TEMP15)

DEST[383:336] ← 000000000000H

DEST[399:384] ← SUM(TEMP16:TEMP23)

DEST[447:400] ← 000000000000H

DEST[463:448] ← SUM(TEMP24:TEMP31)

DEST[511:464] ← 000000000000H

Fi;

DEST[MAXVL-1:VL] ← 0

**VPSADBW (VEX.256 encoded version)**

TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])

(* Repeat operation for bytes 2 through 30*)

TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])

DEST[15:0] ← SUM(TEMP0:TEMP7)

DEST[63:16] ← 000000000000H

DEST[79:64] ← SUM(TEMP8:TEMP15)

DEST[127:80] ← 000000000000H

DEST[143:128] ← SUM(TEMP16:TEMP23)

DEST[191:144] ← 000000000000H

DEST[207:192] ← SUM(TEMP24:TEMP31)

DEST[223:208] ← 000000000000H

DEST[MAXVL-1:256] ← 0
**VPSADBW (VEX.128 encoded version)**

TEMP0 $\leftarrow$ ABS(SRC1[7:0] - SRC2[7:0])

(* Repeat operation for bytes 2 through 14 *)

TEMP15 $\leftarrow$ ABS(SRC1[127:120] - SRC2[127:120])

DEST[15:0] $\leftarrow$ SUM(TEMP0:TEMP7)

DEST[63:16] $\leftarrow$ 000000000000H

DEST[79:64] $\leftarrow$ SUM(TEMP8:TEMP15)

DEST[127:80] $\leftarrow$ 00000000000H

DEST[MAXVL-1:128] $\leftarrow$ 0

**PSADBW (128-bit Legacy SSE version)**

TEMP0 $\leftarrow$ ABS(DEST[7:0] - SRC[7:0])

(* Repeat operation for bytes 2 through 14 *)

TEMP15 $\leftarrow$ ABS(DEST[127:120] - SRC[127:120])

DEST[15:0] $\leftarrow$ SUM(TEMP0:TEMP7)

DEST[63:16] $\leftarrow$ 000000000000H

DEST[79:64] $\leftarrow$ SUM(TEMP8:TEMP15)

DEST[127:80] $\leftarrow$ 00000000000H

DEST[MAXVL-1:128] (Unmodified)

**PSADBW (64-bit operand)**

TEMP0 $\leftarrow$ ABS(DEST[7:0] - SRC[7:0])

(* Repeat operation for bytes 2 through 6 *)

TEMP7 $\leftarrow$ ABS(DEST[63:56] - SRC[63:56])

DEST[15:0] $\leftarrow$ SUM(TEMP0:TEMP7)

DEST[63:16] $\leftarrow$ 000000000000H

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSADBW __m512i _mm512_sad_epu8(__m512i a, __m512i b)

PSADBW:__m64 _mm_sad_epu8(__m64 a, __m64 b)

(V)PSADBW:__m128i _mm_sad_epu8(__m128i a, __m128i b)

VPSADBW:__m256i _mm256_sad_epu8(__m256i a, __m256i b)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.
**PSHUFB — Packed Shuffle Bytes**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 38 00 1r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Shuffle bytes in mm1 according to contents of mm2/m64.</td>
</tr>
<tr>
<td>66 38 00 1r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Shuffle bytes in xmm1 according to contents of xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 00 1r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle bytes in xmm2 according to contents of xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 00 1r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shuffle bytes in ymm2 according to contents of ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.WIG 00 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Shuffle bytes in xmm2 according to contents of xmm3/m128 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.WIG 00 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Shuffle bytes in ymm2 according to contents of ymm3/m256 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.WIG 00 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shuffle bytes in zmm2 according to contents of zmm3/m512 under write mask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**


---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask.
The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

**Operation**

**PSHUFB (with 64 bit operands)**

```
TEMP ← DEST
for i = 0 to 7 {
  if (SRC[(i * 8)+7] = 1) then
    DEST[(i*8)+7..(i*8)+0] ← 0;
  else
    index[2..0] ← SRC[(i*8)+2..(i*8)+0];
    DEST[(i*8)+7..(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
  endif;
}
```

**PSHUFB (with 128 bit operands)**

```
TEMP ← DEST
for i = 0 to 15 {
  if (SRC[(i * 8)+7] = 1) then
    DEST[(i*8)+7..(i*8)+0] ← 0;
  else
    index[3..0] ← SRC[(i*8)+3..(i*8)+0];
    DEST[(i*8)+7..(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
  endif
}
```

**VPSHUFB (VEX.128 encoded version)**

```
for i = 0 to 15 {
  if (SRC2[(i * 8)+7] = 1) then
    DEST[(i*8)+7..(i*8)+0] ← 0;
  else
    index[3..0] ← SRC2[(i*8)+3..(i*8)+0];
    DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
  endif
}
```

**DEST[MAXVL-1:128] ← 0**

**VPSHUFB (VEX.256 encoded version)**

```
for i = 0 to 15 {
  if (SRC2[(i * 8)+7] == 1) then
    DEST[(i*8)+7..(i*8)+0] ← 0;
  else
    index[3..0] ← SRC2[(i*8)+3..(i*8)+0];
    DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
  endif
}
if (SRC2[128 + (i * 8)+7] == 1) then
  DEST[128 + (i*8)+7..(i*8)+0] ← 0;
else
  index[3..0] ← SRC2[128 + (i*8)+3..(i*8)+0];
  DEST[128 + (i*8)+7..(i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
```
VPUSHUF (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

jmask ← (KL-1) & ~0xF  // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1  // dest
    IF kl[i] or no_masking
        index ← src.byte[j];
        IF index & 0x80
            Dest.byte[j] ← 0;
        ELSE
            index ← (index & 0xF) + (j & jmask);  // 16-element in-lane lookup
            Dest.byte[j] ← src.byte[index];
        ELSE if zeroing
            Dest.byte[j] ← 0;
    DEST[MAXVL-1:VL] ← 0;

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i __mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i __mm512_maskz_shuffle_epi8( __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_shuffle_epi8(__m256i a, __m256i b);
VPSHUFB __m256i __mm256_mask_shuffle_epi8(__mmask64 k, __m256i a, __m256i b);
VPSHUFB __m256i __mm256_maskz_shuffle_epi8( __mmask64 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm128_shuffle_epi8(__m128i a, __m128i b);
VPSHUFB __m128i __mm128_mask_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i __mm128_maskz_shuffle_epi8( __mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64 __mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB: __m128i __mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB: __m256i __mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
**PSHUFD—Shuffle Packed Doublewords**

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 70 /r ib PSHUFD xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1G 70 /r ib VPSHUFD xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F:W1G 70 /r ib VPSHUFD ymm1, ymm2/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shuffle the doublewords in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W0 70 /r ib VPSHUFD xmm1{k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W0 70 /r ib VPSHUFD ymm1{k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W0 70 /r ib VPSHUFD zmm1{k1}{z}, zmm2/m512/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.
The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

Note: EVEX.vvvv and VEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

**PSHUFD (128-bit Legacy SSE version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{SRC} \gg (\text{ORDER}[1:0] * 32))[31:0]; \\
\text{DEST}[63:32] & \leftarrow (\text{SRC} \gg (\text{ORDER}[3:2] * 32))[31:0]; \\
\text{DEST}[95:64] & \leftarrow (\text{SRC} \gg (\text{ORDER}[5:4] * 32))[31:0]; \\
\text{DEST}[127:96] & \leftarrow (\text{SRC} \gg (\text{ORDER}[7:6] * 32))[31:0]; \\
\text{DEST}[\text{MAXVL}-1:128] & \text{ (Unmodified)}
\end{align*}
\]

**VPSHUFD (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{SRC} \gg (\text{ORDER}[1:0] * 32))[31:0]; \\
\text{DEST}[63:32] & \leftarrow (\text{SRC} \gg (\text{ORDER}[3:2] * 32))[31:0]; \\
\text{DEST}[95:64] & \leftarrow (\text{SRC} \gg (\text{ORDER}[5:4] * 32))[31:0]; \\
\text{DEST}[127:96] & \leftarrow (\text{SRC} \gg (\text{ORDER}[7:6] * 32))[31:0]; \\
\text{DEST}[\text{MAXVL}-1:128] & \leftarrow 0
\end{align*}
\]
VPUSHUD (VEX.256 encoded version)
DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[MAXVL-1:256] ← 0

VPUSHUD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC is memory*)
    THEN TMP_SRC[i+31:i] ← SRC[31:0]
    ELSE TMP_SRC[i+31:i] ← SRC[i+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] ← (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
  TMP_DEST[95:64] ← (TMP_SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
  FI;
IF VL >= 256
  FI;
IF VL >= 512
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, int n);
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFD __m512i _mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, int n);
VPSHUFD __m256i _mm256_shuffle_epi32(__m256i a, int n);
VPSHUFD __m256i _mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFD __m256i _mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, int n);
VPSHUFD __m128i _mm_mask_shuffle_epi32(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFD __m128i _mm_maskz_shuffle_epi32(__mmask8 k, __m128i a, int n);
(V)PSHUFD:__m128i _mm_shuffle_epi32(__m128i a, int n)
VPSHUFD:__m256i _mm256_shuffle_epi32(__m256i a, const int n)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.
**PSHUFHW—Shuffle Packed High Words**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:WIG 70 /r ib VPUSHUFHW xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:WIG 70 /r ib VPUSHUFHW ymm1, ymm2/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F:WIG 70 /r ib VPUSHUFHW xmm1 [k1][z], xmm2/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F:WIG 70 /r ib VPUSHUFHW ymm1 [k1][z], ymm2/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F:WIG 70 /r ib VPUSHUFHW zmm1 [k1][z], zmm2/m512, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the destination operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.
EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**PSHUFHW (128-bit Legacy SSE version)**

DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] * 16))[79:64]
DEST[MAXVL-1:128] (Unmodified)

**VPSHUFHW (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
DEST[MAXVL-1:128] ← 0

**VPSHUFHW (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
DEST[MAXVL-1:256] ← 0

**VPSHUFHW (EVEX encoded versions)**

KL, VL = (8, 128), (16, 256), (32, 512)

IF VL >= 128

\[\text{TMP\_DEST}[63:0] \leftarrow \text{SRC1}[63:0]\]
\[\text{TMP\_DEST}[79:64] \leftarrow (\text{SRC1} >> (\text{imm}[1:0] * 16))[79:64]\]
\[\text{TMP\_DEST}[95:80] \leftarrow (\text{SRC1} >> (\text{imm}[3:2] * 16))[79:64]\]
\[\text{TMP\_DEST}[111:96] \leftarrow (\text{SRC1} >> (\text{imm}[5:4] * 16))[79:64]\]
\[\text{TMP\_DEST}[127:112] \leftarrow (\text{SRC1} >> (\text{imm}[7:6] * 16))[79:64]\]

Ft;

IF VL >= 256

\[\text{TMP\_DEST}[191:128] \leftarrow \text{SRC1}[191:128]\]
\[\text{TMP\_DEST}[207:192] \leftarrow (\text{SRC1} >> (\text{imm}[1:0] * 16))[207:192]\]
\[\text{TMP\_DEST}[223:208] \leftarrow (\text{SRC1} >> (\text{imm}[3:2] * 16))[207:192]\]
\[\text{TMP\_DEST}[239:224] \leftarrow (\text{SRC1} >> (\text{imm}[5:4] * 16))[207:192]\]
\[\text{TMP\_DEST}[255:240] \leftarrow (\text{SRC1} >> (\text{imm}[7:6] * 16))[207:192]\]

Ft;

IF VL >= 512

\[\text{TMP\_DEST}[319:256] \leftarrow \text{SRC1}[319:256]\]
\[\text{TMP\_DEST}[335:320] \leftarrow (\text{SRC1} >> (\text{imm}[1:0] * 16))[335:320]\]
TMP_DEST[463:448]  =  (SRC1 >> (imm[1:0] * 16))[463:448]

FI;
FOR j  =  0 TO KL-1
i  =  j * 16
IF k1[j] OR *no writemask*
    THEN DEST[i+15:i]  =  TMP_DEST[i+15:i];
ELSE
    IF *merging-masking* ; merging-masking
        THEN DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i]  =  0
FI
ENDFOR
DEST[MAXVL-1:VL]  =  0

Intel C/C++ Compiler Intrinsic Equivalent
VPSHUFHW __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFHW __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFHW __m512i _mm512_maskz_shufflehi_epi16(__mmask16 k, __m512i a, int n);
VPSHUFHW __m256i _mm256_shufflehi_epi16(__m256i a, const int n);
VPSHUFHW __m256i _mm256_mask_shufflehi_epi16(__mmask8 k, __m256i a, int n);
VPSHUFHW __m128i _mm128_shufflehi_epi16(__m128i a, const int n);
(V)PSHUFHW:__m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFHW:__m256i _mm256_shufflehi_epi16(__m256i a, const int n)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb
#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.
**PSHUFLW—Shuffle Packed Low Words**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 70 /r ib PSHUFLW xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.F2.0F:WIG 70 /r ib VPSHUFLW xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F2.0F:WIG 70 /r ib VPSHUFLW ymm1, ymm2/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:WIG 70 /r ib VPSHUFLW xmm1 [k1][z], xmm2/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512Fp</td>
<td>Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:WIG 70 /r ib VPSHUFLW ymm1 [k1][z], ymm2/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512Fp</td>
<td>Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:WIG 70 /r ib VPSHUFLW zmm1 [k1][z], zmm2/m512, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512Fp</td>
<td>Shuffle the low words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**128-bit Legacy SSE version:** The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded version:** The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**VEX.256 encoded version:** The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

**EVEX encoded version:** The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.
Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**PSHUFLW (128-bit Legacy SSE version)**

\[
\begin{align*}
\text{DEST}[15:0] &\leftarrow (\text{SRC} >> (\text{imm}[1:0] \times 16))[15:0] \\
\text{DEST}[31:16] &\leftarrow (\text{SRC} >> (\text{imm}[3:2] \times 16))[15:0] \\
\text{DEST}[47:32] &\leftarrow (\text{SRC} >> (\text{imm}[5:4] \times 16))[15:0] \\
\text{DEST}[63:48] &\leftarrow (\text{SRC} >> (\text{imm}[7:6] \times 16))[15:0] \\
\text{DEST}[127:64] &\leftarrow \text{SRC}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] &\text{(Unmodified)}
\end{align*}
\]

**VPSHUFLW (VEX.128 encoded version)**

\[
\begin{align*}
\text{DEST}[15:0] &\leftarrow (\text{SRC1} >> (\text{imm}[1:0] \times 16))[15:0] \\
\text{DEST}[31:16] &\leftarrow (\text{SRC1} >> (\text{imm}[3:2] \times 16))[15:0] \\
\text{DEST}[47:32] &\leftarrow (\text{SRC1} >> (\text{imm}[5:4] \times 16))[15:0] \\
\text{DEST}[63:48] &\leftarrow (\text{SRC1} >> (\text{imm}[7:6] \times 16))[15:0] \\
\text{DEST}[127:64] &\leftarrow \text{SRC}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] &\text{= 0}
\end{align*}
\]

**VPSHUFLW (VEX.256 encoded version)**

\[
\begin{align*}
\text{DEST}[15:0] &\leftarrow (\text{SRC1} >> (\text{imm}[1:0] \times 16))[15:0] \\
\text{DEST}[31:16] &\leftarrow (\text{SRC1} >> (\text{imm}[3:2] \times 16))[15:0] \\
\text{DEST}[47:32] &\leftarrow (\text{SRC1} >> (\text{imm}[5:4] \times 16))[15:0] \\
\text{DEST}[63:48] &\leftarrow (\text{SRC1} >> (\text{imm}[7:6] \times 16))[15:0] \\
\text{DEST}[127:64] &\leftarrow \text{SRC1}[127:64] \\
\text{DEST}[143:128] &\leftarrow (\text{SRC1} >> (\text{imm}[1:0] \times 16))[143:128] \\
\text{DEST}[159:144] &\leftarrow (\text{SRC1} >> (\text{imm}[3:2] \times 16))[143:128] \\
\text{DEST}[175:160] &\leftarrow (\text{SRC1} >> (\text{imm}[5:4] \times 16))[143:128] \\
\text{DEST}[191:176] &\leftarrow (\text{SRC1} >> (\text{imm}[7:6] \times 16))[143:128] \\
\text{DEST}[255:192] &\leftarrow \text{SRC1}[255:192] \\
\text{DEST}[\text{MAXVL}-1:256] &\text{= 0}
\end{align*}
\]

**VPSHUFLW (EVEX.U1.512 encoded version)**

\[
\begin{align*}
(KL, VL) &=(8, 128), (16, 256), (32, 512) \\
\text{IF } VL >= 128 \\
&\quad \text{TMP_DEST}[15:0] \leftarrow (\text{SRC1} >> (\text{imm}[1:0] \times 16))[15:0] \\
&\quad \text{TMP_DEST}[31:16] \leftarrow (\text{SRC1} >> (\text{imm}[3:2] \times 16))[15:0] \\
&\quad \text{TMP_DEST}[47:32] \leftarrow (\text{SRC1} >> (\text{imm}[5:4] \times 16))[15:0] \\
&\quad \text{TMP_DEST}[63:48] \leftarrow (\text{SRC1} >> (\text{imm}[7:6] \times 16))[15:0] \\
&\quad \text{TMP_DEST}[127:64] \leftarrow \text{SRC1}[127:64] \\
&\text{FI;} \\
\text{IF } VL >= 256 \\
&\quad \text{TMP_DEST}[143:128] \leftarrow (\text{SRC1} >> (\text{imm}[1:0] \times 16))[143:128] \\
&\quad \text{TMP_DEST}[159:144] \leftarrow (\text{SRC1} >> (\text{imm}[3:2] \times 16))[143:128] \\
&\quad \text{TMP_DEST}[175:160] \leftarrow (\text{SRC1} >> (\text{imm}[5:4] \times 16))[143:128] \\
&\quad \text{TMP_DEST}[191:176] \leftarrow (\text{SRC1} >> (\text{imm}[7:6] \times 16))[143:128] \\
&\quad \text{TMP_DEST}[255:192] \leftarrow \text{SRC1}[255:192] \\
&\text{FI;} \\
\text{IF } VL >= 512 \\
&\quad \text{TMP_DEST}[271:256] \leftarrow (\text{SRC1} >> (\text{imm}[1:0] \times 16))[271:256] \\
&\quad \text{TMP_DEST}[287:272] \leftarrow (\text{SRC1} >> (\text{imm}[3:2] \times 16))[271:256] \\
&\quad \text{TMP_DEST}[303:288] \leftarrow (\text{SRC1} >> (\text{imm}[5:4] \times 16))[271:256] \\
&\quad \text{TMP_DEST}[319:304] \leftarrow (\text{SRC1} >> (\text{imm}[7:6] \times 16))[271:256] \\
&\quad \text{TMP_DEST}[383:320] \leftarrow \text{SRC1}[383:320] \\
&\text{FI;}
\end{align*}
\]
TMP_DEST[511:448] ← SRC1[511:448]
FI;

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**
VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFLW __m512i _mm512_maskz_shufflelo_epi16( __mmask16 k, __m512i a, int n);
VPSHUFLW __m256i _mm256_shufflelo_epi16(__m256i a, const int n);
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFLW __m256i _mm256_maskz_shufflelo_epi16( __mmask8 k, __m256i a, int n);
(V)PSHUFLW:__m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW:__m256i _mm256_shufflelo_epi16(__m256i a, const int n)

**Flags Affected**
None.

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb
#UD  If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.
PSHUFW—Shuffle Packed Words

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 70 /r ib</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>PSHUFW mm1, mm2/m64, imm8</td>
<td></td>
<td></td>
<td>Shuffle the words in mm2/m64 based on the encoding in imm8 and store the result in mm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-16. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

\[
\begin{align*}
\text{DEST}[15:0] & \leftarrow (\text{SRC} \gg (\text{ORDER}[1:0] \times 16))[15:0]; \\
\text{DEST}[31:16] & \leftarrow (\text{SRC} \gg (\text{ORDER}[3:2] \times 16))[15:0]; \\
\text{DEST}[47:32] & \leftarrow (\text{SRC} \gg (\text{ORDER}[5:4] \times 16))[15:0]; \\
\text{DEST}[63:48] & \leftarrow (\text{SRC} \gg (\text{ORDER}[7:6] \times 16))[15:0];
\end{align*}
\]

### Intel C/C++ Compiler Intrinsic Equivalent

PSHUFW: 

```
__m64 _mm_shuffle_pi16(__m64 a, int n)
```

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Table 22-7, “Exception Conditions for SIMD/MMX Instructions with Memory Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
### INSTRUCTION SET REFERENCE, M-U

**PSIGNB/PSIGNW/PSIGND — Packed SIGN**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 08 /r¹</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed byte integers in mm1 depending on the corresponding sign in mm2/m64.</td>
</tr>
<tr>
<td>PSIGNB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 08 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed byte integers in xmm1 depending on the corresponding sign in xmm2/m128.</td>
</tr>
<tr>
<td>PSIGNB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 38 09 /r¹</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed word integers in mm1 depending on the corresponding sign in mm2/m64.</td>
</tr>
<tr>
<td>PSIGNW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 09 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed word integers in xmm1 depending on the corresponding sign in xmm2/m128.</td>
</tr>
<tr>
<td>PSIGNW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 38 0A /r¹</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed doubleword integers in mm1 depending on the corresponding sign in mm2/m128.</td>
</tr>
<tr>
<td>PSIGND mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 0A /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed doubleword integers in xmm1 depending on the corresponding sign in xmm2/m128.</td>
</tr>
<tr>
<td>PSIGND xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 08 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Negate/zero/preserve packed byte integers in xmm2 depending on the corresponding sign in xmm3/m128.</td>
</tr>
<tr>
<td>VPSIGNB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 09 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Negate/zero/preserve packed word integers in xmm2 depending on the corresponding sign in xmm3/m128.</td>
</tr>
<tr>
<td>VPSIGNW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.WIG 0A /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Negate/zero/preserve packed doubleword integers in xmm2 depending on the corresponding sign in xmm3/m128.</td>
</tr>
<tr>
<td>VPSIGND xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 08 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Negate packed byte integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.</td>
</tr>
<tr>
<td>VPSIGNB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 09 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Negate packed 16-bit integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.</td>
</tr>
<tr>
<td>VPSIGNW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.WIG 0A /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Negate packed doubleword integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.</td>
</tr>
<tr>
<td>VPSIGND ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers. When the source operand is a 128bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

**Operation**

**PSIGNB (with 64 bit operands)**

```plaintext
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes

IF (SRC[63:56] < 0 )
    DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
    DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
    DEST[63:56] ← DEST[63:56]
```

**PSIGNB (with 128 bit operands)**

```plaintext
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes

IF (SRC[127:120] < 0 )
    DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
    DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
    DEST[127:120] ← DEST[127:120]
```
VPSIGNB (VEX.128 encoded version)
DEST[127:0] ← BYTE_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSIGNB (VEX.256 encoded version)
DEST[255:0] ← BYTE_SIGN_256b(SRC1, SRC2)

PSIGNW (with 64 bit operands)
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
ELSEIF (SRC[15:0] > 0 )
    DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 3rd words
IF (SRC[63:48] < 0 )
    DEST[63:48] ← Neg(DEST[63:48])
ELSEIF (SRC[63:48] = 0 )
    DEST[63:48] ← 0
ELSEIF (SRC[63:48] > 0 )
    DEST[63:48] ← DEST[63:48]

PSIGNW (with 128 bit operands)
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
ELSEIF (SRC[15:0] > 0 )
    DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 7th words
IF (SRC[127:112] < 0 )
    DEST[127:112] ← Neg(DEST[127:112])
ELSEIF (SRC[127:112] = 0 )
    DEST[127:112] ← 0
ELSEIF (SRC[127:112] > 0 )

VPSIGNW (VEX.128 encoded version)
DEST[127:0] ← WORD_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSIGNW (VEX.256 encoded version)
DEST[255:0] ← WORD_SIGN(SRC1, SRC2)

PSIGND (with 64 bit operands)
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
IF (SRC[63:32] < 0 )
    DEST[63:32] ← Neg(DEST[63:32])
ELSEIF (SRC[63:32] = 0 )
    DEST[63:32] ← 0

ELSEIF (SRC[63:32] > 0 )

PSIGND (with 128 bit operands)
IF (SRC[31:0] < 0 )
  DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
  DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
  DEST[31:0] ← DEST[31:0]
Repeat operation for 2nd through 3rd double words
IF (SRC[127:96] < 0 )
  DEST[127:96] ← Neg(DEST[127:96])
ELSEIF (SRC[127:96] = 0 )
  DEST[127:96] ← 0
ELSEIF (SRC[127:96] > 0 )
  DEST[127:96] ← DEST[127:96]

VPSIGND (VEX.128 encoded version)
DEST[127:0] ← DWORD_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSIGND (VEX.256 encoded version)
DEST[255:0] ← DWORD_SIGN(SRC1, SRC2)

Intel C/C++ Compiler Intrinsic Equivalent
PSIGNB:   __m64 _mm_sign_pi8 (__m64 a, __m64 b)
(V)PSIGNB: __m128i _mm_sign_epi8 (__m128i a, __m128i b)
VPSIGNB: __m256i _mm256_sign_epi8 (__m256i a, __m256i b)
PSIGNW:   __m64 _mm_sign_pi16 (__m64 a, __m64 b)
(V)PSIGNW: __m128i _mm_sign_epi16 (__m128i a, __m128i b)
VPSIGNW: __m256i _mm256_sign_epi16 (__m256i a, __m256i b)
PSIGND:   __m64 _mm_sign_pi32 (__m64 a, __m64 b)
(V)PSIGND: __m128i _mm_sign_epi32 (__m128i a, __m128i b)
VPSIGND: __m256i _mm256_sign_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 4; additionally
#UD If VEX.L = 1.
PSLLDQ—Shift Double Quadword Left Logical

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 73 /7 ib PSLLDQ xmm1, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift xmm1 left by imm8 bytes while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F:WIG 73 /7 ib VPSLLDQ xmm1, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDD.256.66.0F:WIG 73 /7 ib VPSLLDQ ymm1, ymm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift ymm2 left by imm8 bytes while shifting in 0s and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDD.128.66.0F:WIG 73 /7 ib VPSLLDQ xmm1, xmm2/m128, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDD.256.66.0F:WIG 73 /7 ib VPSLLDQ ymm1, ymm2/m256, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDD.512.66.0F:WIG 73 /7 ib VPSLLDQ zmm1, zmm2/m512, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lane.

### Operation

**VPSLLDQ (EVEX.U1.512 encoded version)**

\[
\text{TEMP} \leftarrow \text{COUNT} \\
\text{IF} \ [\text{TEMP} > 15] \ \text{THEN} \ \text{TEMP} \leftarrow 16; \ 	ext{FI} \\
\text{DEST}[127:0] \leftarrow \text{SRC}[127:0] \ll (\text{TEMP} \times 8) \\
\text{DEST}[255:128] \leftarrow \text{SRC}[255:128] \ll (\text{TEMP} \times 8) \\
\text{DEST}[383:256] \leftarrow \text{SRC}[383:256] \ll (\text{TEMP} \times 8) \\
\text{DEST}[511:384] \leftarrow \text{SRC}[511:384] \ll (\text{TEMP} \times 8) \\
\text{DEST}[\text{MAXVL}-1:512] \leftarrow 0
\]
VPSLLDQ (VEX.256 and EVEX.256 encoded version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] << (TEMP * 8)
DEST[MAXVL-1:256] ← 0

VPSLLDQ (VEX.128 and EVEX.128 encoded version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[MAXVL-1:128] ← 0

PSLLDQ(128-bit Legacy SSE version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
(V)PSLLDQ: __m128i _mm_slli_si128 (__m128i a, int imm)
VPSLLDQ: __m256i _mm256_slli_si256 (__m256i a, const int imm)
VPSLLDQ: __m512i _mm512_bslli_epi128 (__m512i a, const int imm)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 7.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.
## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 16 /r</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>mm</code> left <code>mm/m64</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in <code>mm</code> left <code>mm/m64</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>66 0F 16 /r</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>xmm1</code> left by <code>xmm2/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in <code>xmm1</code> left by <code>xmm2/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>NP 0F 71 /6 ib</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>mm</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLW mm, imm8</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in <code>mm</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>66 0F 71 /6 ib</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>xmm1</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLW xmm1, imm8</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in <code>xmm1</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>NP 0F 72 /6 ib</td>
<td></td>
<td></td>
<td></td>
<td>Shift doublewords in <code>mm</code> left by <code>mm/m64</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLD mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in <code>mm</code> left by <code>mm/m64</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>66 0F 72 /6 ib</td>
<td></td>
<td></td>
<td></td>
<td>Shift doublewords in <code>xmm1</code> left by <code>xmm2/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doublewords in <code>xmm1</code> left by <code>xmm2/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>NP 0F 73 /6 ib</td>
<td></td>
<td></td>
<td></td>
<td>Shift doublewords in <code>mm</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLQ mm, imm8</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in <code>mm</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>66 0F 73 /6 ib</td>
<td></td>
<td></td>
<td></td>
<td>Shift doublewords in <code>xmm1</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>PSLLQ xmm1, imm8</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doublewords in <code>xmm1</code> left by <code>imm8</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 16 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VPSLLW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F.WIG 71 /6 ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VPSLLW xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 72 /6 ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VPSLLD xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td>Shift doublewords in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F.WIG 73 /6 ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VPSLLQ xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td>Shift doublewords in <code>xmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 16 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in <code>ymmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VPSLLW ymm1, ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>ymmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VEX.NDD.256.66.0F.WIG 71 /6 ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in <code>ymmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VPSLLQ ymm1, ymm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td>Shift words in <code>ymmm2</code> left by amount specified in <code>xmm3/m128</code> while shifting in <code>0s</code>.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG F2 /r</td>
<td>VPSLLD ymm1, ymm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------</td>
<td>---</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>VEX.NND.256.66.0F.Wig 72 /6 ib</td>
<td>VPSLLD ymm1, ymm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG F3 /r</td>
<td>VPSLLQ ymm1, ymm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
</tr>
<tr>
<td>VEX.NND.256.66.0F.Wig 73 /6 ib</td>
<td>VPSLLQ ymm1, ymm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG F1 /r</td>
<td>VPSLLW xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG F1 /r</td>
<td>VPSLLW ymm1 (k1)[z], ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG F1 /r</td>
<td>VPSLLW zmm1 (k1)[z], zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512BW</td>
</tr>
<tr>
<td>EVEX.NND.128.66.0F.Wig 71 /6 ib</td>
<td>VPSLLW xmm1 (k1)[z], xmm2/m128, imm8</td>
<td>E</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
</tr>
<tr>
<td>EVEX.NND.256.66.0F.Wig 71 /6 ib</td>
<td>VPSLLW ymm1 (k1)[z], ymm2/m256, imm8</td>
<td>E</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
</tr>
<tr>
<td>EVEX.NND.512.66.0F.Wig 71 /6 ib</td>
<td>VPSLLW zmm1 (k1)[z], zmm2/m512, imm8</td>
<td>E</td>
<td>V/V</td>
<td>AVX512BW</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W0 F2 /r</td>
<td>VPSLLD xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W0 F2 /r</td>
<td>VPSLLD ymm1 (k1)[z], ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W0 F2 /r</td>
<td>VPSLLD zmm1 (k1)[z], zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
<tr>
<td>EVEX.NND.128.66.0F.W0 72 /6 ib</td>
<td>VPSLLD xmm1 (k1)[z], xmm2/m128/m32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.NND.256.66.0F.W0 72 /6 ib</td>
<td>VPSLLD ymm1 (k1)[z], ymm2/m256/m32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.NND.512.66.0F.W0 72 /6 ib</td>
<td>VPSLLD zmm1 (k1)[z], zmm2/m512/m32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WF1 F3 /r</td>
<td>VPSLLQ xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WF1 F3 /r</td>
<td>VPSLLQ ymm1 (k1)[z], ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WF1 F3 /r</td>
<td>VPSLLQ zmm1 (k1)[z], zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>
PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

![Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand](image)

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.
128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (with 64-bit operand)
IF (COUNT > 15)
THEN
DEST[64:0] ← 0000000000000000H;
ELSE
DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
FI;

PSLLD (with 64-bit operand)
IF (COUNT > 31)
THEN
DEST[64:0] ← 0000000000000000H;
ELSE
DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
FI;

PSLLQ (with 64-bit operand)
IF (COUNT > 63)
THEN
DEST[64:0] ← 0000000000000000H;
ELSE
DEST ← ZeroExtend(DEST << COUNT);
FI;

LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
DEST[127:0] \leftarrow 00000000000000000000000000000000H
ELSE
  DEST[15:0] \leftarrow \text{ZeroExtend}(SRC[15:0] \ll \text{COUNT});
(* Repeat shift operation for 2nd through 7th words *)
  DEST[127:112] \leftarrow \text{ZeroExtend}(SRC[127:112] \ll \text{COUNT});
FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT \leftarrow \text{COUNT}_\text{SRC}[63:0];
IF (COUNT > 31)
  DEST[31:0] \leftarrow 0
ELSE
  DEST[31:0] \leftarrow \text{ZeroExtend}(SRC[31:0] \ll \text{COUNT});
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT \leftarrow \text{COUNT}_\text{SRC}[63:0];
IF (COUNT > 31)
  DEST[127:0] \leftarrow 00000000000000000000000000000000H
ELSE
  DEST[31:0] \leftarrow \text{ZeroExtend}(SRC[31:0] \ll \text{COUNT});
(* Repeat shift operation for 2nd through 3rd words *)
  DEST[127:96] \leftarrow \text{ZeroExtend}(SRC[127:96] \ll \text{COUNT});
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT \leftarrow \text{COUNT}_\text{SRC}[63:0];
IF (COUNT > 63)
  DEST[63:0] \leftarrow 0
ELSE
  DEST[63:0] \leftarrow \text{ZeroExtend}(SRC[63:0] \ll \text{COUNT});
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT \leftarrow \text{COUNT}_\text{SRC}[63:0];
IF (COUNT > 63)
  DEST[127:0] \leftarrow 00000000000000000000000000000000H
ELSE
  DEST[63:0] \leftarrow \text{ZeroExtend}(SRC[63:0] \ll \text{COUNT});
  DEST[127:64] \leftarrow \text{ZeroExtend}(SRC[127:64] \ll \text{COUNT});
FI;

LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT \leftarrow \text{COUNT}_\text{SRC}[63:0];
IF (COUNT > 15)
  DEST[127:0] \leftarrow 00000000000000000000000000000000H
  DEST[255:128] \leftarrow 00000000000000000000000000000000H
ELSE
  DEST[15:0] \leftarrow \text{ZeroExtend}(SRC[15:0] \ll \text{COUNT});
(* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ← ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] << COUNT)
    DEST[255:192] ← ZeroExtend(SRC[255:192] << COUNT);
FI;

VPSLLW (EVEX versions, xmm/m128)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
**VPSLLW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
   TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;

IF VL = 256
   TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;

IF VL = 512
   TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
   TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+15:i] = 0
      FI
   FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSLLW (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0;

**VPSLLW (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0;

**VPSLLW (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

**VPSLLW (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0
PSLLw (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSLLw (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSLLD (EVEX versions, imm8)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPSLLD (EVEX versions, xmm/m128)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
  TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
  TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
  TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
  TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPSLLD (ymm, ymm, xmm/m128) - VEX.256 encoding
DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0;

VPSLLD (ymm, imm8) - VEX.256 encoding
DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0;

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 encoding
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSLLD (xmm, imm8) - VEX.128 encoding
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

PSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSLLD (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSLLQ (EVEX versions, imm8)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:j] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
      ELSE DEST[i+63:j] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:j], imm8)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:j] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:j] ← 0
      FI
  FI;
ENDFOR

VPSLLQ (EVEX versions, xmm/m128)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
  TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
  TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
  TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
  TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\[ i \leftarrow j \times 64 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+63:j] \leftarrow \text{TMP\_DEST}[i+63:j] \)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+63:j] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\( \text{DEST}[i+63:j] \leftarrow 0 \)

FI

FI;

ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \)

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

\( \text{DEST}[255:0] \leftarrow \text{LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b}(\text{SRC1}, \text{SRC2}) \)

\( \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0; \)

VPSLLQ (ymm, imm8) - VEX.256 encoding

\( \text{DEST}[255:0] \leftarrow \text{LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b}(\text{SRC1}, \text{imm8}) \)

\( \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0; \)

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

\( \text{DEST}[127:0] \leftarrow \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{SRC1}, \text{SRC2}) \)

\( \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \)

VPSLLQ (xmm, imm8) - VEX.128 encoding

\( \text{DEST}[127:0] \leftarrow \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{SRC1}, \text{imm8}) \)

\( \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \)

PSLLQ (xmm, xmm, xmm/m128)

\( \text{DEST}[127:0] \leftarrow \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{DEST}, \text{SRC}) \)

\( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)

PSLLQ (xmm, imm8)

\( \text{DEST}[127:0] \leftarrow \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{DEST}, \text{imm8}) \)

\( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);

VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);

VPSLLD __m512i _mm512_maskz_slli_epi32(__mmask16 k, __m512i a, unsigned int imm);

VPSLLD __m256i _mm256_mask_slli_epi32(__mmask8 k, __m256i a, unsigned int imm);

VPSLLD __m256i _mm256_maskz_slli_epi32(__mmask8 k, __m256i a, unsigned int imm);

VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);

VPSLLD __m128i _mm_maskz_slli_epi32(__mmask8 k, __m128i a, unsigned int imm);

VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);

VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);

VPSLLD __m512i _mm512_maskz_sll_epi32(__mmask16 k, __m512i a, __m128i cnt);

VPSLLD __m256i _mm256_sll_epi32(__mmask8 k, __m256i a, __m128i cnt);

VPSLLD __m256i _mm256_mask_sll_epi32(__mmask8 k, __m256i a, __m128i cnt);

VPSLLD __m128i _mm_mask_sll_epi32(__mmask8 k, __m128i a, __m128i cnt);

VPSLLD __m128i _mm_maskz_sll_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_maskz_slli_epi64( __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64( __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64( __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_sll_epi64( __mmask16 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_sll_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_slli_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_slli_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_slli_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_slli_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_slli_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, int count);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);
VPSLLQ __m256i _mm256_slli_epi16 (__m256i m, __m128i cnt);

**Flags Affected**

None.

**Numeric Exceptions**

None.
Other Exceptions
VEX-encoded instructions:
   Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.
   Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSLLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:
   Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.
   Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.
## PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E1 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in mm right by mm/m64 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E1 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in xmm1 right by xmm2/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 71 /4 ib¹</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in mm right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 71 /4 ib</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in xmm1 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F E2 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in mm right by mm/m64 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E2 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doubleword in xmm1 right by xmm2/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 72 /4 ib¹</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in mm right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 72 /4 ib</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doublewords in xmm1 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG E1 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F.WIG 71 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in xmm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG E2 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F.WIG 72 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in xmm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG E1 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW ymm1, ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDD.256.66.0F.WIG 71 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in ymm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW ymm1, ymm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG E2 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW ymm1, ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDD.256.66.0F.WIG 72 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW ymm1, ymm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG E1 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>VPSRAW xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WIG E1 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>VPSRAW ymm1 [k1][z], ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WIG E1 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 right by amount specified in zmm3/m128 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>VPSRAW zmm1 [k1][z], zmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Type</td>
<td>Modifier</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>VPSRAW</td>
<td>E</td>
<td>V/V</td>
<td>Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAW</td>
<td>E</td>
<td>V/V</td>
<td>Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAW</td>
<td>E</td>
<td>V/V</td>
<td>Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAD</td>
<td>G</td>
<td>V/V</td>
<td>Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAD</td>
<td>G</td>
<td>V/V</td>
<td>Shift doublewords in ymm2/m256 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAD</td>
<td>G</td>
<td>V/V</td>
<td>Shift doublewords in zmm2/m512 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAQ</td>
<td>F</td>
<td>V/V</td>
<td>Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAQ</td>
<td>F</td>
<td>V/V</td>
<td>Shift quadwords in ymm2/m256 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAQ</td>
<td>F</td>
<td>V/V</td>
<td>Shift quadwords in zmm2/m512 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

---


---
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>NA</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>Full Mem</td>
<td>EVEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>F</td>
<td>Full</td>
<td>EVEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>G</td>
<td>Mem128</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

![Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand](image)

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.
EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

**Operation**

**PSRAW (with 64-bit operand)**

\[
\text{IF (COUNT} > 15)
\]
\[
\text{THEN COUNT } \leftarrow 16;
\]
\[
\text{FI;}
\]
\[
\text{DEST}[15:0] \leftarrow \text{SignExtend(Dest}[15:0] >> \text{COUNT});
\]
\[
(* \text{Repeat shift operation for 2nd and 3rd words} *)
\]
\[
\text{DEST}[63:48] \leftarrow \text{SignExtend(Dest}[63:48] >> \text{COUNT});
\]

**PSRAD (with 64-bit operand)**

\[
\text{IF (COUNT} > 31)
\]
\[
\text{THEN COUNT } \leftarrow 32;
\]
\[
\text{FI;}
\]
\[
\text{DEST}[31:0] \leftarrow \text{SignExtend(Dest}[31:0] >> \text{COUNT});
\]
\[
\text{DEST}[63:32] \leftarrow \text{SignExtend(Dest}[63:32] >> \text{COUNT});
\]

**ARITHMETIC_RIGHT_SHIFT_DWORDS1**(SRC, COUNT_SRC)

\[
\text{COUNT } \leftarrow \text{COUNT_SRC}[63:0];
\]
\[
\text{IF (COUNT} > 31)
\]
\[
\text{THEN}
\]
\[
\text{DEST}[31:0] \leftarrow \text{SignBit}
\]
\[
\text{ELSE}
\]
\[
\text{DEST}[31:0] \leftarrow \text{SignExtend(SRC}[31:0] >> \text{COUNT});
\]
\[
\text{FI;}
\]

**ARITHMETIC_RIGHT_SHIFT_QWORDS1**(SRC, COUNT_SRC)

\[
\text{COUNT } \leftarrow \text{COUNT_SRC}[63:0];
\]
\[
\text{IF (COUNT} > 63)
\]
\[
\text{THEN}
\]
\[
\text{DEST}[63:0] \leftarrow \text{SignBit}
\]
\[
\text{ELSE}
\]
\[
\text{DEST}[63:0] \leftarrow \text{SignExtend(SRC}[63:0] >> \text{COUNT});
\]
\[
\text{FI;}
\]

**ARITHMETIC_RIGHT_SHIFT_WORDS_256b**(SRC, COUNT_SRC)

\[
\text{COUNT } \leftarrow \text{COUNT_SRC}[63:0];
\]
\[
\text{IF (COUNT} > 15)
\]
\[
\text{THEN}
\]
\[
\text{COUNT } \leftarrow 16;
\]
\[
\text{FI;}
\]
\[
\text{DEST}[15:0] \leftarrow \text{SignExtend(SRC}[15:0] >> \text{COUNT});
\]
\[
(* \text{Repeat shift operation for 2nd through 15th words} *)
\]
\[
\text{DEST}[255:240] \leftarrow \text{SignExtend(SRC}[255:240] >> \text{COUNT});
\]
ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL) ; VL: 128b, 256b or 512b
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN COUNT ← 64;
FI;
DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] ← SignExtend(SRC[VL-1:VL-64] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(SRC[127:112] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);
VPSRAW (EVEX versions, xmm/m128)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPSRAW (EVEX versions, imm8)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPSRAW (ymm, ymm, xmm/m128) - VEX
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

VPSRAW (ymm, imm8) - VEX
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0

VPSRAW (xmm, xmm, xmm/m128) - VEX
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSRAW (xmm, imm8) - VEX
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

PSRAW (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(Dest, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRAW (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(Dest, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSRAD (EVEX versions, imm8)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI
  ENDIF
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPSRAD (EVEX versions, xmm/m128)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
  TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
  TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
  TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
  TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
Fl;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*; zeroing-masking
      DEST[i+31:i] ← 0
  FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPSRAD (ymm, ymm, xmm/m128) - VEX
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

VPSRAD (ymm, imm8) - VEX
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0

VPSRAD (xmm, xmm, xmm/m128) - VEX
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSRAD (xmm, imm8) - VEX
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

PSRAD (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRAD (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSRAQ (EVEX versions, imm8)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
      ELSE DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
    FI;
  ELSE
    IF *merging-masking*; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking*; zeroing-masking
      DEST[i+63:i] ← 0
  FI
ENDFOR
DEST[MAXVL-1:128] (Unmodified)
VPSRAQ (EVEX versions, xmm/m128)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\(\text{TMP\_DEST}[VL-1:0] \Leftarrow \text{ARITHMETIC\_RIGHT\_SHIFT\_QWORDS}(\text{SRC1}[VL-1:0], \text{SRC2}, VL)\)

\(\text{FOR } j \Leftarrow 0 \text{ TO } 7\)
\(\text{i} \Leftarrow j \times 64\)
\(\text{IF } k1[j] \text{ OR } *\text{no writemask}*\)
\(\text{THEN } \text{DEST}[i+63:i] \Leftarrow \text{TMP\_DEST}[i+63:i]\)
\(\text{ELSE}\)
\(\text{IF } *\text{merging-masking}* ; \text{merging-masking}\)
\(\text{THEN } *\text{DEST}[i+63:i] \text{ remains unchanged}*\)
\(\text{ELSE } *\text{zeroing-masking}* ; \text{zeroing-masking}\)
\(\text{DEST}[i+63:i] \Leftarrow 0\)
\(\text{FI}\)
\(\FI\)
\(\text{ENDFOR}\)
\(\text{DEST}[MAXVL-1:VL] \Leftarrow 0\)

**Intel C/C++ Compiler Intrinsic Equivalents**

\(\text{VPSRAD} \text{__m512i \_mm512_srai_epi32(___m512i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m512i \_mm512_mask_srai_epi32(___m512i s, __mmask16 k, ___m512i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m512i \_mm512_maskz_srai_epi32(___mmask8 k, ___m512i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m256i \_mm256_srai_epi32(__m256i s, __m256i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m256i \_mm256_mask_srai_epi32(___m256i s, ___mmask8 k, ___m256i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m256i \_mm256_maskz_srai_epi32(___mmask8 k, ___m256i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m128i \_mm128_srai_epi32(___m128i s, __m128i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m128i \_mm128_mask_srai_epi32(___mmask8 k, ___m128i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m128i \_mm128_maskz_srai_epi32(___mmask8 k, ___m128i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m512i \_mm512_sra_epi32(__m512i a, __m128i cnt);}\)
\(\text{VPSRAD} \text{__m512i \_mm512_mask_sra_epi32(___m512i s, __mmask16 k, __m512i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m512i \_mm512_maskz_sra_epi32(___mmask8 k, __m512i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m256i \_mm256_mask_sra_epi32(___m256i s, ___mmask8 k, ___m256i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m256i \_mm256_maskz_sra_epi32(___mmask8 k, ___m256i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m128i \_mm128_mask_sra_epi32(___m128i s, __m128i a, unsigned int imm);}\)
\(\text{VPSRAD} \text{__m128i \_mm128_maskz_sra_epi32(___mmask8 k, ___m128i a, unsigned int imm);}\)
\(\text{VPSRAW} \text{__m512i \_mm512_srai_epi16(__m512i a, unsigned int imm);}\)
\(\text{VPSRAW} \text{__m512i \_mm512_mask_srai_epi16(___m512i s, __mmask32 k, ___m512i a, unsigned int imm);}\)
VPSRAW __m512i __mm512_maskz_srai_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m256i __mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i __mm256_maskz_srai_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m128i __mm_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m128i __mm_maskz_srai_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m512i __mm512_sra_epi16(__m512i a, __m128i cnt);
VPSRAW __m512i __mm512_mask_sra_epi16(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m512i __mm512_maskz_sra_epi16(__mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m256i __mm256_mask_sra_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m256i __mm256_maskz_sra_epi16(__mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m128i __mm_mask_sra_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m128i __mm_maskz_sra_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRAW: __m64__mm_srai_pi16(__m64 m, int count)
PSRAW: __m64__mm_sra_pi16(__m64 m, __m64 count)
(V)PSRAW: __m128i__mm_srai_epi16(__m128i m, int count)
(V)PSRAW: __m128i__mm_sra_epi16(__m128i m, __m128i count)
VPSRAW: __m256i__mm256_srai_epi16(__m256i m, int count)
VPSRAW: __m256i__mm256_sra_epi16(__m256i m, __m256i count)
PSRAD: __m64__mm_sra_pi32(__m64 m, int count)
PSRAD: __m64__mm_sra_epi32(__m64 m, __m64 count)
(V)PSRAD: __m128i__mm_sra_epi32(__m128i m, int count)
(V)PSRAD: __m128i__mm_sra_epi32(__m128i m, __m128i count)
VPSRAD: __m256i__mm256_sra_epi32(__m256i m, int count)
VPSRAD: __m256i__mm256_sra_epi32(__m256i m, __m256i count)

**Flags Affected**
None.

**Numeric Exceptions**
None.

**Other Exceptions**
VEX-encoded instructions:
Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.
Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSRAW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSRAD/Q:
Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.
Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.
PSRLDQ—Shift Double Quadword Right Logical

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 73 /3 ib PSRLDQ xmm1, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift xmm1 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDD.128.66.0F:WIG 73 /3 ib VPSRLDQ xmm1, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift xmm2 right by imm8 bytes while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDD.256.66.0F:WIG 73 /3 ib VPSRLDQ ymm1, ymm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift ymm1 right by imm8 bytes while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.NDD.128.66.0F:WIG 73 /3 ib VPSRLDQ xmm1, xmm2/m128, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Shift xmm2/m128 right by imm8 bytes while shifting in 0s and store result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDD.256.66.0F:WIG 73 /3 ib VPSRLDQ ymm1, ymm2/m256, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Shift ymm2/m256 right by imm8 bytes while shifting in 0s and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDD.512.66.0F:WIG 73 /3 ib VPSRLDQ zmm1, zmm2/m512, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Shift zmm2/m512 right by imm8 bytes while shifting in 0s and store result in zmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (r, w)</td>
<td>immB</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>immB</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.
**Operation**

**VPSRLDQ (EVEX.512 encoded version)**

\[ \text{TEMP} \leftarrow \text{COUNT} \]

\[ \text{IF} (\text{TEMP} > 15) \text{ THEN } \text{TEMP} \leftarrow 16; \text{FI} \]

\[ \text{DEST}[127:0] \leftarrow \text{SRC}[127:0] \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[255:128] \leftarrow \text{SRC}[255:128] \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[383:256] \leftarrow \text{SRC}[383:256] \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[511:384] \leftarrow \text{SRC}[511:384] \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[\text{MAXVL}-1:512] \leftarrow 0; \]

**VPSRLDQ (VEX.256 and EVEX.256 encoded version)**

\[ \text{TEMP} \leftarrow \text{COUNT} \]

\[ \text{IF} (\text{TEMP} > 15) \text{ THEN } \text{TEMP} \leftarrow 16; \text{FI} \]

\[ \text{DEST}[127:0] \leftarrow \text{SRC}[127:0] \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[255:128] \leftarrow \text{SRC}[255:128] \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0; \]

**VPSRLDQ (VEX.128 and EVEX.128 encoded version)**

\[ \text{TEMP} \leftarrow \text{COUNT} \]

\[ \text{IF} (\text{TEMP} > 15) \text{ THEN } \text{TEMP} \leftarrow 16; \text{FI} \]

\[ \text{DEST} \leftarrow \text{SRC} \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0; \]

**PSRLDQ (128-bit Legacy SSE version)**

\[ \text{TEMP} \leftarrow \text{COUNT} \]

\[ \text{IF} (\text{TEMP} > 15) \text{ THEN } \text{TEMP} \leftarrow 16; \text{FI} \]

\[ \text{DEST} \leftarrow \text{DEST} \gg (\text{TEMP} * 8) \]

\[ \text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)} \]

**Intel C/C++ Compiler Intrinsic Equivalents**

(V)PSRLDQ __m128i __mm_srli_si128 (__m128i a, int imm)

VPSRLDQ __m256i __mm256_bsrli_epi128 (__m256i, const int)

VPSRLDQ __m512i __mm512_bsrli_epi128 (__m512i, int)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.
## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D1 / r&lt;sup&gt;1&lt;/sup&gt; PSRLW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in mm right by amount specified in mm/m64 while shifting in 0s.</td>
</tr>
<tr>
<td>66 0F D1 / r PSRLW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>NP OF 71 /2 ib&lt;sup&gt;1&lt;/sup&gt; PSRLW mm, imm8</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in mm right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>66 0F 71 /2 ib PSRLW xmm1, imm8</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in xmm1 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>NP 0F D2 / r&lt;sup&gt;1&lt;/sup&gt; PSRLD mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in mm right by amount specified in mm/m64 while shifting in 0s.</td>
</tr>
<tr>
<td>66 0F D2 / r PSRLD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>NP OF 72 /2 ib&lt;sup&gt;1&lt;/sup&gt; PSRLD mm, imm8</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in mm right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>66 0F 72 /2 ib PSRLD xmm1, imm8</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doublewords in xmm1 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>NP 0F D3 / r&lt;sup&gt;1&lt;/sup&gt; PSRLQ mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift mm right by amount specified in mm/m64 while shifting in 0s.</td>
</tr>
<tr>
<td>66 0F D3 / r PSRLQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>NP OF 73 /2 ib&lt;sup&gt;1&lt;/sup&gt; PSRLQ mm, imm8</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift mm right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>66 0F 73 /2 ib PSRLQ xmm1, imm8</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift quadwords in xmm1 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG D1 / r VPSRLW xmm1, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 71 /2 ib VPSRLW xmm1, xmm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in xmm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG D2 / r VPSRLD xmm1, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 72 /2 ib VPSRLD xmm1, xmm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in xmm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG D3 / r VPSRLQ xmm1, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 73 /2 ib VPSRLQ xmm1, xmm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift quadwords in xmm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG D1 / r VPSRLW ymm1, ymm2, ymm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 71 /2 ib VPSRLW ymm1, ymm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in ymm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>Instruction</td>
<td>VEX.NDS.256.66.0F.W1G</td>
<td>VPSRLD ymm1, ymm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------</td>
<td>--------------------------------</td>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>VEX.NDS.256.66.0F.W1G ib</td>
<td>VPSRLD ymm1, ymm2, imm8</td>
<td>D</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>VEX.NDS.256.66.0F.W1G</td>
<td>VPSRLQ ymm1, ymm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.256.66.0F.W1G</td>
<td>VPSRLQ ymm1 [k1][z], ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.256.66.0F.W1G</td>
<td>VPSRLQ ymm1 [k1][z], ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.256.66.0F.W1G</td>
<td>VPSRLQ zmm1 [k1][z], zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.128.66.0F.W0</td>
<td>VPSRLD xmm1 [k1][z], xmm2/m128, imm8</td>
<td>E</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.256.66.0F.W0</td>
<td>VPSRLD xmm1 [k1][z], ymm2/m256, imm8</td>
<td>E</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.512.66.0F.W0</td>
<td>VPSRLD zmm1 [k1][z], zmm2/m512, imm8</td>
<td>E</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.256.66.0F.W0</td>
<td>VPSRLD xmm1 [k1][z], xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.512.66.0F.W0</td>
<td>VPSRLD zmm1 [k1][z], zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.128.66.0F.W0</td>
<td>VPSRLD xmm1 [k1][z], xmm2/m128/32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.256.66.0F.W0</td>
<td>VPSRLD xmm1 [k1][z], ymm2/m256/32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.512.66.0F.W0</td>
<td>VPSRLD zmm1 [k1][z], zmm2/m512/32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.128.66.0F.W1</td>
<td>VPSRLQ xmm1 [k1][z], xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.256.66.0F.W1</td>
<td>VPSRLQ ymm1 [k1][z], ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
<tr>
<td></td>
<td>EVEX.NDS.512.66.0F.W1</td>
<td>VPSRLQ zmm1 [k1][z], zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
</tr>
</tbody>
</table>
Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.
128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (with 64-bit operand)
IF (COUNT > 15)
THEN
DEST[64:0] ← 0000000000000000H
ELSE
DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
FI;

PSRLD (with 64-bit operand)
IF (COUNT > 31)
THEN
DEST[64:0] ← 0000000000000000H
ELSE
DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
FI;

PSRLQ (with 64-bit operand)
IF (COUNT > 63)
THEN
DEST[64:0] ← 0000000000000000H
ELSE
DEST ← ZeroExtend(DEST >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
DEST[31:0] ← 0
ELSE
DEST[31:0] \leftarrow \text{ZeroExtend}(SRC[31:0] >> \text{COUNT});
FI;

\text{LOGICAL\_RIGHT\_SHIFT\_QWORDS1}(SRC, \text{COUNT\_SRC})
\text{COUNT} \leftarrow \text{COUNT\_SRC}[63:0];
\text{IF (COUNT > 63)}
\text{THEN}
\quad \text{DEST}[63:0] \leftarrow 0
\text{ELSE}
\quad \text{DEST}[63:0] \leftarrow \text{ZeroExtend}(SRC[63:0] >> \text{COUNT});
\text{FI};

\text{LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b}(SRC, \text{COUNT\_SRC})
\text{COUNT} \leftarrow \text{COUNT\_SRC}[63:0];
\text{IF (COUNT > 15)}
\text{THEN}
\quad \text{DEST}[255:0] \leftarrow 0
\text{ELSE}
\quad \text{DEST}[15:0] \leftarrow \text{ZeroExtend}(SRC[15:0] >> \text{COUNT});
\text{FI};

(* Repeat shift operation for 2nd through 15th words *)
\text{DEST}[255:240] \leftarrow \text{ZeroExtend}(SRC[255:240] >> \text{COUNT});

\text{LOGICAL\_RIGHT\_SHIFT\_WORDS}(SRC, \text{COUNT\_SRC})
\text{COUNT} \leftarrow \text{COUNT\_SRC}[63:0];
\text{IF (COUNT > 15)}
\text{THEN}
\quad \text{DEST}[127:0] \leftarrow 00000000000000000000000000000000H
\text{ELSE}
\quad \text{DEST}[15:0] \leftarrow \text{ZeroExtend}(SRC[15:0] >> \text{COUNT});
\text{FI};

(* Repeat shift operation for 2nd through 7th words *)
\text{DEST}[127:112] \leftarrow \text{ZeroExtend}(SRC[127:112] >> \text{COUNT});

\text{LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b}(SRC, \text{COUNT\_SRC})
\text{COUNT} \leftarrow \text{COUNT\_SRC}[63:0];
\text{IF (COUNT > 31)}
\text{THEN}
\quad \text{DEST}[255:0] \leftarrow 0
\text{ELSE}
\quad \text{DEST}[31:0] \leftarrow \text{ZeroExtend}(SRC[31:0] >> \text{COUNT});
\text{FI};

(* Repeat shift operation for 2nd through 3rd words *)
\text{DEST}[255:224] \leftarrow \text{ZeroExtend}(SRC[255:224] >> \text{COUNT});

\text{LOGICAL\_RIGHT\_SHIFT\_DWORDS}(SRC, \text{COUNT\_SRC})
\text{COUNT} \leftarrow \text{COUNT\_SRC}[63:0];
\text{IF (COUNT > 31)}
\text{THEN}
\quad \text{DEST}[127:0] \leftarrow 00000000000000000000000000000000H
\text{ELSE}
\quad \text{DEST}[31:0] \leftarrow \text{ZeroExtend}(SRC[31:0] >> \text{COUNT});
\text{FI};

(* Repeat shift operation for 2nd through 3rd words *)
\text{DEST}[127:96] \leftarrow \text{ZeroExtend}(SRC[127:96] >> \text{COUNT});
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
    DEST[255:192] ← ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
FI;

VPSRLW (EVEX versions, xmm/m128)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] = 0
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPSRLW (EVEX versions, imm8)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
  TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
  FI;

IF VL = 256
  TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
  FI;

IF VL = 512
  TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
  TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
  FI;

FOR j ← 0 TO KL-1
  i ← j * 16
  IF K1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 encoding

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0;

VPSRLW (ymm, imm8) - VEX.256 encoding

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0;

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 encoding

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSRLW (xmm, imm8) - VEX.128 encoding

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

PSRLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRLW (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
VPSRLD (EVEX versions, xmm/m128)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPSRLD (EVEX versions, imm8)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
            ELSE DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
        FI;
    ELSE IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 encoding
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0;

VPSRLD (ymm, imm8) - VEX.256 encoding
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0;
VPSRLD (xmm, xmm, xmm/m128) - VEX.128 encoding
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSRLD (xmm, imm8) - VEX.128 encoding
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

PSRLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRLD (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSRLQ (EVEX versions, xmm/m128)
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
   TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
   TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
   TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
   TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
      ELSE
         IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
         ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+63:i] ← 0
         FI
   FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPSRLQ (EVEX versions, imm8)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+63:j] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
            ELSE DEST[i+63:j] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:j], imm8)
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:j] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:j] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 encoding
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0;

VPSRLQ (ymm, imm8) - VEX.256 encoding
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0;

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

PSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDSDEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDSDEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalents**
VPSRLD __m512i __mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i __mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i __mm512_maskz_srli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i __mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m256i __mm256_maskz_srli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i __mm128_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i __mm128_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i __mm128_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i __mm128_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i __mm128_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i __mm128_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i __mm128_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);

VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__mmask8 k, __m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm256_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_maskz_srl_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_maskz_srl_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_maskz_srl_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, unsigned int imm);

Flags Affected

None.
**Numeric Exceptions**
None.

**Other Exceptions**
VEX-encoded instructions:
Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.
Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSRLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:
Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.
Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.
### PSUBB/PSUBW/PSUBD—Subtract Packed Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 8B /r&lt;sup&gt;1&lt;/sup&gt;</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Subtract packed byte integers in mm/m64 from packed byte integers in mm.</td>
</tr>
<tr>
<td>PSUBB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 8B /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Subtract packed byte integers in xmm2/m128 from packed byte integers in xmm1.</td>
</tr>
<tr>
<td>PSUBB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 89 /r&lt;sup&gt;1&lt;/sup&gt;</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Subtract packed word integers in mm/m64 from packed word integers in mm.</td>
</tr>
<tr>
<td>PSUBW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 89 /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Subtract packed word integers in xmm2/m128 from packed word integers in xmm1.</td>
</tr>
<tr>
<td>PSUBW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F F9 /r&lt;sup&gt;1&lt;/sup&gt;</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Subtract packed doubleword integers in mm/m64 from packed doubleword integers in mm.</td>
</tr>
<tr>
<td>PSUBD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F F9 /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Subtract packed doubleword integers in xmm2/m128 from packed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>PSUBD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG 8B /r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Subtract packed byte integers in xmm3/m128 from xmm2.</td>
</tr>
<tr>
<td>VPSUBB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG 89 /r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Subtract packed word integers in xmm3/m128 from xmm2.</td>
</tr>
<tr>
<td>VPSUBW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG FA /r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Subtract packed doubleword integers in xmm3/m128 from xmm2.</td>
</tr>
<tr>
<td>VPSUBD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG 8B /r</td>
<td>B V/V</td>
<td>AVX2</td>
<td></td>
<td>Subtract packed byte integers in ymm3/m256 from ymm2.</td>
</tr>
<tr>
<td>VPSUBB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG 89 /r</td>
<td>B V/V</td>
<td>AVX2</td>
<td></td>
<td>Subtract packed word integers in ymm3/m256 from ymm2.</td>
</tr>
<tr>
<td>VPSUBW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG F8 /r</td>
<td>C V/V</td>
<td>AVX512VL, AVX512BW</td>
<td></td>
<td>Subtract packed byte integers in ymm3/m128 from ymm2 and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBB xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG F8 /r</td>
<td>C V/V</td>
<td>AVX512VL, AVX512BW</td>
<td></td>
<td>Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBB ymm1 [k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG F8 /r</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td></td>
<td>Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBB zmm1 [k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG FA /r</td>
<td>C V/V</td>
<td>AVX512VL, AVX512BW</td>
<td></td>
<td>Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBW xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG FA /r</td>
<td>C V/V</td>
<td>AVX512VL, AVX512BW</td>
<td></td>
<td>Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBW ymm1 [k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG FA /r</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td></td>
<td>Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
INSTRUCTION SET REFERENCE, M-U

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two’s complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

NOTES:
VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation**

**PSUBB (with 64-bit operands)**

\[ \text{DEST}[7:0] \leftarrow \text{DEST}[7:0] \ – \text{SRC}[7:0]; \]

(* Repeat subtract operation for 2nd through 7th byte *)

\[ \text{DEST}[63:56] \leftarrow \text{DEST}[63:56] \ – \text{SRC}[63:56]; \]

**PSUBW (with 64-bit operands)**

\[ \text{DEST}[15:0] \leftarrow \text{DEST}[15:0] \ – \text{SRC}[15:0]; \]

(* Repeat subtract operation for 2nd and 3rd word *)

\[ \text{DEST}[63:48] \leftarrow \text{DEST}[63:48] \ – \text{SRC}[63:48]; \]

**PSUBD (with 64-bit operands)**

\[ \text{DEST}[31:0] \leftarrow \text{DEST}[31:0] \ – \text{SRC}[31:0]; \]

\[ \text{DEST}[63:32] \leftarrow \text{DEST}[63:32] \ – \text{SRC}[63:32]; \]

**PSUBD (with 128-bit operands)**

\[ \text{DEST}[31:0] \leftarrow \text{DEST}[31:0] \ – \text{SRC}[31:0]; \]

(* Repeat subtract operation for 2nd and 3rd doubleword *)

\[ \text{DEST}[127:96] \leftarrow \text{DEST}[127:96] \ – \text{SRC}[127:96]; \]

**VPSUBB (EVEX encoded versions)**

\[(KL, VL) = (16, 128), (32, 256), (64, 512)\]

\[\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]

\[ i \leftarrow j \times 8 \]

**IF k1[j] OR *no writemask*  
THEN \[ \text{DEST}[i+7:i] \leftarrow \text{SRC1}[i+7:i] \ – \text{SRC2}[i+7:i] \]
ELSE  
**IF *merging-masking*  
THEN *DEST[i+7:i] remains unchanged*  
ELSE *zeroing-masking*  
\[ \text{DEST}[i+7:i] = 0 \]

**FI**

\[ \text{FI}; \]

\[ \text{ENDFOR}; \]

\[ \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \]

**VPSUBW (EVEX encoded versions)**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]

\[ i \leftarrow j \times 16 \]

**IF k1[j] OR *no writemask*  
THEN \[ \text{DEST}[i+15:i] \leftarrow \text{SRC1}[i+15:i] \ – \text{SRC2}[i+15:i] \]
ELSE  
**IF *merging-masking*  
THEN *DEST[i+15:i] remains unchanged*  
ELSE *zeroing-masking*  
\[ \text{DEST}[i+15:i] = 0 \]

**FI**

\[ \text{FI}; \]

\[ \text{ENDFOR}; \]
THEN *DEST[i+15:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+15:i] = 0
FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPSUBD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0]
            ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPSUBB (VEX.256 encoded version)

DEST[7:0] ← SRC1[7:0]-SRC2[7:0]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]
DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[143:136] ← SRC1[143:136]-SRC2[143:136]
DEST[151:144] ← SRC1[151:144]-SRC2[151:144]
DEST[159:152] ← SRC1[159:152]-SRC2[159:152]
PSUBB/PSUBW/PSUBD—Subtract Packed Integers

VPSUBB (VEX.128 encoded version)
DEST[MAXVL-1:256] ← 0

VPSUBW (VEX.256 encoded version)
DEST[7:0] ← SRC1[7:0]-SRC2[7:0]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]
DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[MAXVL-1:128] ← 0

PSUBB (128-bit Legacy SSE version)
DEST[7:0] ← DEST[7:0]-SRC[7:0]
DEST[47:40] ← DEST[47:40]-SRC[47:40]
DEST[63:56] ← DEST[63:56]-SRC[63:56]
DEST[71:64] ← DEST[71:64]-SRC[71:64]
DEST[79:72] ← DEST[79:72]-SRC[79:72]
DEST[87:80] ← DEST[87:80]-SRC[87:80]
DEST[103:96] ← DEST[103:96]-SRC[103:96]
DEST[111:104] ← DEST[111:104]-SRC[111:104]
DEST[127:120] ← DEST[127:120]-SRC[127:120]
DEST[MAXVL-1:128] (Unmodified)

VPSUBW (VEX.256 encoded version)
DEST[15:0] ← SRC1[15:0]-SRC2[15:0]
DEST[31:16] ← SRC1[31:16]-SRC2[31:16]
DEST[79:64] ← SRC1[79:64]-SRC2[79:64]
DEST[95:80] ← SRC1[95:80]-SRC2[95:80]
DEST[111:96] ← SRC1[111:96]-SRC2[111:96]
VPSUBB/PSUBW/PSUBD—Subtract Packed Integers

VPSUBB (VEX.128 encoded version)
DEST[MAXVL-1:256] ← 0

VPSUBW (VEX.128 encoded version)
DEST[15:0] ← SRC[15:0] - SRC2[15:0]
DEST[79:64] ← SRC[79:64] - SRC2[79:64]
DEST[MAXVL-1:128] ← 0

PSUBB (128-bit Legacy SSE version)
DEST[MAXVL-1:128] ← 0

VPSUBD (VEX.256 encoded version)
DEST[31:0] ← SRC[31:0] - SRC2[31:0]
DEST[95:64] ← SRC[95:64] - SRC2[95:64]
DEST[MAXVL-1:256] ← 0

VPSUBD (VEX.128 encoded version)
DEST[31:0] ← SRC[31:0] - SRC2[31:0]
DEST[95:64] ← SRC[95:64] - SRC2[95:64]
DEST[MAXVL-1:128] ← 0

PSUBD (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] - SRC[31:0]

4-470 Vol. 2B
PSUBB/PSUBW/PSUBD—Subtract Packed Integers
**Intel C/C++ Compiler Intrinsic Equivalents**

VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_maskz_sub_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBB __m512i _mm512_mask_sub_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_maskz_sub_epi16(__mmask8 k, __m128i a, __m128i b);
VPSUBB __m512i _mm512_mask_sub_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi32(__mmask16 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi32(__mmask8 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_maskz_sub_epi32(__mmask8 k, __m128i a, __m128i b);

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPSUBD, see Exceptions Type E4.
EVEX-encoded VPSUBW, see Exceptions Type E4.nb.
### PSUBQ—Subtract Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F FB /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract quadword integer in mm1 from mm2/m64.</td>
</tr>
<tr>
<td>PSUBQ mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FB /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed quadword integers in xmm1 from xmm2/m128.</td>
</tr>
<tr>
<td>PSUBQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:W1G FB/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed quadword integers in xmm3/m128 from xmm2.</td>
</tr>
<tr>
<td>VPSUBQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:W1G FB/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed quadword integers in ymm3/m256 from ymm2.</td>
</tr>
<tr>
<td>VPSUBQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:W1B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBQ xmm1 {k1}[z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:W1B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBQ ymm1 {k1}[z], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:W1B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBQ zmm1 {k1}[z], zmm2, zmm3/m512/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.
VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation**

**PSUBQ (with 64-Bit operands)**

\[ \text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]; \]

**PSUBQ (with 128-Bit operands)**

\[ \text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]; \]
\[ \text{DEST}[127:64] \leftarrow \text{DEST}[127:64] - \text{SRC}[127:64]; \]

**VPSUBQ (VEX.128 encoded version)**

\[ \text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]; \]
\[ \text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] - \text{SRC2}[127:64]; \]
\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

**VPSUBQ (VEX.256 encoded version)**

\[ \text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]; \]
\[ \text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] - \text{SRC2}[127:64]; \]
\[ \text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] - \text{SRC2}[191:128]; \]
\[ \text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] - \text{SRC2}[255:192]; \]
\[ \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \]

**VPSUBQ (EVEX encoded versions)**

\((K, V) = (2, 128), (4, 256), (8, 512)\)

\[
\text{FOR } j \leftarrow 0 \text{ TO } K-1
\]
\[
i \leftarrow j * 64
\]

\[
\text{IF } k1[j] \text{ OR } \text{no writemask* THEN}
\]
\[
\text{IF (EVEX.b = 1) AND (SRC2 *is memory*)}
\]
\[
\text{THEN } \text{DEST}[i+63:j] \leftarrow \text{SRC1}[i+63:j] - \text{SRC2}[63:0]
\]
\[
\text{ELSE } \text{DEST}[i+63:j] \leftarrow \text{SRC1}[i+63:j] - \text{SRC2}[i+63:j]
\]
\[
\text{FI};
\]
\[
\text{ELSE}
\]
\[
\text{IF *merging-masking* ; merging-masking}
\]
\[
\text{THEN *DEST*[i+63:j] remains unchanged*}
\]
\[
\text{ELSE *zeroing-masking* ; zeroing-masking}
\]
\[
\text{DEST}[i+63:j] \leftarrow 0
\]
\[
\text{FI}
\]
\[
\text{ENDFOR;}
\]
\[
\text{DEST}[\text{MAXVL}-1:V] \leftarrow 0
\]
**Intel C/C++ Compiler Intrinsic Equivalents**

VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
VPSUBQ __m64__ _mm_sub_epi64(__m64 m1, __m64 m2)
(V)PSUBQ: __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ: __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

**Flags Affected**
None.

**Numeric Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPSUBQ, see Exceptions Type E4.
## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E8 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSUBSB mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract signed packed bytes in mm/m64 from signed packed bytes in mm and saturate results.</td>
</tr>
<tr>
<td>66 0F E8 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSUBSB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.</td>
</tr>
<tr>
<td>NP 0F E9 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSUBSW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract signed packed words in mm/m64 from signed packed words in mm and saturate results.</td>
</tr>
<tr>
<td>66 0F E9 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSUBSW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG E8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG E9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSw xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG E8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG E9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSw ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG E8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG E8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG E8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG E9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSw xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG E9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSw ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBSB (with 64-bit operands)

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToSignedByte (DEST}[7:0] - \text{SRC}[7:0]);
\]

(* Repeat subtract operation for 2nd through 7th bytes *)

\[
\text{DEST}[63:56] \leftarrow \text{SaturateToSignedByte (DEST}[63:56] - \text{SRC}[63:56]);
\]

NOTES:

PSUBSW (with 64-bit operands)

(* Repeat subtract operation for 2nd and 7th words *)

VPSUBSB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8;
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
    ELSE
      IF *merging-masking*
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking*
        DEST[i+7:i] ← 0;
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPSUBSW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
    ELSE
      IF *merging-masking*
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking*
        DEST[i+15:i] ← 0;
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPSUBSB (VEX.256 encoded version)

DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[MAXVL-1:256] ← 0;

VPSUBSB (VEX.128 encoded version)

DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[MAXVL-1:128] ← 0;

PSUBSB (128-bit Legacy SSE Version)

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[MAXVL-1:128] (Unmodified);
VPSUBSW (VEX.256 encoded version)
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[MAXVL-1:256] ← 0;

VPSUBSW (VEX.128 encoded version)
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] ← 0;

PSUBSW (128-bit Legacy SSE Version)
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0];
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] (Unmodified);

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm128i_subs_epi8(__m128i a, __m128i b);
VPSUBSB __m128i _mm128i_mask_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm128i_maskz_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_subs_epi16(__m256i a, __m256i b);
VPSUBSW __m256i _mm256_mask_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm128i_subs_epi16(__m128i a, __m128i b);
VPSUBSW __m128i _mm128i_mask_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm128i_maskz_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
PSUBSB:__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
(V)PSUBSB:__m128i _mm_subs_epi8(__m128i m1, __m128i m2)
VPSUBSB:__m256i _mm256_subs_epi8(__m256i m1, __m256i m2)
PSUBSW:__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
(V)PSUBSW:__m128i _mm_subs_epi16(__m128i m1, __m128i m2)
VPSUBSW:__m256i _mm256_subs_epi16(__m256i m1, __m256i m2)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.
## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D8 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract unsigned packed bytes in ( mm/m64 ) from unsigned packed bytes in ( mm ) and saturate result.</td>
</tr>
<tr>
<td>66 0F D8 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed unsigned byte integers in ( xmm2/m128 ) from packed unsigned byte integers in ( xmm1 ) and saturate result.</td>
</tr>
<tr>
<td>NP 0F D9 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract unsigned packed words in ( mm/m64 ) from unsigned packed words in ( mm ) and saturate result.</td>
</tr>
<tr>
<td>66 0F D9 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed unsigned word integers in ( xmm2/m128 ) from packed unsigned word integers in ( xmm1 ) and saturate result.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG D8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed unsigned byte integers in ( xmm3/m128 ) from packed unsigned byte integers in ( xmm2 ) and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F:WIG D9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed unsigned word integers in ( xmm3/m128 ) from packed unsigned word integers in ( xmm2 ) and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG D8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed unsigned byte integers in ( ymm3/m256 ) from packed unsigned byte integers in ( ymm2 ) and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F:WIG D9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed unsigned word integers in ( ymm3/m256 ) from packed unsigned word integers in ( ymm2 ) and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG D8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed unsigned byte integers in ( xmm3/m128 ) from packed unsigned byte integers in ( xmm2 ), saturate results and store in ( xmm1 ) using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSB xmm1 {k1}z, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG D8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed unsigned byte integers in ( ymm3/m256 ) from packed unsigned byte integers in ( ymm2 ), saturate results and store in ( ymm1 ) using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSB ymm1 {k1}z, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F:WIG D8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Subtract packed unsigned byte integers in ( zmm3/m512 ) from packed unsigned byte integers in ( zmm2 ), saturate results and store in ( zmm1 ) using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSB zmm1 {k1}z, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F:WIG D9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed unsigned word integers in ( xmm3/m128 ) from packed unsigned word integers in ( xmm2 ) and saturate results and store in ( xmm1 ) using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSW xmm1 {k1}z, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F:WIG D9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed unsigned word integers in ( ymm3/m256 ) from packed unsigned word integers in ( ymm2 ), saturate results and store in ( ymm1 ) using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSW ymm1 {k1}z, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INSTRUCTION SET REFERENCE, M-U

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPSUBUSBW zmm1 {k1}[z], zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

Description
Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either a MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation
PSUBUSB (with 64-bit operands)

\[
\text{DEST}[7:0] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[7:0] - \text{SRC}(7:0));
\]

(* Repeat add operation for 2nd through 7th bytes *)

\[
\text{DEST}[63:56] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[63:56] - \text{SRC}[63:56]);
\]
PSUBUSW (with 64-bit operands)

\[
\text{DEST}[15:0] \leftarrow \text{SaturateToUnsignedWord} \ (\text{DEST}[15:0] - \text{SRC}[15:0]); \\
(* \text{Repeat add operation for 2nd and 3rd words} *) \\
\text{DEST}[63:48] \leftarrow \text{SaturateToUnsignedWord} \ (\text{DEST}[63:48] - \text{SRC}[63:48]);
\]

VPSUBUSB (EVEX encoded versions)

\( (KL, VL) = (16, 128), (32, 256), (64, 512) \)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\( i \leftarrow j \times 8; \)

IF \( k1[j] \) OR \(* \text{no writemask} *\)

THEN \( \text{DEST}[i+7:i] \leftarrow \text{SaturateToUnsignedByte} \ (\text{SRC1}[i+7:i] - \text{SRC2}[i+7:i]) \)
ELSE

IF \(* \text{merging-masking} *\) \ ; \ \text{merging-masking}

THEN \(* \text{DEST}[i+7:i] \text{ remains unchanged} *\)
ELSE \(* \text{zeroing-masking} *\) \ ; \ \text{zeroing-masking}

\( \text{DEST}[i+7:i] \leftarrow 0; \)
FI

FI;
ENDFOR;
\( \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0; \)

VPSUBUSW (EVEX encoded versions)

\( (KL, VL) = (8, 128), (16, 256), (32, 512) \)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\( i \leftarrow j \times 16; \)

IF \( k1[j] \) OR \(* \text{no writemask} *\)

THEN \( \text{DEST}[i+15:i] \leftarrow \text{SaturateToUnsignedWord} \ (\text{SRC1}[i+15:i] - \text{SRC2}[i+15:i]) \)
ELSE

IF \(* \text{merging-masking} *\) \ ; \ \text{merging-masking}

THEN \(* \text{DEST}[i+15:i] \text{ remains unchanged} *\)
ELSE \(* \text{zeroing-masking} *\) \ ; \ \text{zeroing-masking}

\( \text{DEST}[i+15:i] \leftarrow 0; \)
FI

FI;
ENDFOR;
\( \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0; \)

VPSUBUSB (VEX.256 encoded version)

\( \text{DEST}[7:0] \leftarrow \text{SaturateToUnsignedByte} \ (\text{SRC1}[7:0] - \text{SRC2}[7:0]); \)
(* Repeat subtract operation for 2nd through 31st bytes *)
\( \text{DEST}[255:148] \leftarrow \text{SaturateToUnsignedByte} \ (\text{SRC1}[255:248] - \text{SRC2}[255:248]); \)
\( \text{DEST}[\text{MAXVL}-1:126] \leftarrow 0; \)

VPSUBUSB (VEX.128 encoded version)

\( \text{DEST}[7:0] \leftarrow \text{SaturateToUnsignedByte} \ (\text{SRC1}[7:0] - \text{SRC2}[7:0]); \)
(* Repeat subtract operation for 2nd through 14th bytes *)
\( \text{DEST}[127:120] \leftarrow \text{SaturateToUnsignedByte} \ (\text{SRC1}[127:120] - \text{SRC2}[127:120]); \)
\( \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \)

PSUBUSB (128-bit Legacy SSE Version)

\( \text{DEST}[7:0] \leftarrow \text{SaturateToUnsignedByte} \ (\text{DEST}[7:0] - \text{SRC}[7:0]); \)
(* Repeat subtract operation for 2nd through 14th bytes *)
\( \text{DEST}[127:120] \leftarrow \text{SaturateToUnsignedByte} \ (\text{DEST}[127:120] - \text{SRC}[127:120]); \)
\( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)
VPSUBUSW (VEX.256 encoded version)
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[MAXVL-1:256] ← 0;

VPSUBUSW (VEX.128 encoded version)
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] ← 0

PSUBUSW (128-bit Legacy SSE Version)
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents
VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8(__mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_subs_epu8(__m256i a, __m256i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8(__mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm128_subs_epu8(__m128i a, __m128i b);
VPSUBUSB __m128i _mm128_mask_subs_epu8(__mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm128_maskz_subs_epu8(__mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m64 _mm_sub_epi8(__m64 m1, __m64 m2);

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
PTEST- Logical Compare

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 17 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Set ZF if xmm2/m128 AND xmm1 result is all 0s. Set CF if xmm2/m128 AND NOT xmm1 result is all 0s.</td>
</tr>
<tr>
<td>PTEST xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F:38:W1G 17 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on bitwise AND and ANDN of sources.</td>
</tr>
<tr>
<td>VPTEST xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F:38:W1G 17 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on bitwise AND and ANDN of sources.</td>
</tr>
<tr>
<td>VPTEST ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModRM reg field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)PTEST (128-bit version)

IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;

VPTEST (VEX.256 encoded version)

IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
**Intel C/C++ Compiler Intrinsic Equivalent**

**PTEST**

```c
int _mm_testz_si128 (__m128i s1, __m128i s2);
int _mm_testc_si128 (__m128i s1, __m128i s2);
int _mm_testnzc_si128 (__m128i s1, __m128i s2);
```

**VPTEST**

```c
int _mm256_testz_si256 (__m256i s1, __m256i s2);
int _mm256_testc_si256 (__m256i s1, __m256i s2);
int _mm256_testnzc_si256 (__m256i s1, __m256i s2);
int _mm_testz_si128 (__m128i s1, __m128i s2);
int _mm_testc_si128 (__m128i s1, __m128i s2);
int _mm_testnzc_si128 (__m128i s1, __m128i s2);
```

**Flags Affected**

The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.vvv ≠ 1111B.
PTWRITE - Write Data to a Processor Trace Packet

**Description**

This instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTW packet if TriggerEn, ContextEn, FilterEn, and PTWEn are all set to 1. For more details on these values, see Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, Section 35.2.2, “Software Trace Instrumentation with PTWRITE”. The size of data is 64-bit if using REX.W in 64-bit mode, otherwise 32-bits of data are copied from the source operand.

Note: The instruction will #UD if prefix 66H is used.

**Operation**

IF (IA32_RTIT_STATUS.TriggerEn & IA32_RTIT_STATUS.ContextEn & IA32_RTIT_STATUS.FilterEn & IA32_RTIT_CTL.PTWEn) = 1
    PTW.PayloadBytes ← Encoded payload size;
    PTW.IP ← IA32_RTIT_CTL.FUPonPTW
    IF IA32_RTIT_CTL.FUPonPTW = 1
        Insert FUP packet with IP of PTWRITE;
    FI;
FI;

**Flags Affected**

None.

**Other Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
    If LOCK prefix is used.
    If 66H prefix is used.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 REX.W 0F AE /4</td>
<td>RM</td>
<td>V/N.E</td>
<td></td>
<td>Reads the data from r64/m64 to encode into a PTW packet if dependencies are met (see details below).</td>
</tr>
<tr>
<td>PTWRITE r64/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F AE /4</td>
<td>RM</td>
<td>V/V</td>
<td></td>
<td>Reads the data from r32/m32 to encode into a PTW packet if dependencies are met (see details below).</td>
</tr>
<tr>
<td>PTWRITE r32/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:rmm (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
    If LOCK prefix is used.
    If 66H prefix is used.

Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
    If LOCK prefix is used.
    If 66H prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
    current privilege level is 3.
#UD If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
    If LOCK prefix is used.
    If 66H prefix is used.
**PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 68 /r1</td>
<td>A/V/V</td>
<td>MMX</td>
<td></td>
<td>Unpack and interleave high-order bytes from (mm) and (mm/m64) into (mm).</td>
</tr>
<tr>
<td>PUNPCKHBW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 68 /r</td>
<td>A/V/V</td>
<td>SSE2</td>
<td></td>
<td>Unpack and interleave high-order bytes from (xmm1) and (xmm2/m128) into (xmm1).</td>
</tr>
<tr>
<td>PUNPCKHWD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 69 /r1</td>
<td>A/V/V</td>
<td>MMX</td>
<td></td>
<td>Unpack and interleave high-order words from (mm) and (mm/m64) into (mm).</td>
</tr>
<tr>
<td>PUNPCKHWD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 69 /r</td>
<td>A/V/V</td>
<td>SSE2</td>
<td></td>
<td>Unpack and interleave high-order words from (xmm1) and (xmm2/m128) into (xmm1).</td>
</tr>
<tr>
<td>PUNPCKHWD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 6A /r1</td>
<td>A/V/V</td>
<td>MMX</td>
<td></td>
<td>Unpack and interleave high-order doublewords from (mm) and (mm/m64) into (mm).</td>
</tr>
<tr>
<td>PUNPCKHDQ mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 6A /r</td>
<td>A/V/V</td>
<td>SSE2</td>
<td></td>
<td>Unpack and interleave high-order doublewords from (xmm1) and (xmm2/m128) into (xmm1).</td>
</tr>
<tr>
<td>PUNPCKHDQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 6D /r</td>
<td>A/V/V</td>
<td>SSE2</td>
<td></td>
<td>Unpack and interleave high-order quadwords from (xmm1) and (xmm2/m128) into (xmm1).</td>
</tr>
<tr>
<td>PUNPCKHQDQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F.WIG 68/r</td>
<td>B/V/V</td>
<td>AVX</td>
<td></td>
<td>Interleave high-order bytes from (xmm2) and (xmm3/m128) into (xmm1).</td>
</tr>
<tr>
<td>VPUNPCKHBW xmm1,xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 69/r</td>
<td>B/V/V</td>
<td>AVX</td>
<td></td>
<td>Interleave high-order words from (xmm2) and (xmm3/m128) into (xmm1).</td>
</tr>
<tr>
<td>VPUNPCKHWD xmm1,xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 6A/r</td>
<td>B/V/V</td>
<td>AVX</td>
<td></td>
<td>Interleave high-order doublewords from (xmm2) and (xmm3/m128) into (xmm1).</td>
</tr>
<tr>
<td>VPUNPCKHDQ xmm1,xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 6D/r</td>
<td>B/V/V</td>
<td>AVX</td>
<td></td>
<td>Interleave high-order quadword from (xmm2) and (xmm3/m128) into (xmm1) register.</td>
</tr>
<tr>
<td>VPUNPCKHQDQ xmm1,xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 68/r</td>
<td>B/V/V</td>
<td>AVX2</td>
<td></td>
<td>Interleave high-order bytes from (ymm2) and (ymm3/m256) into (ymm1) register.</td>
</tr>
<tr>
<td>VPUNPCKHBW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 69/r</td>
<td>B/V/V</td>
<td>AVX2</td>
<td></td>
<td>Interleave high-order words from (ymm2) and (ymm3/m256) into (ymm1) register.</td>
</tr>
<tr>
<td>VPUNPCKHWD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 6A/r</td>
<td>B/V/V</td>
<td>AVX2</td>
<td></td>
<td>Interleave high-order doublewords from (ymm2) and (ymm3/m256) into (ymm1) register.</td>
</tr>
<tr>
<td>VPUNPCKHDQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 6D/r</td>
<td>B/V/V</td>
<td>AVX2</td>
<td></td>
<td>Interleave high-order quadword from (ymm2) and (ymm3/m256) into (ymm1) register.</td>
</tr>
<tr>
<td>VPUNPCKHQDQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 68/r</td>
<td>C/V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td>Interleave high-order bytes from (xmm2) and (xmm3/m128) into (xmm1) register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHBW xmm1[k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 69/r</td>
<td>C/V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td>Interleave high-order words from (xmm2) and (xmm3/m128) into (xmm1) register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHWD xmm1[k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 6A/r</td>
<td>C/V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td>Interleave high-order doublewords from (xmm2) and (xmm3/m128) into (xmm1) register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHDQ xmm1[k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 6D/r</td>
<td>D/V/V</td>
<td>AVX512VL AVX512F</td>
<td></td>
<td>Interleave high-order quadword from (xmm2) and (xmm3/m128/m64bcst) into (xmm1) register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHQDQ xmm1[k1][z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.
When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.
EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKHBW instruction with 64-bit operands:
  \[
  \text{DEST}[7:0] \leftarrow \text{DEST}[39:32]; \\
  \text{DEST}[15:8] \leftarrow \text{SRC}[39:32]; \\
  \text{DEST}[23:16] \leftarrow \text{DEST}[47:40]; \\
  \text{DEST}[31:24] \leftarrow \text{SRC}[47:40]; \\
  \text{DEST}[39:32] \leftarrow \text{DEST}[55:48]; \\
  \text{DEST}[47:40] \leftarrow \text{SRC}[55:48]; \\
  \text{DEST}[55:48] \leftarrow \text{DEST}[63:56]; \\
  \text{DEST}[63:56] \leftarrow \text{SRC}[63:56];
  \]

PUNPCKHW instruction with 64-bit operands:
  \[
  \text{DEST}[15:0] \leftarrow \text{DEST}[47:32]; \\
  \text{DEST}[31:16] \leftarrow \text{SRC}[47:32]; \\
  \text{DEST}[47:32] \leftarrow \text{DEST}[63:48]; \\
  \text{DEST}[63:48] \leftarrow \text{SRC}[63:48];
  \]

PUNPCKHDQ instruction with 64-bit operands:
  \[
  \text{DEST}[31:0] \leftarrow \text{DEST}[63:32]; \\
  \text{DEST}[63:32] \leftarrow \text{SRC}[63:32];
  \]

INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)
\[
\text{TMP\_DEST}[255:0] \leftarrow \text{INTERLEAVE\_HIGH\_BYTES\_256b(SRC1}[255:0], \text{SRC}[255:0])
\]
\[
\text{TMP\_DEST}[511:256] \leftarrow \text{INTERLEAVE\_HIGH\_BYTES\_256b(SRC1}[511:256], \text{SRC}[511:256])
\]

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)
\[
\text{DEST}[7:0] \leftarrow \text{SRC1}[7:64]
\]
\[
\text{DEST}[15:8] \leftarrow \text{SRC2}[7:64]
\]
\[
\text{DEST}[23:16] \leftarrow \text{SRC1}[7:96]
\]
\[
\text{DEST}[31:24] \leftarrow \text{SRC2}[7:96]
\]
\[
\text{DEST}[39:32] \leftarrow \text{SRC1}[87:80]
\]
\[
\text{DEST}[47:40] \leftarrow \text{SRC2}[87:80]
\]
\[
\text{DEST}[55:48] \leftarrow \text{SRC1}[95:68]
\]
\[
\text{DEST}[63:56] \leftarrow \text{SRC2}[95:68]
\]
\[
\text{DEST}[71:64] \leftarrow \text{SRC1}[103:96]
\]
\[
\text{DEST}[79:72] \leftarrow \text{SRC2}[103:96]
\]
\[
\text{DEST}[87:80] \leftarrow \text{SRC1}[111:104]
\]
\[
\text{DEST}[95:88] \leftarrow \text{SRC2}[111:104]
\]
\[
\text{DEST}[103:96] \leftarrow \text{SRC1}[119:112]
\]
\[
\text{DEST}[111:104] \leftarrow \text{SRC2}[119:112]
\]
\[
\text{DEST}[119:112] \leftarrow \text{SRC1}[127:120]
\]
\[
\text{DEST}[127:120] \leftarrow \text{SRC2}[127:120]
\]
\[
\text{DEST}[135:128] \leftarrow \text{SRC1}[199:192]
\]
\[
\text{DEST}[143:136] \leftarrow \text{SRC2}[199:192]
\]
\[
\text{DEST}[151:144] \leftarrow \text{SRC1}[207:200]
\]
\[
\text{DEST}[159:152] \leftarrow \text{SRC2}[207:200]
\]
DEST[175:168] ← SRC2[215:208]
DEST[183:176] ← SRC1[223:216]
DEST[199:192] ← SRC1[231:224]
DEST[207:200] ← SRC2[231:224]
DEST[231:224] ← SRC1[247:240]
DEST[255:248] ← SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]
DEST[143:128] ← SRC1[207:192]
DEST[159:144] ← SRC2[207:192]
DEST[175:160] ← SRC1[223:208]
DEST[207:192] ← SRC1[239:224]
DEST[223:208] ← SRC2[239:224]
DEST[239:224] ← SRC1[255:240]
DEST[255:240] ← SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)


DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[255:224] ← SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[255:192] ← SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]

PUNPCKHBW (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHBW (VEX.128 encoded version)
DEST[127:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:127] ← 0

VPUNPCKHBW (VEX.256 encoded version)
DEST[255:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0
VPUNPCKHBW (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
   TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
   TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
   TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
FOR j ← 0 TO KL-1
   i ← j * 8
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+7:i] ← 0
   FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

PUNPCKHWD (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHWD (VEX.128 encoded version)
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ← 0

VPUNPCKHWD (VEX.256 encoded version)
DEST[255:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

VPUNPCKHWD (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
   TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
   TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
   TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask*
THEN DEST[i+15:i] \leftarrow TMP_DEST[i+15:i]
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] \leftarrow 0
    FI
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0

PUNPCKHDQ (128-bit Legacy SSE Version)
DEST[127:0] \leftarrow INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHDQ (VEX.128 encoded version)
DEST[127:0] \leftarrow INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] \leftarrow 0

VPUNPCKHDQ (VEX.256 encoded version)
DEST[255:0] \leftarrow INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] \leftarrow 0

VPUNPCKHDQ (EVEX.512 encoded version)
KL, VL = \{4, 128\}, \{8, 256\}, \{16, 512\)
FOR j \leftarrow 0 TO KL-1
    i \leftarrow j \times 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] \leftarrow SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] \leftarrow SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] \leftarrow INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] \leftarrow INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] \leftarrow INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
FOR j \leftarrow 0 TO KL-1
    i \leftarrow j \times 32
    IF k1[] OR *no writemask*
        THEN DEST[i+31:i] \leftarrow TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+31:i] \leftarrow 0
            FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

**PUNPCKHQDQ (128-bit Legacy SSE Version)**
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

**VPUNPCKHQDQ (VEX.128 encoded version)**
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

**VPUNPCKHQDQ (VEX.256 encoded version)**
DEST[255:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

**VPUNPCKHQDQ (EVEX encoded versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;

IF VL = 128
  TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fl;

IF VL = 256
  TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fl;

IF VL = 512
  TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fl;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);

VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);

VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8( __mmask64 k, __m512i a, __m512i b);

VPUNPCKHBW __m256i _mm256_unpackhi_epi8(__m256i a, __m256i b);

VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8( __mmask32 k, __m256i a, __m256i b);

VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8(v, __mmask16 k, __m256i a, __m256i b);

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data
VPUNPCKHBW __m128i __mm_maskz_unpackhi_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKHWD __m512i __mm512_unpackhi_epi16(__mm512i a, __m512i b);
VPUNPCKHWD __m512i __m512_maskz_unpackhi_epi16(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHWD __m512i __mm512_maskz_unpackhi_epi16(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHWD __m256i __mm256_maskz_unpackhi_epi16(__mm512i a, __m256i b);
VPUNPCKHWD __m256i __mm256_maskz_unpackhi_epi16(__mm512i a, __m256i b);
VPUNPCKHWD __m128i __mm_maskz_unpackhi_epi16(__m512i a, __m128i b);
VPUNPCKHWD __m128i __mm_maskz_unpackhi_epi16(__m512i a, __m128i b);
VPUNPCKHHD __m512i __mm512_unpackhi_epi32(__m512i a, __m512i b);
VPUNPCKHHD __m512i __mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHHD __m256i __mm256_maskz_unpackhi_epi32(__m512i a, __m256i b);
VPUNPCKHHD __m256i __mm256_maskz_unpackhi_epi32(__m512i a, __m256i b);
VPUNPCKHHD __m128i __mm_maskz_unpackhi_epi32(__m512i a, __m128i b);
VPUNPCKHHD __m128i __mm_maskz_unpackhi_epi32(__m512i a, __m128i b);
VPUNPCKHQDQ __m512i __mm512_unpackhi_epi64(__m512i a, __m512i b);
VPUNPCKHQDQ __m512i __mm512_maskz_unpackhi_epi64(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i __mm256_maskz_unpackhi_epi64(__m512i a, __m256i b);
VPUNPCKHQDQ __m256i __mm256_maskz_unpackhi_epi64(__m512i a, __m256i b);
VPUNPCKHQDQ __m128i __mm_maskz_unpackhi_epi64(__m512i a, __m128i b);
VPUNPCKHQDQ __m128i __mm_maskz_unpackhi_epi64(__m512i a, __m128i b);

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPUNPCKHQDQ/QDQ, see Exceptions Type E4NF.
EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb.
## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 60 /r&lt;sup&gt;1&lt;/sup&gt; PUNPCKLBW mm, mm/m32</td>
<td>A V/V</td>
<td>MMX</td>
<td>Interleave low-order bytes from mm and mm/m32 into mm.</td>
<td></td>
</tr>
<tr>
<td>66 0F 60 /r PUNPCKLBW xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>NP 0F 61 /r&lt;sup&gt;1&lt;/sup&gt; PUNPCKLWD mm, mm/m32</td>
<td>A V/V</td>
<td>MMX</td>
<td>Interleave low-order words from mm and mm/m32 into mm.</td>
<td></td>
</tr>
<tr>
<td>66 0F 61 /r PUNPCKLWD xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Interleave low-order words from xmm1 and xmm2/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>NP 0F 62 /r&lt;sup&gt;1&lt;/sup&gt; PUNPCKLDQ mm, mm/m32</td>
<td>A V/V</td>
<td>MMX</td>
<td>Interleave low-order doublewords from mm and mm/m32 into mm.</td>
<td></td>
</tr>
<tr>
<td>66 0F 62 /r PUNPCKLDQ xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 6C /r PUNPCKLQDQ xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW xmm1,xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD xmm1,xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Interleave low-order words from xmm2 and xmm3/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 60/r VPUNPCKLBW ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 61/r VPUNPCKLWD ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 62/r VPUNPCKLDQ ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Interleave low-order doublewords from ymm2 and ymm3/m256 into ymm1 register.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 6C/r VPUNPCKLDQ ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Interleave low-order quadword from ymm2 and ymm3/m256 into ymm1 register.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 60 /r VPUNPCKLBw xmm1 [k1][z], xmm2, xmm3/m128</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WIG 61 /r VPUNPCKLWd xmm1 [k1][z], xmm2, xmm3/m128</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WD 62 /r VPUNPCKLDQ xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W0 6C /r VPUNPCKLQDQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Interleave low-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register subject to write mask k1.</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Set Reference, M-U

#### PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Encoding</th>
<th>Flags</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.</td>
<td>C V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W0 62 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W0 62 /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td></td>
</tr>
</tbody>
</table>

#### Notes:


### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.
When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.
EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

**PUNPCKLBW instruction with 64-bit operands:**
- `DEST[63:56] ← SRC[31:24]`
- `DEST[47:40] ← SRC[23:16]`
- `DEST[15:8] ← SRC[7:0]`
- `DEST[7:0] ← DEST[7:0]`

**PUNPCKLWD instruction with 64-bit operands:**
- `DEST[63:48] ← SRC[31:16]`
- `DEST[31:16] ← SRC[15:0]`
- `DEST[15:0] ← DEST[15:0]`

**PUNPCKLDQ instruction with 64-bit operands:**
- `DEST[63:32] ← SRC[31:0]`
- `DEST[31:0] ← DEST[31:0]`

**INTERLEAVE_BYTES_512b (SRC1, SRC2)**
- `TMP_DEST[255:0] ← INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])`
- `TMP_DEST[511:256] ← INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])`

**INTERLEAVE_BYTES_256b (SRC1, SRC2)**
- `DEST[7:0] ← SRC1[7:0]`
- `DEST[15:8] ← SRC2[7:0]`
- `DEST[31:24] ← SRC2[15:8]`
- `DEST[47:40] ← SRC2[23:16]`
- `DEST[63:56] ← SRC2[31:24]`
- `DEST[71:64] ← SRC1[39:32]`
- `DEST[87:80] ← SRC1[47:40]`
- `DEST[95:88] ← SRC2[47:40]`
- `DEST[103:96] ← SRC1[55:48]`
- `DEST[119:112] ← SRC1[63:56]`
- `DEST[127:120] ← SRC2[63:56]`
- `DEST[151:144] ← SRC1[143:136]`
- `DEST[159:152] ← SRC2[143:136]`
DEST[175:168] ← SRC2[151:144]
DEST[183:176] ← SRC1[159:152]
DEST[191:184] ← SRC2[159:152]
DEST[207:200] ← SRC2[167:160]
DEST[215:208] ← SRC1[175:168]
DEST[223:216] ← SRC2[175:168]
DEST[231:224] ← SRC1[183:176]
DEST[239:232] ← SRC2[183:176]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC2[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[47:40] ← SRC2[23:16]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]
DEST[143:128] ← SRC1[143:128]
DEST[159:144] ← SRC2[143:128]
DEST[175:160] ← SRC1[159:144]
DEST[191:176] ← SRC2[159:144]
DEST[207:192] ← SRC1[175:160]
DEST[223:208] ← SRC2[175:160]
DEST[239:224] ← SRC1[191:176]
DEST[255:240] ← SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]

INTERLEAVE_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[255:224] ← SRC2[191:160]

INTERLEAVE_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]

PUNPCKLBW
DEST[127:0] ← INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLBW (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] ← 0

VPUNPCKLBW (VEX.256 encoded instruction)
DEST[255:0] ← INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0
**VPUNPCKLBW (EVEX.512 encoded instruction)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128

TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j * 8

IF k1[j] OR *no writemask*

THEN DEST[i+7:i] ← TMP_DEST[i+7:i]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+7:i] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

DEST[511:0] ← INTERLEAVE_BYTES_512b(SRC1, SRC2)

**PUNPCKLWD**

DEST[127:0] ← INTERLEAVE_WORDS(DEST, SRC)

DEST[255:127] (Unmodified)

**VPUNPCKLWd (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE_WORDS(SRC1, SRC2)

DEST[MAXVL-1:127] ← 0

**VPUNPCKLWd (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE_WORDS_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0

**VPUNPCKLWd (EVEX.512 encoded instruction)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j * 16

IF k1[j] OR *no writemask*
THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+15:i] remains unchanged*
  ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+15:i] ← 0
  FI
FI;
ENDIF

DEST[MAXVL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_WORDS_512b(SRC1, SRC2)

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

PUNPCKLDQ
DEST[127:0] ← INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

VPUNPCKLDQ (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPUNPCKLDQ (VEX.256 encoded instruction)
DEST[255:0] ← INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

VPUNPCKLDQ (EVEX encoded instructions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
  ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDIF;

IF VL = 128
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

IF VL = 256
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

IF VL = 512
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI
FI;
DEST511:0 ← INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] ← 0

PUNPCKLQDQ
DEST[127:0] ← INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] ← 0

VPUNPCKLQDQ (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPUNPCKLQDQ (VEX.256 encoded instruction)
DEST[255:0] ← INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

VPUNPCKLQDQ (EVEX encoded instructions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;

IF VL = 128
  TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

IF VL = 256
  TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

IF VL = 512
  TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents
VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i _mm256_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m256i __mm256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i __mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i __mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i __mm512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i __mm512_mask_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i __mm256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i __mm256_mask_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i __mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i __mm_mask_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDX __m512i __mm512_maskz_unpacklo_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDX __m512i __mm512_mask_unpacklo_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDX __m256i __mm256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDX __m256i __mm256_mask_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDX __m128i __mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDX __m128i __mm_mask_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDX __m64i __mm64_maskz_unpacklo_epi32(__mmask8 k, __m64i a, __m64i b);
VPUNPCKLQDX __m64i __mm64_maskz_unpacklo_epi32(__mmask8 k, __m64i a, __m64i b);

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPUNPCKLQDX/QD, see Exceptions Type E4NF.
EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb.
**PUSH—Push Word, Doubleword or Quadword Onto the Stack**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF /6</td>
<td>PUSH r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r/m16.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r/m32.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r/m64.</td>
</tr>
<tr>
<td>50+rw</td>
<td>PUSH r16</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r16.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r32</td>
<td>0</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r32.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r64</td>
<td>0</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r64.</td>
</tr>
<tr>
<td>68 iw</td>
<td>PUSH imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Push imm16.</td>
</tr>
<tr>
<td>68 id</td>
<td>PUSH imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Push imm32.</td>
</tr>
<tr>
<td>0E</td>
<td>PUSH CS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push CS.</td>
</tr>
<tr>
<td>16</td>
<td>PUSH SS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push SS.</td>
</tr>
<tr>
<td>1E</td>
<td>PUSH DS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push DS.</td>
</tr>
<tr>
<td>06</td>
<td>PUSH ES</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push ES.</td>
</tr>
<tr>
<td>0F A0</td>
<td>PUSH FS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push FS.</td>
</tr>
<tr>
<td>0F A8</td>
<td>PUSH GS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push GS.</td>
</tr>
</tbody>
</table>

**NOTES:**
* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRMr/m(r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd(r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Decrement the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- **Address size.** The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).
  The address size is used only when referencing a source operand in memory.

- **Operand size.** The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).
  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

  If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

- **Stack-address size.** Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.
The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

(* See Description section for possible sign-extension or zero-extension of source operand and for *)
(* a case in which the size of the memory store may be smaller than the instruction's operand size *)

IF StackAddrSize = 64
THEN
  IF OperandSize = 64
  THEN
    RSP ← RSP - 8;
    Memory[SS:RSP] ← SRC; (* push quadword *)
  ELSE IF OperandSize = 32
  THEN
    RSP ← RSP - 4;
    Memory[SS:RSP] ← SRC; (* push dword *)
  ELSE (* OperandSize = 16 *)
    RSP ← RSP - 2;
    Memory[SS:RSP] ← SRC; (* push word *)
  FI;
ELSE IF StackAddrSize = 32
THEN
  IF OperandSize = 64
  THEN
    ESP ← ESP - 8;
    Memory[SS:ESP] ← SRC; (* push quadword *)
  ELSE IF OperandSize = 32
  THEN
    ESP ← ESP - 4;
    Memory[SS:ESP] ← SRC; (* push dword *)
  ELSE (* OperandSize = 16 *)
    ESP ← ESP - 2;
    Memory[SS:ESP] ← SRC; (* push word *)
  FI;
ELSE (* StackAddrSize = 16 *)
IF OperandSize = 32
THEN
   SP ← SP – 4;
   Memory[SS:SP] ← SRC; (* push dword *)
ELSE (* OperandSize = 16 *)
   SP ← SP – 2;
   Memory[SS:SP] ← SRC; (* push word *)
FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the new value of the SP or ESP register is outside the stack segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
If the PUSH is of CS, SS, DS, or ES.
PUSHA/PUSHAD—Push All General-Purpose Registers

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>PUSHA</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push AX, CX, DX, BX, original SP, BP, SI, and DI.</td>
</tr>
<tr>
<td>60</td>
<td>PUSHAD</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the “Operation” section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

**IF 64-bit Mode**

THEN #UD

FI;

IF OperandSize = 32 (* PUSHA instruction *)

THEN

Temp ← (ESP);
Push(EAX);
Push(ECX);
Push(EDX);
Push(EBX);
Push(Temp);
Push(EBP);
Push(ESI);
Push(EDI);
ELSE (* OperandSize = 16, PUSHAD instruction *)

Temp ← (SP);
Push(AX);
Push(CX);
Push(DX);
Push(BX);
Push(Temp);
Push(BP);
Push(SI);
Push(DI);
Fi;

Flags Affected

None.

Protected Mode Exceptions

#SS(0) If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.
PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register onto the Stack

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C</td>
<td>PUSHF</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push lower 16 bits of EFLAGS.</td>
</tr>
<tr>
<td>9C</td>
<td>PUSHFD</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push EFLAGS.</td>
</tr>
<tr>
<td>9C</td>
<td>PUSHFQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push RFLAGS.</td>
</tr>
</tbody>
</table>

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFD instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction’s default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the PUSHF/PUSHFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), PUSHF (but not PUSHFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. PUSHF, however, is not affected by CR4.PVI.)

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Operation

```
IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))
(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)
THEN
  IF OperandSize = 32
    THEN
      push (EFLAGS AND 00FCFFFFH);
      (* VM and RF bits are cleared in image stored on the stack *)
    ELSE
      push (EFLAGS); (* Lower 16 bits only *)
```
PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register onto the Stack

Flags Affected
None.

Protected Mode Exceptions
#SS(0) If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the I/O privilege level is less than 3.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.
### PXOR—Logical Exclusive OR

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUIID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EF 1r</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Bitwise XOR of mm/m64 and mm.</td>
</tr>
<tr>
<td>PXOR mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F EF 1r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Bitwise XOR of xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>PXOR xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0FWig EF 1r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Bitwise XOR of xmm3/m128 and xmm2.</td>
</tr>
<tr>
<td>VPXOR xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0FWig EF 1r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Bitwise XOR of ymm3/m256 and ymm2.</td>
</tr>
<tr>
<td>VPXOR ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0FW0 EF 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.</td>
</tr>
<tr>
<td>VPXORD xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0FW0 EF 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.</td>
</tr>
<tr>
<td>VPXORD ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0FW0 EF 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>VPXORD zmm1 [k1][z], zmm2, zmm3/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0FW1 EF 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.</td>
</tr>
<tr>
<td>VPXORQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0FW1 EF 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.</td>
</tr>
<tr>
<td>VPXORQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0FW1 EF 1r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>VPXORQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMrm (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMrm (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.
128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**Operation**

**PXOR (64-bit operand)**

DEST ← DEST XOR SRC

**PXOR (128-bit Legacy SSE version)**

DEST ← DEST XOR SRC
DEST[MAXVL-1:128] (Unmodified)

**VPXOR (VEX.128 encoded version)**

DEST ← SRC1 XOR SRC2
DEST[MAXVL-1:128] ← 0

**VPXOR (VEX.256 encoded version)**

DEST ← SRC1 XOR SRC2
DEST[MAXVL-1:256] ← 0

**VPXORD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0]
      ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
        DEST[31:0] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
VPXORQ (EVEX encoded versions)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 64\)
  IF \(k1[j]\) OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN \(\text{DEST}[i+63:i] \leftarrow \text{SRC}1[i+63:i]\)
      ELSE \(\text{DEST}[i+63:i] \leftarrow \text{SRC}1[i+63:i]\)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN \(\text{DEST}[63:0]\) remains unchanged*
    ELSE ; zeroing-masking
      \(\text{DEST}[63:0] \leftarrow 0\)
    FI;
  FI;
ENDFOR;
\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\begin{align*}
\text{VPXORD} \ & \ _{\text{m512i}} \ _{\text{mm512}_x}{\text{or}}{\text{epi32}}(_{\text{m512i}} \ a, _{\text{m512i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m512i}} \ _{\text{mm512}_x}{\text{or}}{\text{epi32}}(_{\text{m512i}} \ s, _{\text{mmask16}} \ m, _{\text{m512i}} \ a, _{\text{m512i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m512i}} \ _{\text{mm512}_x}{\text{or}}{\text{epi32}}(_{\text{m512i}} \ s, _{\text{mmask16}} \ m, _{\text{m512i}} \ a, _{\text{m512i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi32}}(_{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi32}}(_{\text{m256i}} \ s, _{\text{mmask8}} \ m, _{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi32}}(_{\text{m256i}} \ s, _{\text{mmask8}} \ m, _{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi32}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi32}}(_{\text{m128i}} \ s, _{\text{mmask8}} \ m, _{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi32}}(_{\text{m128i}} \ s, _{\text{mmask8}} \ m, _{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi32}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi32}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi32}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi64}}(_{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi64}}(_{\text{m256i}} \ s, _{\text{mmask8}} \ m, _{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi64}}(_{\text{m256i}} \ s, _{\text{mmask8}} \ m, _{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi64}}(_{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{epi64}}(_{\text{m256i}} \ a, _{\text{m256i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi64}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi64}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi64}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXORD} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{epi64}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{PXOR} \ & \ _{\text{m64}} \ _{\text{mm64}_x}{\text{or}}{\text{si64}}(_{\text{m64}} \ m1, _{\text{m64}} m2) \\
\text{VPXOR} \ & \ _{\text{m128i}} \ _{\text{mm128}_x}{\text{or}}{\text{si128}}(_{\text{m128i}} \ a, _{\text{m128i}} \ b) \\
\text{VPXOR} \ & \ _{\text{m256i}} \ _{\text{mm256}_x}{\text{or}}{\text{si256}}(_{\text{m256i}} \ a, _{\text{m256i}} \ b)
\end{align*}
\]

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
# RCL/RCR/ROL/ROR—Rotate

<table>
<thead>
<tr>
<th>Opcode**</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/O Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 /2</td>
<td>RCL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) left once.</td>
</tr>
<tr>
<td>REX + D0 /2</td>
<td>RCL r/m8*, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) left once.</td>
</tr>
<tr>
<td>D2 /2</td>
<td>RCL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) left CL times.</td>
</tr>
<tr>
<td>REX + D2 /2</td>
<td>RCL r/m8*, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) left CL times.</td>
</tr>
<tr>
<td>C0 /2 ib</td>
<td>RCL r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) left imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /2 ib</td>
<td>RCL r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) left imm8 times.</td>
</tr>
<tr>
<td>D1 /2</td>
<td>RCL r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) left once.</td>
</tr>
<tr>
<td>D3 /2</td>
<td>RCL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) left CL times.</td>
</tr>
<tr>
<td>C1 /2 ib</td>
<td>RCL r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) left imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /2 ib</td>
<td>RCL r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) left imm8 times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D0 /3</td>
<td>RCR r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) right once.</td>
</tr>
<tr>
<td>REX + D0 /3</td>
<td>RCR r/m8*, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) right once.</td>
</tr>
<tr>
<td>D2 /3</td>
<td>RCR r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) right CL times.</td>
</tr>
<tr>
<td>REX + D2 /3</td>
<td>RCR r/m8*, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) right CL times.</td>
</tr>
<tr>
<td>C0 /3 ib</td>
<td>RCR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) right imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /3 ib</td>
<td>RCR r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) right imm8 times.</td>
</tr>
<tr>
<td>D1 /3</td>
<td>RCR r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) right once.</td>
</tr>
<tr>
<td>D3 /3</td>
<td>RCR r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) right CL times.</td>
</tr>
<tr>
<td>C1 /3 ib</td>
<td>RCR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) right imm8 times.</td>
</tr>
<tr>
<td>D1 /3</td>
<td>RCR r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) right once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>REX.W + D1 /3</td>
<td>RCR r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) right once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D3 /3</td>
<td>RCR r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) right CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /3</td>
<td>RCR r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) right CL times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>C1 /3 ib</td>
<td>RCR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) right imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /3 ib</td>
<td>RCR r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) right imm8 times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D0 /0</td>
<td>ROL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 left once.</td>
</tr>
<tr>
<td>REX + D0 /0</td>
<td>ROL r/m8*, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 left once.</td>
</tr>
<tr>
<td>D2 /0</td>
<td>ROL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 left CL times.</td>
</tr>
<tr>
<td>REX + D2 /0</td>
<td>ROL r/m8*, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 left CL times.</td>
</tr>
<tr>
<td>C0 /0 ib</td>
<td>ROL r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 left imm8 times.</td>
</tr>
<tr>
<td>Opcode**</td>
<td>Instruction</td>
<td>Op/En</td>
<td>64-Bit Mode</td>
<td>Compat/ Leg Mode</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------</td>
<td>-------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>REX + C0 /0 ib</td>
<td>ROL r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 left imm8 times.</td>
</tr>
<tr>
<td>D1 /0</td>
<td>ROL r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 left once.</td>
</tr>
<tr>
<td>D3 /0</td>
<td>ROL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 left CL times.</td>
</tr>
<tr>
<td>C1 /0 ib</td>
<td>ROL r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 left imm8 times.</td>
</tr>
<tr>
<td>D1 /0</td>
<td>ROL r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 left once.</td>
</tr>
<tr>
<td>REX.W + D1 /0</td>
<td>ROL r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 left once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D3 /0</td>
<td>ROL r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 left CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /0</td>
<td>ROL r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 left CL times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>C1 /0 ib</td>
<td>ROL r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 left imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /0 ib</td>
<td>ROL r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 left imm8 times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D0 /1</td>
<td>ROR r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 right once.</td>
</tr>
<tr>
<td>REX + D0 /1</td>
<td>ROR r/m8*, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 right once.</td>
</tr>
<tr>
<td>D2 /1</td>
<td>ROR r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 right CL times.</td>
</tr>
<tr>
<td>REX + D2 /1</td>
<td>ROR r/m8*, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 right CL times.</td>
</tr>
<tr>
<td>C0 /1 ib</td>
<td>ROR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m16 right imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /1 ib</td>
<td>ROR r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m16 right imm8 times.</td>
</tr>
<tr>
<td>D1 /1</td>
<td>ROR r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 right once.</td>
</tr>
<tr>
<td>D3 /1</td>
<td>ROR r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 right CL times.</td>
</tr>
<tr>
<td>C1 /1 ib</td>
<td>ROR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 right imm8 times.</td>
</tr>
<tr>
<td>D1 /1</td>
<td>ROR r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 right once.</td>
</tr>
<tr>
<td>REX.W + D1 /1</td>
<td>ROR r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 right once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D3 /1</td>
<td>ROR r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 right CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /1</td>
<td>ROR r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 right CL times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>C1 /1 ib</td>
<td>ROR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 right imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /1 ib</td>
<td>ROR r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 right imm8 times. Uses a 6 bit count.</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
** See IA-32 Architecture Compatibility section below.

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>M1</td>
</tr>
<tr>
<td>MC</td>
</tr>
<tr>
<td>MI</td>
</tr>
</tbody>
</table>
Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

(* RCL and RCR instructions *)
SIZE ← OperandSize;
CASE (determine count) OF
  SIZE ← 8: tempCOUNT ← (COUNT AND 1FH) MOD 9;
  SIZE ← 16: tempCOUNT ← (COUNT AND 1FH) MOD 17;
  SIZE ← 32: tempCOUNT ← COUNT AND 1FH;
  SIZE ← 64: tempCOUNT ← COUNT AND 3FH;
ESAC;

(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← MSB(DEST);
    DEST ← (DEST * 2) + CF;
    CF ← tempCF;
    tempCOUNT ← tempCOUNT - 1;
  OD;
ELSE;
IF (COUNT & COUNTMASK) = 1
  THEN OF ← MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
RCL/ROR/ROL/ROR—Rotate

(* RCR instruction operation *)
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (CF * 2\(^{\text{SIZE}}\));
        CF ← tempCF;
        tempCOUNT ← tempCOUNT - 1;
    OD;

(* ROL and ROR instructions *)
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* ROL instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← MSB(DEST);
        DEST ← (DEST * 2) + tempCF;
        tempCOUNT ← tempCOUNT - 1;
    OD;
ELIHw;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF ← LSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* ROR instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (tempCF * 2\(^{\text{SIZE}}\));
        tempCOUNT ← tempCOUNT - 1;
    OD;
ELIHw;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF ← MSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR MSB(DEST) - 1;
    ELSE OF is undefined;
FI;
**Flags Affected**
If the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined. The CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

**Protected Mode Exceptions**
- #GP(0) If the source operand is located in a non-writable segment.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If the DS, ES, FS, or GS register contains a NULL segment selector.
- #AC(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**
- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**
- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the source operand is located in a non-writable segment.
- #PF(fault-code) If the memory address is in a non-canonical form.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 53 /r RCPPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128 and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 53 /r VRCPPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of packed single-precision values in xmm2/mem and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 53 /r VRCPPS ymm1, ymm2/m256</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of packed single-precision values in ymm2/mem and stores the results in ymm1.</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 \times 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111101000000000B\times2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.0000000000110000000B\times2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8–XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
**Operation**

**RCPPS (128-bit Legacy SSE version)**
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

**VRCPSS (VEX.128 encoded version)**
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] ← 0

**VRCPSS (VEX.256 encoded version)**
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[159:128] ← APPROXIMATE(1/SRC[159:128])
DEST[191:160] ← APPROXIMATE(1/SRC[191:160])
DEST[223:192] ← APPROXIMATE(1/SRC[223:192])
DEST[255:224] ← APPROXIMATE(1/SRC[255:224])

**Intel C/C++ Compiler Intrinsic Equivalent**
RCCPS: __m128 _mm_rcp_ps(__m128 a)
RCPPS: __m256 _mm256_rcp_ps (__m256 a);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Exceptions Type 4; additionally
#UD If VEX.vvvv ≠ 1111B.
RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode* Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 53 r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm2/m32 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VRCPSS xmm1, xmm2, xmm3/m32</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the result in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

Description

Computes of an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

\[ |\text{Relative Error}| \leq 1.5 \times 2^{-12} \]

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \( \infty \) of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign).

Tiny results (see Section 4.9.1.5, “Numeric Underflow Exception (#U)” in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to \( 1.11111111011000000000000B \times 2^{125} \) are guaranteed to not produce tiny results; input values less than or equal to \( 1.00000000001100000000001B \times 2^{126} \) are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

**RCPSS (128-bit Legacy SSE version)**

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])

DEST[MAXVL-1:32] (Unmodified)
VRCPSS (VEX.128 encoded version)
DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
RCPSS: __m128 _mm_rcp_ss(__m128 a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 5.
RDFSBASE/RDGSBASE—Read FS/GS Segment Base

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F AE /0 RDFSBASE r32</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 32-bit destination register with the FS base address.</td>
</tr>
<tr>
<td>F3 REX.W OF AE /0 RDFSBASE r64</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 64-bit destination register with the FS base address.</td>
</tr>
<tr>
<td>F3 0F AE /1 RDGSBASE r32</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 32-bit destination register with the GS base address.</td>
</tr>
<tr>
<td>F3 REX.W OF AE /1 RDGSBASE r64</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 64-bit destination register with the GS base address.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Loads the general-purpose register indicated by the modR/M:r/m field with the FS or GS segment base address.

The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

**Operation**

DEST ← FS/GS segment base address;

**Flags Affected**

None

**C/C++ Compiler Intrinsic Equivalent**

RDFSBASE: unsigned int _readfsbase_u32(void);
RDFSBASE: unsigned __int64 _readfsbase_u64(void);
RDGSBASE: unsigned int _readgsbase_u32(void);
RDGSBASE: unsigned __int64 _readgsbase_u64(void);

**Protected Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

**Real-Address Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in compatibility mode.
64-Bit Mode Exceptions

#UD  If the LOCK prefix is used.
    If CR4.FSGSBASE[bit 16] = 0.
    If CPUID.07H:0H:EBX.FSGSBASE[bit 0] = 0.
RDMSR—Read from Model Specific Register

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF 32</td>
<td>RDMSR</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Read MSR specified by ECX into EDX:EAX.</td>
</tr>
</tbody>
</table>

**NOTES:**
* See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 2, “Model-Specific Registers (MSRs)” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4,* lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

**IA-32 Architecture Compatibility**

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

See "Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C,* for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

EDX:EAX ← MSR[ECX];

**Flags Affected**

None.

**Protected Mode Exceptions**

- **#GP(0)** If the current privilege level is not 0.
  - If the value in ECX specifies a reserved or unimplemented MSR address.
- **#UD** If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP If the value in ECX specifies a reserved or unimplemented MSR address.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) The RDMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
RDPI D—Read Processor ID

<table>
<thead>
<tr>
<th>Opcode/Description</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF C7 /7 RDPID r32</td>
<td>R</td>
<td>N.E./V</td>
<td>RDPID</td>
<td>Read IA32_TSC_AUX into r32.</td>
</tr>
<tr>
<td>F3 OF C7 /7 RDPID r64</td>
<td>R</td>
<td>V/N.E.</td>
<td>RDPID</td>
<td>Read IA32_TSC_AUX into r64.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>ModRM r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the destination register. The value of CS.D and operand-size prefixes (66H and REX.W) do not affect the behavior of the RDPID instruction.

**Operation**

DEST ← IA32_TSC_AUX

**Flags Affected**

None.

**Protected Mode Exceptions**

#UD If the LOCK prefix is used.
If CPUID.7H:0:ECX.RDPID[bit 22] = 0.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

---

1. ModRM.MOD = 011B required
RDPKRU—Read Protection Key Rights for User Pages

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 EE</td>
<td>RDPKRU</td>
<td>ZO</td>
<td>V/V</td>
<td>OSPKE</td>
<td>Reads PKRU into EAX.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Reads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX are ignored and the high-order 32-bits of RDX and RAX are cleared.

Operation

IF (ECX = 0)
THEN
    EAX ← PKRU;
    EDX ← 0;
ELSE #GP(0);
FI;

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDPKRU: uint32_t _rdpkru_u32(void);

Protected Mode Exceptions

#GP(0)  If ECX ≠ 0.
#UD    If the LOCK prefix is used.
        If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
RDPMC—Read Performance-Monitoring Counters

 Opcode*  Instruction  Op/En  64-Bit Mode  Compat/Leg Mode  Description
0F 33  RDPMC  ZO  Valid  Valid  Read performance-monitoring counter specified by ECX into EDX:EAX.

Table 4-16. Valid General and Special Purpose Performance Counter Index Range for RDPMC

<table>
<thead>
<tr>
<th>Processor Family</th>
<th>DisplayFamily_DisplayModel/Other Signatures</th>
<th>Valid PMC Index Range</th>
<th>General-purpose Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>P6</td>
<td>06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>Processors Based on Intel NetBurst microarchitecture (No L3)</td>
<td>0FH_00H, 0FH_01H, 0FH_02H, 0FH_03H, 0FH_04H, 0FH_06H</td>
<td>≥ 0 and ≤ 17</td>
<td>≥ 0 and ≤ 17</td>
</tr>
<tr>
<td>Pentium M processors</td>
<td>06H_09H, 06H_0DH</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>Processors Based on Intel NetBurst microarchitecture (No L3)</td>
<td>0FH_03H, 0FH_04H and (L3 is present)</td>
<td>≥ 0 and ≤ 25</td>
<td>≥ 0 and ≤ 17</td>
</tr>
</tbody>
</table>

Description
The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do not support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. See below for the treatment of the EDX register for “fast” reads.

The ECX register specifies the counter type (if the processor supports architectural performance monitoring) and counter index. Counter type is specified in ECX[30] to select one of two type of performance counters. If the processor does not support architectural performance monitoring, ECX[30:0] specifies the counter index; otherwise ECX[29:0] specifies the index relative to the base of each counter type. ECX[31] selects “fast” read mode if supported. The two counter types are:

- General-purpose or special-purpose performance counters are specified with ECX[30] = 0: The number of general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 18, “Performance Monitoring” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B. Special-purpose counters are available only in selected processor members, see Table 4-16.


The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters. The width of special-purpose performance counters are implementation specific.

Table 4-16 lists valid indices of the general-purpose and special-purpose performance counters according to the DisplayFamily_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).
Processors based on Intel NetBurst microarchitecture support "fast" (32-bit) and "slow" (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on these processors than a full 40-bit read.

On processors based on Intel NetBurst microarchitecture with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDMPC with ECX between from 4000_0000H and 4000_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Chapter 19, "Performance Monitoring Events," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If
an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

**Operation**

(* Intel processors that support architectural performance monitoring *)

Most significant counter bit (MSCB) = 47

IF (CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0)
THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
FI;

(* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series*)

Most significant counter bit (MSCB) = 39

IF (CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0)
THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
FI;

(* P6 family processors and Pentium processor with MMX technology *)

IF (ECX = 0 or 1) and (CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0)
THEN
    EAX ← PMC(ECX)[31:0];
    EDX ← PMC(ECX)[39:32];
ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
FI;

(* Processors based on Intel NetBurst microarchitecture *)

IF (CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0)
THEN IF (ECX[30:0] = 0:17)
    THEN IF ECX[31] = 0
THEN
   EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
   EDX ← PMC(ECX[30:0])[39:32];
ELSE (* ECX[31] = 1 *)
   THEN
   EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
   EDX ← 0;
   FI;
ELSE IF (* 64-bit Intel processor based on Intel NetBurst microarchitecture with L3 *)
   THEN IF (ECX[30:0] = 18:25 )
   EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
   EDX ← 0;
   FI;
   ELSE (* Invalid PMC index in ECX[30:0], see Table 4-19. *)
     GP(0);
   FI;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
   #GP(0);
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
   If an invalid performance counter index is specified (see Table 4-16).
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If an invalid performance counter index is specified (see Table 4-16).
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the PCE flag in the CR4 register is clear.
   If an invalid performance counter index is specified (see Table 4-16).
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
   If an invalid performance counter index is specified (see Table 4-16).
#UD If the LOCK prefix is used.
RDRAND—Read Random Number

<table>
<thead>
<tr>
<th>Opcode/* Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C7 /6 RDRAND r16</td>
<td>M</td>
<td>V/V</td>
<td>RDRAND</td>
<td>Read a 16-bit random number and store in the destination register.</td>
</tr>
<tr>
<td>0F C7 /6 RDRAND r32</td>
<td>M</td>
<td>V/V</td>
<td>RDRAND</td>
<td>Read a 32-bit random number and store in the destination register.</td>
</tr>
<tr>
<td>REX.W + 0F C7 /6 RDRAND r64</td>
<td>M</td>
<td>V/I</td>
<td>RDRAND</td>
<td>Read a 64-bit random number and store in the destination register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRMr/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, Section 7.3.17, “Random Number Generator Instructions”).

This instruction is available at all privilege levels.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF Hw_RND_GEN.ready = 1
  THEN
    CASE of
      osize is 64: DEST[63:0] ← Hw_RND_GEN.data;
      osize is 32: DEST[31:0] ← Hw_RND_GEN.data;
      osize is 16: DEST[15:0] ← Hw_RND_GEN.data;
    ESAC
    CF ← 1;
  ELSE
    CASE of
      osize is 64: DEST[63:0] ← 0;
      osize is 32: DEST[31:0] ← 0;
      osize is 16: DEST[15:0] ← 0;
    ESAC
    CF ← 0;
  FI
OF, SF, ZF, AF, PF ← 0;

Flags Affected

The CF flag is set according to the result (see the “Operation” section above). The OF, SF, ZF, AF, and PF flags are set to 0.
Intel C/C++ Compiler Intrinsic Equivalent

RDRAND: int _rdrand16_step( unsigned short * );
RDRAND: int _rdrand32_step( unsigned int * );
RDRAND: int _rdrand64_step( unsigned __int64 * );

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.01H:ECX.RDRAND[bit 30] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
RDSEED—Read Random SEED

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF C7 /7 RDSEED r16</td>
<td>M</td>
<td>V/V</td>
<td>RDSEED</td>
<td>Read a 16-bit NIST SP800-90B &amp; C compliant random value and store in the destination register.</td>
</tr>
<tr>
<td>OF C7 /7 RDSEED r32</td>
<td>M</td>
<td>V/V</td>
<td>RDSEED</td>
<td>Read a 32-bit NIST SP800-90B &amp; C compliant random value and store in the destination register.</td>
</tr>
<tr>
<td>REX.W + OF C7 /7 RDSEED r64</td>
<td>M</td>
<td>V/I</td>
<td>RDSEED</td>
<td>Read a 64-bit NIST SP800-90B &amp; C compliant random value and store in the destination register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF HW_NRND_GEN.ready = 1
    THEN
        CASE of
            osize is 64: DEST[63:0] ← HW_NRND_GEN.data;
            osize is 32: DEST[31:0] ← HW_NRND_GEN.data;
            osize is 16: DEST[15:0] ← HW_NRND_GEN.data;
        ESAC;
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC;
        CF ← 0;
    FI;
    OF, SF, ZF, AF, PF ← 0;
Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

C/C++ Compiler Intrinsic Equivalent

RDSEED int _rdseed16_step( unsigned short * );
RDSEED int _rdseed32_step( unsigned int * );
RDSEED int _rdseed64_step( unsigned __int64 * );

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If the F2H or F3H prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.
RDTSC—Read Time-Stamp Counter

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See ‘’Time Stamp Counter’’ in Chapter 17 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSC:

- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads are globally visible,\(^1\) it can execute LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads and stores are globally visible, it can execute the sequence MFENCE;LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute the sequence LFENCE immediately after RDTSC.

This instruction was introduced by the Pentium processor.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

\[
\text{IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)} \\
\quad \text{THEN EDX:EAX ← TimeStampCounter;} \\
\quad \text{ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)} \\
\quad \quad \text{#GP(0);} \\
\text{FI;}
\]

Flags Affected

None.

---

\(^1\) A load is considered to become globally visible when the value to be loaded is determined.
Protected Mode Exceptions
#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the TSD flag in register CR4 is set.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
RDTSCP—Read Time-Stamp Counter and Processor ID

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 F9</td>
<td>RDTSCP</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Read 64-bit time-stamp counter and IA32_TSC_AUX value into EDX:EAX and ECX.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Reads the current value of the processor’s time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32_TSC_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See “Time Stamp Counter” in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction is not a serializing instruction, but it does wait until all previous instructions have executed and all previous loads are globally visible. But it does not wait for previous stores to be globally visible, and subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSCP:

- If software requires RDTSCP to be executed only after all previous stores are globally visible, it can execute MFENCE immediately before RDTSCP.
- If software requires RDTSCP to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute LFENCE immediately after RDTSCP.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN
        EDX:EAX ← TimeStampCounter;
        ECX ← IA32_TSC_AUX[31:0];
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

### Flags Affected

None.

---

1. A load is considered to become globally visible when the value to be loaded is determined.
Protected Mode Exceptions
#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD If the LOCK prefix is used.
If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Virtual-8086 Mode Exceptions
#GP(0) If the TSD flag in register CR4 is set.
#UD If the LOCK prefix is used.
If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 6C</td>
<td>REP INS m8, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input (E)CX bytes from port DX into ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 6C</td>
<td>REP INS m8, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Input RCX bytes from port DX into [RDI].</td>
</tr>
<tr>
<td>F3 6D</td>
<td>REP INS m16, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input (E)CX words from port DX into ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 6D</td>
<td>REP INS m32, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input (E)CX doublewords from port DX into ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 6D</td>
<td>REP INS r/m32, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Input RCX default size from port DX into [RDI].</td>
</tr>
<tr>
<td>F3 A4</td>
<td>REP MOVS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 REX.W A4</td>
<td>REP MOVS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move RCX bytes from [RSI] to [RDI].</td>
</tr>
<tr>
<td>F3 A5</td>
<td>REP MOVS m16, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 A5</td>
<td>REP MOVS m32, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 REX.W A5</td>
<td>REP MOVS m64, m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move RCX quadwords from [RSI] to [RDI].</td>
</tr>
<tr>
<td>F3 6E</td>
<td>REP OUTS DX, r/m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output (E)CX bytes from DS:[(E)SI] to port DX.</td>
</tr>
<tr>
<td>F3 REX.W 6E</td>
<td>REP OUTS DX, r/m8*</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Output RCX bytes from [RSI] to port DX.</td>
</tr>
<tr>
<td>F3 6F</td>
<td>REP OUTS DX, r/m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output (E)CX words from DS:[(E)SI] to port DX.</td>
</tr>
<tr>
<td>F3 6F</td>
<td>REP OUTS DX, r/m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output (E)CX doublewords from DS:[(E)SI] to port DX.</td>
</tr>
<tr>
<td>F3 REX.W 6F</td>
<td>REP OUTS DX, r/m32</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Output RCX default size from [RSI] to port DX.</td>
</tr>
<tr>
<td>F3 AC</td>
<td>REP LODS AL</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Load (E)CX bytes from DS:[(E)SI] to AL.</td>
</tr>
<tr>
<td>F3 REX.W AC</td>
<td>REP LODS AL</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load RCX bytes from [RSI] to AL.</td>
</tr>
<tr>
<td>F3 AD</td>
<td>REP LODS AX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Load (E)CX words from DS:[(E)SI] to AX.</td>
</tr>
<tr>
<td>F3 AD</td>
<td>REP LODS EAX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Load (E)CX doublewords from DS:[(E)SI] to EAX.</td>
</tr>
<tr>
<td>F3 REX.W AD</td>
<td>REP LODS RAX</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load RCX quadwords from [RSI] to RAX.</td>
</tr>
<tr>
<td>F3 AA</td>
<td>REP STOS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fill (E)CX bytes at ES:[(E)DI] with AL.</td>
</tr>
<tr>
<td>F3 REX.W AA</td>
<td>REP STOS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Fill RCX bytes at [RDI] with AL.</td>
</tr>
<tr>
<td>F3 AB</td>
<td>REP STOS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fill (E)CX words at ES:[(E)DI] with AX.</td>
</tr>
<tr>
<td>F3 AB</td>
<td>REP STOS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fill (E)CX doublewords at ES:[(E)DI] with EAX.</td>
</tr>
<tr>
<td>F3 REX.W AB</td>
<td>REP STOS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Fill RCX quadwords at [RDI] with RAX.</td>
</tr>
<tr>
<td>F3 A6</td>
<td>REPE CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F3 REX.W A6</td>
<td>REPE CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find non-matching bytes in [RDI] and [RSI].</td>
</tr>
<tr>
<td>F3 A7</td>
<td>REPE CMPS m16, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F3 A7</td>
<td>REPE CMPS m32, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F3 REX.W A7</td>
<td>REPE CMPS m64, m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find non-matching quadwords in [RDI] and [RSI].</td>
</tr>
<tr>
<td>F3 AE</td>
<td>REPE SCAS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find non-AL byte starting at ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 REX.W AE</td>
<td>REPE SCAS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find non-AL byte starting at [RDI].</td>
</tr>
<tr>
<td>F3 AF</td>
<td>REPE SCAS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find non-AX word starting at ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 AF</td>
<td>REPE SCAS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find non-EAX doubleword starting at ES:[(E)DI].</td>
</tr>
</tbody>
</table>
Rep/Repz/Repne/Repnz—Repeat String Operation Prefix

Description
Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPZ (repeat while zero), and REPNE (repeat while not equal) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPZ and REPNE prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNE prefixes are synonymous forms of the REP prefix, respectively.) The F3H prefix is defined for the following instructions and undefined for the rest:

- F3H as REP/REPZ for string and input/output instruction.
- F3H is a mandatory prefix for POPCNT, LZCNT, and ADOX.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-17.

### Table 4-17. Repeat Prefixes

<table>
<thead>
<tr>
<th>Repeat Prefix</th>
<th>Termination Condition 1</th>
<th>Termination Condition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>RCX or (E)CX = 0</td>
<td>None</td>
</tr>
<tr>
<td>REP/REPZ</td>
<td>RCX or (E)CX = 0</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>REPNE/REPNZ</td>
<td>RCX or (E)CX = 0</td>
<td>ZF = 1</td>
</tr>
</tbody>
</table>

NOTES:
* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

REP INS may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

**Operation**

IF AddressSize = 16
THEN
   Use CX for CountReg;
   Implicit Source/Dest operand for memory use of SI/DI;
ELSE IF AddressSize = 64
THEN Use RCX for CountReg;
   Implicit Source/Dest operand for memory use of RSI/RDI;
ELSE
   Use ECX for CountReg;
   Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
   DO
      Service pending interrupts (if any);
      Execute associated string instruction;
      CountReg ← (CountReg - 1);
      IF CountReg = 0
      THEN exit WHILE loop; FI;
      IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; FI;
   OD;

**Flags Affected**

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.
Exceptions (All Operating Modes)
Exceptions may be generated by an instruction associated with the prefix.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
RET—Return from Procedure

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>RET</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Near return to calling procedure.</td>
</tr>
<tr>
<td>CB</td>
<td>RET</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Far return to calling procedure.</td>
</tr>
<tr>
<td>C2 iw</td>
<td>RET imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Near return to calling procedure and pop imm16 bytes from stack.</td>
</tr>
<tr>
<td>CA iw</td>
<td>RET imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Far return to calling procedure and pop imm16 bytes from stack.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>imm16</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure’s stack and the calling procedure’s stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.
Operation

(* Near return *)
IF instruction = near return
THEN;
  IF OperandSize = 32
  THEN
    IF top 4 bytes of stack not within stack limits
      THEN #SS(0); FI;
    EIP ← Pop();
  ELSE
    IF OperandSize = 64
    THEN
      IF top 8 bytes of stack not within stack limits
        THEN #SS(0); FI;
      RIP ← Pop();
      ELSE (* OperandSize = 16 *)
        IF top 2 bytes of stack not within stack limits
          THEN #SS(0); FI;
        tempEIP ← Pop();
        tempEIP ← tempEIP AND 0000FFFFH;
        IF tempEIP not within code segment limits
          THEN #GP(0); FI;
        EIP ← tempEIP;
      FI;
    FI;
  IF instruction has immediate operand
  THEN (* Release parameters from stack *)
    IF StackAddressSize = 32
    THEN
      ESP ← ESP + SRC;
    ELSE
      IF StackAddressSize = 64
      THEN
        RSP ← RSP + SRC;
      ELSE (* StackAddressSize = 16 *)
        SP ← SP + SRC;
      FI;
    FI;
  FI;
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
THEN
  IF OperandSize = 32
  THEN
    IF top 8 bytes of stack not within stack limits
      THEN #SS(0); FI;
    EIP ← Pop();
    CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
  ELSE (* OperandSize = 16 *)
    IF top 4 bytes of stack not within stack limits
      THEN #SS(0); FI;
tempEIP ← Pop();
tempEIP ← tempEIP AND 0000FFFFH;
IF tempEIP not within code segment limits
THEN #GP(0); Fl;
EIP ← tempEIP;
CS ← Pop(); (* 16-bit pop *)
FI;
IF instruction has immediate operand
THEN (* Release parameters from stack *)
SP ← SP + (SRC AND FFFFH);
FI;
FI;
(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
THEN
IF OperandSize = 32
THEN
IF second doubleword on stack is not within stack limits
THEN #SS(0); Fl;
ELSE (* OperandSize = 16 *)
IF second word on stack is not within stack limits
THEN #SS(0); Fl;
FI;
IF return code segment selector is NULL
THEN #GP(0); Fl;
IF return code segment selector addresses descriptor beyond descriptor table limit
THEN #GP(selector); Fl;
Obtain descriptor to which return code segment selector points from descriptor table;
IF return code segment descriptor is not a code segment
THEN #GP(selector); Fl;
IF return code segment selector RPL < CPL
THEN #GP(selector); Fl;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
THEN #GP(selector); Fl;
IF return code segment descriptor is non-conforming and return code
segment DPL ≠ return code segment selector RPL
THEN #GP(selector); Fl;
IF return code segment descriptor is not present
THEN #NP(selector); Fl;
IF return code segment selector RPL > CPL
THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
FI;
FI;
RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
THEN #GP(0); Fl;
IF OperandSize = 32
THEN
EIP ← Pop();
CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
ELSE (* OperandSize = 16 *)
    EIP ← Pop();
    EIP ← EIP AND 0000FFFFH;
    CS ← Pop(); (* 16-bit pop *)
Fl;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
            FI;
        FI;
RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector
        THEN #GP(selector); FI;
    IF stack segment not present
        THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    CPL ← ReturnCodeSegmentSelector(RPL);
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
            CS(RPL)← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                            ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                        FI;
                    FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
            ELSE (* OperandSize = 16 *)
                EIP ← Pop();
    ELSE (* OperandSize = 16 *)
        EIP ← Pop();
        EIP ← EIP AND 0000FFFFH;
        CS ← Pop(); (* 16-bit pop *)
        Fl;
EIP ← EIP AND 0000FFFFH;
CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
CS(RPL) ← CPL;
IF instruction has immediate operand
    THEN (* Release parameters from called procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP ← SP + SRC;
            FI;
        FI;
    FI;
    tempESP ← Pop();
    tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
    ESP ← tempESP;
    SS ← tempSS;
FI;
FOR each SegReg in (ES, FS, GS, and DS)
    DO
        tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
        IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code))
            THEN (* Segment register invalid *)
                SegmentSelector ← 0; (*Segment selector becomes null*)
        FI;
    OD;
IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP ← SP + SRC;
            FI;
        FI;
(* IA-32e Mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
                IF first or second doubleword on stack is not in canonical space
                    THEN #SS(0); FI;
            ELSE
                IF OperandSize = 16
                    THEN
                        IF second word on stack is not within stack limits
                            THEN #SS(0); FI;
                        IF first or second word on stack is not in canonical space
                            THEN #SS(0); FI;
                    ELSE (* OperandSize = 64 *)
IF first or second quadword on stack is not in canonical space
    THEN #SS(0); FI;
FI
IF return code segment selector is NULL
    THEN GP(0); FI;
IF return code segment selector addresses descriptor beyond descriptor table limit
    THEN GP(selector); FI;
IF return code segment selector addresses descriptor in non-canonical space
    THEN GP(selector); FI;
Obtain descriptor to which return code segment selector points from descriptor table;
IF return code segment descriptor is not a code segment
    THEN #GP(selector); FI;
IF return code segment descriptor has L-bit = 1 and D-bit = 1
    THEN #GP(selector); FI;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); FI;
IF return code segment descriptor is non-conforming
and return code segment DPL ≠ return code segment selector RPL
    THEN #GP(selector); FI;
IF return code segment descriptor is not present
    THEN #NP(selector); FI:
IF return code segment selector RPL < CPL
    THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
FI;
FI;
IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
            FI;
        FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
THEN
ESP ← ESP + SRC;
ELSE
  IF StackAddressSize = 16
    THEN
      SP ← SP + SRC;
      ELSE (* StackAddressSize = 64 *)
        RSP ← RSP + SRC;
    FI;  
FI;
FI;

IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
  THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
  THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
  THEN
    IF new CS descriptor L-bit = 0
      THEN #GP(selector);
    IF stack segment selector RPL = 3
      THEN #GP(selector);
    FI;
IF return stack segment descriptor is not within descriptor table limits
  THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
  THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
  THEN #GP(selector); FI;
IF stack segment not present
  THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
  THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
  THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
  THEN
    EIP ← Pop();
    CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
    CS(RPL) ← CPL;
    IF instruction has immediate operand
      THEN (* Release parameters from called procedure's stack *)
        IF StackAddressSize = 32
          THEN
            ESP ← ESP + SRC;
ELSE
  IF StackAddressSize = 16
    THEN
      SP ← SP + SRC;
ELSE (* StackAddressSize = 64 *)
  RSP ← RSP + SRC;
  FI;
  FI;
  FI;
  tempESP ← Pop();
tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
  ESP ← tempESP;
  SS ← tempSS;
ELSE
  IF OperandSize = 16
    THEN
      EIP ← Pop();
      EIP ← EIP AND 0000FFFFH;
      CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
      CS(RPL) ← CPL;
      IF instruction has immediate operand
        THEN (* Release parameters from called procedure's stack *)
          IF StackAddressSize = 32
            THEN
              ESP ← ESP + SRC;
          ELSE
            IF StackAddressSize = 16
              THEN
                SP ← SP + SRC;
          ELSE (* StackAddressSize = 64 *)
            RSP ← RSP + SRC;
          FI;
        FI;
        FI;
        tempESP ← Pop();
tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
      ESP ← tempESP;
      SS ← tempSS;
ELSE (* OperandSize = 64 *)
  RIP ← Pop();
  CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; segment descriptor loaded *)
  CS(RPL) ← CPL;
  IF instruction has immediate operand
    THEN (* Release parameters from called procedure's stack *)
      RSP ← RSP + SRC;
    FI;
    tempESP ← Pop();
tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; segment descriptor loaded *)
  ESP ← tempESP;
  SS ← tempSS;
  FI;
  FI;

FOR each of segment register (ES, FS, GS, and DS)
DO
  IF segment register points to data or non-conforming code segment
  and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
  THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
  FI;
OD;

IF instruction has immediate operand
  THEN (* Release parameters from calling procedure's stack *)
    IF StackAddressSize = 32
      THEN
        ESP ← ESP + SRC;
      ELSE StackAddressSize = 16
        THEN
          SP ← SP + SRC;
        ELSE (* StackAddressSize = 64 *)
          RSP ← RSP + SRC;
        FI;
    FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the return code or stack segment selector is NULL.
If the return instruction pointer is not within the return code segment limit
#GP(selector) If the RPL of the return code segment selector is less than the CPL.
If the return code or stack segment selector index is not within its descriptor table limits.
If the return code segment descriptor does not indicate a code segment.
If the return code segment is non-conforming and the segment selector’s DPL is not equal to
the RPL of the code segment’s segment selector
If the return code segment is conforming and the segment selector’s DPL greater than the RPL
of the code segment’s segment selector
If the stack segment is not a writable data segment.
If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
If the stack segment descriptor DPL is not equal to the RPL of the return code segment
selector.
#SS(0) If the top bytes of stack are not within stack limits.
If the return stack segment is not present.
#NP(selector) If the return code segment is not present.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions
#GP If the return instruction pointer is not within the return code segment limit
#SS If the top bytes of stack are not within stack limits.
Virtual-8086 Mode Exceptions

#GP(0) If the return instruction pointer is not within the return code segment limit.
#SS(0) If the top bytes of stack are not within stack limits.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions
Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0) If the return instruction pointer is non-canonical.
If the return instruction pointer is not within the return code segment limit.
If the stack segment selector is NULL going back to compatibility mode.
If the stack segment selector is NULL going back to CPL3 64-bit mode.
If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
If the return code segment selector is NULL.

#GP(selector) If the proposed segment descriptor for a code segment does not indicate it is a code segment.
If the proposed new code segment descriptor has both the D-bit and L-bit set.
If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
If CPL is greater than the RPL of the code segment selector.
If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
If a segment selector index is outside its descriptor table limits.
If a segment descriptor memory address is non-canonical.
If the stack segment is not a writable data segment.
If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

#SS(0) If an attempt to pop a value off the stack violates the SS limit.
If an attempt to pop a value off the stack causes a non-canonical address to be referenced.

#NP(selector) If the return code or stack segment is not present.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
RORX — Rotate Right Logical Without Affecting Flags

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.F2.0F3A.W0 */r */ib RORX r32, r/m32, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>BMI2</td>
<td>Rotate 32-bit r/m32 right imm8 times without affecting arithmetic flags.</td>
</tr>
<tr>
<td>VEX.LZ.F2.0F3A.W1 */r */ib RORX r64, r/m64, imm8</td>
<td>RMI</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Rotate 64-bit r/m64 right imm8 times without affecting arithmetic flags.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description
Rotates the bits of second operand right by the count value specified in imm8 without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation
IF (OperandSize = 32)
y ← imm8 AND 1FH;
DEST ← (SRC >> y) | (SRC << (32-y));
ELSEIF (OperandSize = 64)
y ← imm8 AND 3FH;
DEST ← (SRC >> y) | (SRC << (64-y));
ENDIF

### Flags Affected
None

### Intel C/C++ Compiler Intrinsic Equivalent
Auto-generated from high-level language.

### SIMD Floating-Point Exceptions
None

### Other Exceptions
See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.
### ROUND PD — Round Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode* Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mod Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 09 Ir ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Round packed double precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 09 Ir ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round packed double-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.WIG 09 Ir ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round packed double-precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Round the 2 double-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a double-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Operation

IF (imm[2] = ‘1)  
    THEN  // rounding mode is determined by MXCSR.RC  
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);  
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);  
    ELSE  // rounding mode is determined by IMM8.RC  
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);  
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);  
FI

ROUNDPD (128-bit Legacy SSE version)
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)  
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)  
DEST[MAXVL-1:128] ← 0  

VROUNDPD (VEX.128 encoded version)
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)  
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)  
DEST[MAXVL-1:128] ← 0

VROUNDPD (VEX.256 encoded version)
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)  
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)  
DEST[191:128] ← RoundToInteger(SRC[191:128], ROUND_CONTROL)  
DEST[255:192] ← RoundToInteger(SRC[255:192], ROUND_CONTROL)
Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_round_pd(__m128d s1, int iRoundMode);
__m128 _mm_floor_pd(__m128d s1);
__m128 _mm_ceil_pd(__m128d s1)
__m256 _mm256_round_pd(__m256d s1, int iRoundMode);
__m256 _mm256_floor_pd(__m256d s1);
__m256 _mm256_ceil_pd(__m256d s1)

SIMD Floating-Point Exceptions
Invalid (signaled only if SRC = SNaN)
Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)
Note that Denormal is not signaled by ROUNDPD.

Other Exceptions
See Exceptions Type 2; additionally
#UD If VEX.vvvv ≠ 1111B.
ROUNDPS — Round Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode*/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 08 /r ib ROUNDS xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Round packed single precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round packed single-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS ymm1, ymm2/m256, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round packed single-precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Round the 4 single-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

IF (imm[2] = '1')
    THEN  // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE  // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;

ROUNDPS (128-bit Legacy SSE version)

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)

VROUNDPS (VEX.128 encoded version)

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0

VROUNDPS (VEX.256 encoded version)

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[159:128] ← RoundToInteger(SRC[159:128], ROUND_CONTROL)
DEST[191:160] ← RoundToInteger(SRC[191:160], ROUND_CONTROL)
DEST[223:192] ← RoundToInteger(SRC[223:192], ROUND_CONTROL)
DEST[255:224] ← RoundToInteger(SRC[255:224], ROUND_CONTROL)

Intel C/C++ Compiler Intrinsic Equivalent

__m128 __mm_round_ps(__m128 s1, int iRoundMode);
__m128 __mm_floor_ps(__m128 s1);
__m128 __mm_ceil_ps(__m128 s1)
__m256 __mm256_round_ps(__m256 s1, int iRoundMode);
__m256 __mm256_floor_ps(__m256 s1);
__m256 __mm256_ceil_ps(__m256 s1)
**SIMD Floating-Point Exceptions**

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

**Other Exceptions**

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.
ROUNDSD — Round Scalar Double Precision Floating-Point Values

**Op/En/ Instruction**  |  **Op/ En** |  **64/32 bit Mode Support** |  **CPUID Feature Flag** |  **Description**
---|---|---|---|---
66 0F 3A 0B /r ib  | RMI | V/V | SSE4_1 | Round the low packed double precision floating-point value in xmm2/m64 and place the result in xmm1. The rounding mode is determined by imm8.
VEX.NDS.LIG.66.0F3A.WIG 0B /r ib  | RVMI | V/V | AVX | Round the low packed double precision floating-point value in xmm3/m64 and place the result in xmm1. The rounding mode is determined by imm8. Upper packed double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Round the DP FP value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double-precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1’ then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation**

IF (imm[2] = ’1’)
    THEN  // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE  // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;

**ROUNDSD (128-bit Legacy SSE version)**

DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[MAXVL-1:64] (Unmodified)
VROUNDSD (VEX.128 encoded version)
DEST[63:0] ← RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VROUNDSD: __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
          __m128d mm_floor_sd(__m128d dst, __m128d s1);
          __m128d mm_ceil_sd(__m128d dst, __m128d s1);

SIMD Floating-Point Exceptions
Invalid (signaled only if SRC = SNaN)
Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)
Note that Denormal is not signaled by VROUNDSD.

Other Exceptions
See Exceptions Type 3.
ROUNDSS — Round Scalar Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode*/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 0A /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Round the low packed single precision floating-point value in xmm2/m32 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.66.0F3A.WIG 0A /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round the low packed single precision floating-point value in xmm3/m32 and place the result in xmm1. The rounding mode is determined by imm8. Also, upper packed single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Round the single-precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single-precision floating-point input to an integer value and returns the result as a single-precision floating-point value in the lowest position. The upper three single-precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1’ then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation**

IF (imm[2] = ‘1’)
    THEN // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged;

**ROUNDSS (128-bit Legacy SSE version)**

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[MAXVL-1:32] (Unmodified)
VROUNDSS (VEX.128 encoded version)
DEST[31:0] ← RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VROUNDSS:  __m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
            __m128 mm_floor_ss(__m128 dst, __m128 s1);
            __m128 mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions
Invalid (signaled only if SRC = SNaN)
Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)
Note that Denormal is not signaled by ROUNDSS.

Other Exceptions
See Exceptions Type 3.
RSM—Resume from System Management Mode

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF AA</td>
<td>RSM</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Resume operation of interrupted program.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.


Operation

ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH )
    THEN
        ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
    FI

Flags Affected

All.

Protected Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.
    If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode* Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 52 /r RSQRTPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in xmm2/m128 and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F:WIG 52 /r VRSQRTPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of the square roots of packed single-precision values in xmm2/mem and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F:WIG 52 /r VRSQRTPS ymm1, ymm2/m256</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of the square roots of packed single-precision values in ymm2/mem and stores the results in ymm1.</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

\[ |\text{Relative Error}| \leq 1.5 \times 2^{-12} \]

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \( \infty \) of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than \( -0.0 \)), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Operation

**RSQRTPS (128-bit Legacy SSE version)**
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC[127:96]))
DEST[MAXVL-1:128] (Unmodified)

**VRSQRTPS (VEX.128 encoded version)**
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC[127:96]))
DEST[MAXVL-1:128] ← 0

**VRSQRTPS (VEX.256 encoded version)**
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC[127:96]))
DEST[159:128] ← APPROXIMATE(1/SQRT(SRC[159:128]))
DEST[191:160] ← APPROXIMATE(1/SQRT(SRC[191:160]))
DEST[223:192] ← APPROXIMATE(1/SQRT(SRC[223:192]))
DEST[255:224] ← APPROXIMATE(1/SQRT(SRC[255:224]))

**Intel C/C++ Compiler Intrinsic Equivalent**
RSQRTPS: __m128 _mm_rsqrt_ps(__m128 a)
RSQRTPS: __m256 _mm256_rsqrt_ps (__m256 a);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Exceptions Type 4; additionally
#UD If VEX.vvvv ≠ 1111B.
RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

\[ |\text{Relative Error}| \leq 1.5 \times 2^{-12} \]

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \( \infty \) of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than \( -0.0 \)), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

RSQRTSS (128-bit Legacy SSE version)
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[MAXVL-1:32] (Unmodified)

VRSQRTSS (VEX.128 encoded version)
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[MAXVL-1:128] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
RSQRTSS: _m128 __mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 5.
**SAHF—Store AH into Flags**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9E</td>
<td>SAHF</td>
<td>ZO</td>
<td>Invalid*</td>
<td>Valid</td>
<td>Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register.</td>
</tr>
</tbody>
</table>

**NOTES:**
* Valid in specific steppings. See Description section.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

**Operation**

IF IA-64 Mode

THEN

IF CPUID.80000001H:ECX[0] = 1;

THEN

RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;

ELSE

#UD;

ELSE

EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;

FI;

**Flags Affected**

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

**Protected Mode Exceptions**

None.

**Real-Address Mode Exceptions**

None.

**Virtual-8086 Mode Exceptions**

None.

**Compatibility Mode Exceptions**

None.
64-Bit Mode Exceptions

#UD If CPUID.80000001H.ECX[0] = 0.
If the LOCK prefix is used.
## SAL/SAR/SHL/SHR—Shift

<table>
<thead>
<tr>
<th>Opcode***</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 /4</td>
<td>SAL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, once.</td>
</tr>
<tr>
<td>REX + D0 /4</td>
<td>SAL r/m8**, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, once.</td>
</tr>
<tr>
<td>D2 /4</td>
<td>SAL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>REX + D2 /4</td>
<td>SAL r/m8**, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>C0 /4 ib</td>
<td>SAL r/m8, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /4 ib</td>
<td>SAL r/m8**, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SAL r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, once.</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SAL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SAL r/m16, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /4 ib</td>
<td>SAL r/m64, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Multiply r/m64 by 2, imm8 times.</td>
</tr>
<tr>
<td>D0 /7</td>
<td>SAR r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m8 by 2, once.</td>
</tr>
<tr>
<td>REX + D0 /7</td>
<td>SAR r/m8**, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide* r/m8 by 2, once.</td>
</tr>
<tr>
<td>D2 /7</td>
<td>SAR r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>REX + D2 /7</td>
<td>SAR r/m8**, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide* r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>C0 /7 ib</td>
<td>SAR r/m8, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m8 by 2, imm8 time.</td>
</tr>
<tr>
<td>REX + C0 /7 ib</td>
<td>SAR r/m8**, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide* r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /7</td>
<td>SAR r/m16,1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m16 by 2, once.</td>
</tr>
<tr>
<td>D3 /7</td>
<td>SAR r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m16 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /7 ib</td>
<td>SAR r/m16, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m16 by 2, imm8 times.</td>
</tr>
<tr>
<td>REX.W + D1 /7</td>
<td>SAR r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide* r/m64 by 2, once.</td>
</tr>
<tr>
<td>D3 /7</td>
<td>SAR r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m32 by 2, CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /7</td>
<td>SAR r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide* r/m64 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /7 ib</td>
<td>SAR r/m32, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide* r/m32 by 2, imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /7 ib</td>
<td>SAR r/m64, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide* r/m64 by 2, imm8 times.</td>
</tr>
<tr>
<td>D0 /4</td>
<td>SHL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, once.</td>
</tr>
<tr>
<td>REX + D0 /4</td>
<td>SHL r/m8**, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, once.</td>
</tr>
<tr>
<td>D2 /4</td>
<td>SHL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>REX + D2 /4</td>
<td>SHL r/m8**, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>C0 /4 ib</td>
<td>SHL r/m8, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /4 ib</td>
<td>SHL r/m8**, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m16,1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, once.</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SHL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SHL r/m16, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m32,1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m32 by 2, once.</td>
</tr>
</tbody>
</table>
Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position’s shifted value with the sign of the unshifted value (see Figure 7-9 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the “quotient” of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the “remainder” is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction’s default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

**IA-32 Architecture Compatibility**

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

**Operation**

IF 64-Bit Mode and using REX.W

THEN

countMASK ← 3FH;

ELSE

countMASK ← 1FH;

FI

tempCOUNT ← (COUNT AND countMASK);

tempDEST ← DEST;

WHILE (tempCOUNT ≠ 0)

DO

IF instruction is SAL or SHL

THEN

CF ← MSB(DEST);

ELSE (* Instruction is SAR or SHR *)

CF ← LSB(DEST);

FI;

IF instruction is SAL or SHL

THEN

DEST ← DEST * 2;

ELSE

IF instruction is SAR
THEN
    DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
ELSE (* Instruction is SHR *)
    DEST ← DEST / 2; (* Unsigned divide *)
FI;
FI;
tempCOUNT ← tempCOUNT - 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
            FI;
        ELSE IF (COUNT AND countMASK) = 0
            THEN
                All flags unchanged;
            ELSE (* COUNT not 1 or 0 *)
                OF ← undefined;
        FI;
    FI;

Flags Affected
The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.
**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.
SARX/SHLX/SHRX — Shift Without Affecting Flags

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.LZ.F3.0F38.W0 F7 /r SARX r32a, r/m32, r32b</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Shift r/m32 arithmetically right with count specified in r32b.</td>
</tr>
<tr>
<td>VEX.NDS.LZ66.0F38.W0 F7 /r SHLX r32a, r/m32, r32b</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Shift r/m32 logically left with count specified in r32b.</td>
</tr>
<tr>
<td>VEX.NDS.LZ.F2.0F38.W0 F7 /r SHRX r32a, r/m32, r32b</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Shift r/m32 logically right with count specified in r32b.</td>
</tr>
<tr>
<td>VEX.NDS.LZ.F3.0F38.W1 F7 /r SARX r64a, r/m64, r64b</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Shift r/m64 arithmetically right with count specified in r64b.</td>
</tr>
<tr>
<td>VEX.NDS.LZ66.0F38.W1 F7 /r SHLX r64a, r/m64, r64b</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Shift r/m64 logically left with count specified in r64b.</td>
</tr>
<tr>
<td>VEX.NDS.LZ.F2.0F38.W1 F7 /r SHRX r64a, r/m64, r64b</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Shift r/m64 logically right with count specified in r64b.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>VEX.vvvv (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

If the value specified in the first source operand exceeds OperandSize -1, the COUNT value is masked. SARX, SHRX, and SHLX instructions do not update flags.

Operation

\[
\text{TEMP} \leftarrow \text{SRC1} ; \\
\text{IF VEX.W1 and CS.L = 1} \\
\text{THEN} \\
\quad \text{countMASK} \leftarrow \text{3FH} ; \\
\text{ELSE} \\
\quad \text{countMASK} \leftarrow \text{1FH} ; \\
\text{FI} \\
\text{COUNT} \leftarrow (\text{SRC2 AND countMASK}) \\
\text{DEST[OperandSize -1]} \leftarrow \text{TEMP[OperandSize -1]} ; \\
\text{DO WHILE (COUNT} \neq 0) \\
\quad \text{IF instruction is SHLX} \\
\quad \text{THEN} \\
\quad \quad \text{DEST[]} \leftarrow \text{DEST} \ast 2 ;
\]

SARX/SHLX/SHRX — Shift Without Affecting Flags
ELSE IF instruction is SHRX
    THEN
        DEST[] ← DEST /2; // unsigned divide
    ELSE
        // SARX
        DEST[] ← DEST /2; // signed divide, round toward negative infinity
    FI;
    COUNT ← COUNT - 1;
OD

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
Auto-generated from high-level language.

SIMD Floating-Point Exceptions
None

Other Exceptions
See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.
SBB—Integer Subtraction with Borrow

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1C ib</td>
<td>SBB AL,imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm8 from AL.</td>
</tr>
<tr>
<td>1D iw</td>
<td>SBB AX,imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm16 from AX.</td>
</tr>
<tr>
<td>1D id</td>
<td>SBB EAX,imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm32 from EAX.</td>
</tr>
<tr>
<td>REX.W + 1D id</td>
<td>SBB RAX,imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow sign-extended imm32 to 64-bits from RAX.</td>
</tr>
<tr>
<td>80 /3 ib</td>
<td>SBB r/m8,imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm8 from r/m8.</td>
</tr>
<tr>
<td>REX + 80 /3 ib</td>
<td>SBB r/m8*,imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow imm8 from r/m8.</td>
</tr>
<tr>
<td>81 /3 iw</td>
<td>SBB r/m16,imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm16 from r/m16.</td>
</tr>
<tr>
<td>81 /3 id</td>
<td>SBB r/m32,imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm32 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /3 id</td>
<td>SBB r/m64,imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow sign-extended imm32 to 64-bits from r/m64.</td>
</tr>
<tr>
<td>83 /3 ib</td>
<td>SBB r/m16,imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow sign-extended imm8 from r/m16.</td>
</tr>
<tr>
<td>83 /3 ib</td>
<td>SBB r/m32,imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow sign-extended imm8 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 83 /3 ib</td>
<td>SBB r/m64,imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow sign-extended imm8 from r/m64.</td>
</tr>
<tr>
<td>18 /r</td>
<td>SBB r/m8,r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r8 from r/m8.</td>
</tr>
<tr>
<td>REX + 18 /r</td>
<td>SBB r/m8*,r8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r8 from r/m8.</td>
</tr>
<tr>
<td>19 /r</td>
<td>SBB r/m16,r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r16 from r/m16.</td>
</tr>
<tr>
<td>19 /r</td>
<td>SBB r/m32,r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r32 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 19 /r</td>
<td>SBB r/m64,r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r64 from r/m64.</td>
</tr>
<tr>
<td>1A /r</td>
<td>SBB r8,r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r/m8 from r8.</td>
</tr>
<tr>
<td>REX + 1A /r</td>
<td>SBB r8*,r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r/m8 from r8.</td>
</tr>
<tr>
<td>1B /r</td>
<td>SBB r16,r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r/m16 from r16.</td>
</tr>
<tr>
<td>1B /r</td>
<td>SBB r32,r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r/m32 from r32.</td>
</tr>
<tr>
<td>REX.W + 1B /r</td>
<td>SBB r64,r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r/m64 from r64.</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM+r/m (w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM+r/m (w)</td>
<td>ModRM+reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM+reg (w)</td>
<td>ModRM+r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← (DEST – (SRC + CF));

Intel C/C++ Compiler Intrinsic Equivalent

SBB: extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);
SBB: extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);
SBB: extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *diff_out);
SBB: extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *diff_out);

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
     If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
     If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
### Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES:(E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes...
some action based on the setting of status flags. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

Non-64-bit Mode:

IF (Byte comparison)
   THEN
      temp ← AL − SRC;
      SetStatusFlags(temp);
      THEN IF DF = 0
         THEN (E)DI ← (E)DI + 1;
         ELSE (E)DI ← (E)DI - 1; FI;
      ELSE IF (Word comparison)
         THEN
         temp ← AX − SRC;
         SetStatusFlags(temp);
         IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI - 2; FI;
         FI;
      ELSE IF (Doubleword comparison)
         THEN
         temp ← EAX - SRC;
         SetStatusFlags(temp);
         IF DF = 0
            THEN (E)DI ← (E)DI + 4;
            ELSE (E)DI ← (E)DI - 4; FI;
         FI;
   FI;

64-bit Mode:

IF (Byte comparison)
   THEN
      temp ← AL − SRC;
      SetStatusFlags(temp);
      THEN IF DF = 0
         THEN (R|E)DI ← (R|E)DI + 1;
         ELSE (R|E)DI ← (R|E)DI - 1; FI;
      ELSE IF (Word comparison)
         THEN
         temp ← AX − SRC;
         SetStatusFlags(temp);
         IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 2;
            ELSE (R|E)DI ← (R|E)DI - 2; FI;
         FI;
ELSE IF (Doubleword comparison)
    THEN
        temp ← EAX – SRC;
        SetStatusFlags(temp);
        IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 4;
            ELSE (R|E)DI ← (R|E)DI – 4; FI;
        FI;
    ELSE IF (Quadword comparison using REX.W )
        THEN
            temp ← RAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 8;
                ELSE (R|E)DI ← (R|E)DI – 8; FI;
            FI;
        FI;

Flags Affected
The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the limit of the ES segment.
    If the ES register contains a NULL segment selector.
    If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
## SETcc—Set Byte on Condition

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 97</td>
<td>SETA r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>REX + 0F 97</td>
<td>SETA r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 93</td>
<td>SETAE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if above or equal (CF=0).</td>
</tr>
<tr>
<td>REX + 0F 93</td>
<td>SETAE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if above or equal (CF=0).</td>
</tr>
<tr>
<td>0F 92</td>
<td>SETB r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if below (CF=1).</td>
</tr>
<tr>
<td>REX + 0F 92</td>
<td>SETB r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if below (CF=1).</td>
</tr>
<tr>
<td>0F 96</td>
<td>SETBE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>REX + 0F 96</td>
<td>SETBE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 92</td>
<td>SETC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if carry (CF=1).</td>
</tr>
<tr>
<td>REX + 0F 92</td>
<td>SETC r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if carry (CF=1).</td>
</tr>
<tr>
<td>0F 94</td>
<td>SETD r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if equal (ZF=1).</td>
</tr>
<tr>
<td>REX + 0F 94</td>
<td>SETD r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if equal (ZF=1).</td>
</tr>
<tr>
<td>0F 9F</td>
<td>SETG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9F</td>
<td>SETG r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 9D</td>
<td>SETGE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9D</td>
<td>SETGE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 9C</td>
<td>SETL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if less (SF≠ OF).</td>
</tr>
<tr>
<td>REX + 0F 9C</td>
<td>SETL r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if less (SF≠ OF).</td>
</tr>
<tr>
<td>0F 9E</td>
<td>SETLE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if less or equal (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>REX + 0F 9E</td>
<td>SETLE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if less or equal (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>0F 96</td>
<td>SETNA r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>REX + 0F 96</td>
<td>SETNA r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 92</td>
<td>SETNAE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not above or equal (CF=1).</td>
</tr>
<tr>
<td>REX + 0F 92</td>
<td>SETNAE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 93</td>
<td>SETNB r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not below (CF=0).</td>
</tr>
<tr>
<td>REX + 0F 93</td>
<td>SETNB r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not below (CF=0).</td>
</tr>
<tr>
<td>0F 97</td>
<td>SETNBE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>REX + 0F 97</td>
<td>SETNBE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 93</td>
<td>SETNC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not carry (CF=0).</td>
</tr>
<tr>
<td>REX + 0F 93</td>
<td>SETNC r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not carry (CF=0).</td>
</tr>
<tr>
<td>0F 95</td>
<td>SETNE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not equal (ZF=0).</td>
</tr>
<tr>
<td>REX + 0F 95</td>
<td>SETNE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 9E</td>
<td>SETNG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not greater (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>REX + 0F 9E</td>
<td>SETNG r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not greater (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>0F 9C</td>
<td>SETNGE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not greater or equal (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9C</td>
<td>SETNGE r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 9D</td>
<td>SETNL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not less (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9D</td>
<td>SETNL r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 9F</td>
<td>SETNLE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not less or equal (ZF=0 and SF=OF).</td>
</tr>
</tbody>
</table>
### Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (cc) indicates the condition being tested for.

The terms "above" and "below" are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SETcc instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNL (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, "EFLAGS Condition Codes," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SETcc instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Notes:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
The reg field of the ModR/M byte is not used for the SETCC instruction and those opcode bits are ignored by the processor.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction’s operation is the same as in legacy mode and compatibility mode.

**Operation**

IF condition
    THEN DEST ← 1;
    ELSE DEST ← 0;
FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If the destination is located in a non-writable segment.
    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.
SFENCE—Store Fence

Description
Orders processor execution relative to all memory stores prior to the SFENCE instruction. The processor ensures that every store prior to SFENCE is globally visible before any store after SFENCE becomes globally visible. The SFENCE instruction is ordered with respect to memory stores, other SFENCE instructions, MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to memory loads or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

Operation
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);

Intel C/C++ Compiler Intrinsic Equivalent
void _mm_sfence(void)

Exceptions (All Operating Modes)
#UD If CPUID.01H:EDX.SSE[bit 25] = 0.  
If the LOCK prefix is used.
**SGDT—Store Global Descriptor Table Register**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /0</td>
<td>SGDT m</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Store GDTR to m.</td>
</tr>
</tbody>
</table>

**Description**

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 or 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

**IA-32 Architecture Compatibility**

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

**Operation**

IF instruction is SGDT
  IF OperandSize = 16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
    THEN
      DEST[0:15] ← GDTR(Limit);
      DEST[16:47] ← GDTR(Base); (* Full 32-bit base address stored *)
      FI;
  ELSE (* 64-bit Mode *)
    DEST[0:15] ← GDTR(Limit);
    DEST[16:79] ← GDTR(Base); (* Full 64-bit base address stored *)
  FI;

**Flags Affected**

None.
Protected Mode Exceptions

#UD If the LOCK prefix is used.
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment
selector.
If CR4.UMIP = 1 and CPL > 0.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If CR4.UMIP = 1.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#UD If the LOCK prefix is used.
#GP(0) If the memory address is in a non-canonical form.
If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
SHA1RNDS4—Perform Four Rounds of SHA1 Operation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 3A CC /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA1RNDS4 xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description
The SHA1RNDS4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

Operation
SHA1RNDS4
The function f() and Constant K are dependent on the value of the immediate.

\[
\begin{align*}
    \text{IF } (\text{imm8}[1:0] = 0) & \text{ THEN } f() \leftarrow f0(), K \leftarrow K_0; \\
    \text{ELSE IF } (\text{imm8}[1:0] = 1) & \text{ THEN } f() \leftarrow f1(), K \leftarrow K_1; \\
    \text{ELSE IF } (\text{imm8}[1:0] = 2) & \text{ THEN } f() \leftarrow f2(), K \leftarrow K_2; \\
    \text{ELSE IF } (\text{imm8}[1:0] = 3) & \text{ THEN } f() \leftarrow f3(), K \leftarrow K_3; \\
    f();
\end{align*}
\]

A \leftarrow SRC1[127:96];
B \leftarrow SRC1[95:64];
C \leftarrow SRC1[63:32];
D \leftarrow SRC1[31:0];
W0E \leftarrow SRC2[127:96];
W1 \leftarrow SRC2[95:64];
W2 \leftarrow SRC2[63:32];
W3 \leftarrow SRC2[31:0];

Round i = 0 operation:
A_1 \leftarrow f(B, C, D) + (A ROL 5) + W0E + K;
B_1 \leftarrow A;
C_1 \leftarrow B ROL 30;
D_1 \leftarrow C;
E_1 \leftarrow D;

FOR i = 1 to 3
A_(i+1) \leftarrow f(B_i, C_i, D_i) + (A_i ROL 5) + W_i + E_i + K;
B_(i+1) \leftarrow A_i;
C_{i+1} \leftarrow B_i \text{ ROL 30;}
D_{i+1} \leftarrow C_i;
E_{i+1} \leftarrow D_i;
\text{ENDFOR}

\text{DEST[127:96]} \leftarrow A_4;
\text{DEST[95:64]} \leftarrow B_4;
\text{DEST[63:32]} \leftarrow C_4;
\text{DEST[31:0]} \leftarrow D_4;

\textbf{Intel C/C++ Compiler Intrinsic Equivalent}
SHA1RNDS4: \_mm_sha1rnds4_epu32(__m128i, __m128i, const int);

\textbf{Flags Affected}
None

\textbf{SIMD Floating-Point Exceptions}
None

\textbf{Other Exceptions}
See Exceptions Type 4.
SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 3B CB /r</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

Operation

SHA1NEXTE

```
TMP ← (SRC1[127:96] ROL 30);  
DEST[95:64] ← SRC2[95:64];  
DEST[63:32] ← SRC2[63:32];  
DEST[31:0] ← SRC2[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
SHA1NEXTE: __m128i _mm_sha1nexte_epu32(__m128i, __m128i);
```

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.
SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 C9 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.</td>
</tr>
</tbody>
</table>

SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

**Operation**

SHA1MSG1

\[
\begin{align*}
W_0 & \leftarrow \text{SRC1}[127:96]; \\
W_1 & \leftarrow \text{SRC1}[95:64]; \\
W_2 & \leftarrow \text{SRC1}[63:32]; \\
W_3 & \leftarrow \text{SRC1}[31:0]; \\
W_4 & \leftarrow \text{SRC2}[127:96]; \\
W_5 & \leftarrow \text{SRC2}[95:64]; \\
\text{DEST}[127:96] & \leftarrow W_2 \text{ XOR } W_0; \\
\text{DEST}[95:64] & \leftarrow W_3 \text{ XOR } W_1; \\
\text{DEST}[63:32] & \leftarrow W_4 \text{ XOR } W_2; \\
\text{DEST}[31:0] & \leftarrow W_5 \text{ XOR } W_3;
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA1MSG1: `__m128i _mm_sha1msg1_epu32(__m128i, __m128i);`

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.
SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 CA r</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

**Operation**

SHA1MSG2

W13 ← SRC2[95:64];
W14 ← SRC2[63:32];
W15 ← SRC2[31:0];
W16 ← (SRC1[127:96] XOR W13) ROL 1;
W17 ← (SRC1[95:64] XOR W14) ROL 1;
W18 ← (SRC1[63:32] XOR W15) ROL 1;
W19 ← (SRC1[31:0] XOR W16) ROL 1;

DEST[127:96] ← W16;
DEST[95:64] ← W17;
DEST[63:32] ← W18;
DEST[31:0] ← W19;

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA1MSG2: __m128i _mm_sha1msg2_epu32(__m128i, __m128i);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.
SHA256RNDS2—Perform Two Rounds of SHA256 Operation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 CB /r</td>
<td>RMI</td>
<td>V/V</td>
<td>SHA</td>
<td>Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>Implicit XMM0 (r)</td>
</tr>
</tbody>
</table>

**Description**

The SHA256RNDS2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

**Operation**

SHA256RNDS2

A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
WK0 ← XMM0[31:0];
WK1 ← XMM0[63:32];

FOR i = 0 to 1
    A_(i +1) ← Ch (E_i, F_i, G_i) + Σ1( E_i) + WK_i + H_i + Maj(A_i, B_i, C_i) + Σ0( A_i);
    B_(i +1) ← A_i;
    C_(i +1) ← B_i;
    D_(i +1) ← C_i;
    E_(i +1) ← Ch (E_i, F_i, G_i) + Σ1( E_i) + WK_i + H_i + D_i;
    F_(i +1) ← E_i;
    G_(i +1) ← F_i;
    H_(i +1) ← G_i;
ENDFOR

DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;
Intel C/C++ Compiler Intrinsic Equivalent
SHA256RNDS2: __m128i __mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);

Flags Affected
None

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4.
SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 3B CC /r</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.</td>
</tr>
<tr>
<td>SHA256MSG1 Xmm1, Xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModR:mreg (r, w)</td>
<td>ModR:mreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**
The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

**Operation**

**SHA256MSG1**

\[
\begin{align*}
W4 & \leftarrow SRC2[31:0 ]; \\
W3 & \leftarrow SRC1[127:96 ]; \\
W2 & \leftarrow SRC1[95:64 ]; \\
W1 & \leftarrow SRC1[63:32 ]; \\
W0 & \leftarrow SRC1[31:0 ];
\end{align*}
\]

\[
\begin{align*}
\text{DEST}[127:96] & \leftarrow W3 + \sigma_0( W4); \\
\text{DEST}[95:64] & \leftarrow W2 + \sigma_0( W3); \\
\text{DEST}[63:32] & \leftarrow W1 + \sigma_0( W2); \\
\text{DEST}[31:0] & \leftarrow W0 + \sigma_0( W1);
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA256MSG1: __m128i _mm_sha256msg1_epu32(__m128i, __m128i);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.
SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 CD /r SHA256MSG2 xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

Operation

SHA256MSG2

W14 ← SRC2[95:64];
W15 ← SRC2[127:96];
W16 ← SRC1[31:0] + σ1( W14 );
W17 ← SRC1[63:32] + σ1( W15 );
W18 ← SRC1[95:64] + σ1( W16 );
W19 ← SRC1[127:96] + σ1( W17 );

DEST[127:96] ← W19;
DEST[95:64] ← W18;
DEST[63:32] ← W17;
DEST[31:0] ← W16;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG2 : __m128i _mm_sha256msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.
**SHLD—Double Precision Shift Left**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F A4 /r ib</td>
<td>SHLD r/m16, r16, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to left imm8 places while shifting bits from r16 in from the right.</td>
</tr>
<tr>
<td>0F A5 /r</td>
<td>SHLD r/m16, r16, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to left CL places while shifting bits from r16 in from the right.</td>
</tr>
<tr>
<td>0F A4 /r ib</td>
<td>SHLD r/m32, r32, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to left imm8 places while shifting bits from r32 in from the right.</td>
</tr>
<tr>
<td>REX.W + 0F A4 /r ib</td>
<td>SHLD r/m64, r64, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to left imm8 places while shifting bits from r64 in from the right.</td>
</tr>
<tr>
<td>0F A5 /r</td>
<td>SHLD r/m32, r32, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to left CL places while shifting bits from r32 in from the right.</td>
</tr>
<tr>
<td>REX.W + 0F A5 /r</td>
<td>SHLD r/m64, r64, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to left CL places while shifting bits from r64 in from the right.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>MRC</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>CL</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF (In 64-Bit Mode and REX.W = 1)
  THEN COUNT ← COUNT MOD 64;
  ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
  THEN
  No operation;
  ELSE
IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, SIZE – COUNT];
        (* Last bit shifted out on exit *)
        FOR i ← SIZE – 1 DOWN TO COUNT
            DO
                Bit(DEST, i) ← Bit(DEST, i – COUNT);
            OD;
        FOR i ← COUNT – 1 DOWN TO 0
            DO
                BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
            OD;
        FI;
    FI;

Flags Affected
If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF,
and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred;
otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is unde-
fined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags
are undefined.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
    current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
SHRD—Double Precision Shift Right

### Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AC /r ib</td>
<td>SHRD r/m16, r16, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to right imm8 places while shifting bits from r16 in from the left.</td>
</tr>
<tr>
<td>0F AD /r</td>
<td>SHRD r/m16, r16, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to right CL places while shifting bits from r16 in from the left.</td>
</tr>
<tr>
<td>0F AC /r ib</td>
<td>SHRD r/m32, r32, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to right imm8 places while shifting bits from r32 in from the left.</td>
</tr>
<tr>
<td>REX.W + 0F AC /r ib</td>
<td>SHRD r/m64, r64, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to right imm8 places while shifting bits from r64 in from the left.</td>
</tr>
<tr>
<td>0F AD /r</td>
<td>SHRD r/m32, r32, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to right CL places while shifting bits from r32 in from the left.</td>
</tr>
<tr>
<td>REX.W + 0F AD /r</td>
<td>SHRD r/m64, r64, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to right CL places while shifting bits from r64 in from the left.</td>
</tr>
</tbody>
</table>

### Instruction Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>NA</td>
</tr>
<tr>
<td>MRC</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>CL</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

### Operation

IF (In 64-Bit Mode and REX.W = 1)
   THEN COUNT ← COUNT MOD 64;
   ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
   THEN
      No operation;
   ELSE

4-610  Vol. 2B   SHRD—Double Precision Shift Right
IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, COUNT – 1]; (* Last bit shifted out on exit *)
        FOR i ← 0 TO SIZE – 1 – COUNT
            DO
                BIT[DEST, i] ← BIT[DEST, i + COUNT];
            OD;
        FOR i ← SIZE – COUNT TO SIZE – 1
            DO
                BIT[DEST, i] ← BIT[SRC, i + COUNT – SIZE];
            OD;
    FI;
FI;

Flags Affected
If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used.
SHUFPD—Packed Interleave Shuffle of Pairs of Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F C6 / r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shuffle two pairs of double-precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128:66.0F:W1 / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128:66.0F:W1 / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle eight pairs of double-precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. Store interleaved results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512:66.0F:W1 / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle eight pairs of double-precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. Store interleaved results in xmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Selects a double-precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double-precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to receive the shuffled result of an input pair.

EVEN encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4] are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2] are ignored.
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2] are ignored.

**Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of DP FP Values**

**Operation**

**VSHUFPD (EVEX encoded versions when SRC2 is a vector register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF IMMO[0] = 0

THEN TMP_DEST[63:0] ← SRC1[63:0]

ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;

IF IMMO[1] = 0

THEN TMP_DEST[127:64] ← SRC2[63:0]

ELSE TMP_DEST[127:64] ← SRC2[127:64] FI;

IF VL >= 256

IF IMMO[2] = 0


IF IMMO[3] = 0


FI;

IF VL >= 512

IF IMMO[4] = 0

THEN TMP_DEST[319:256] ← SRC1[319:256]


IF IMMO[5] = 0


IF IMMO[6] = 0


IF IMMO[7] = 0

THEN TMP_DEST[511:448] ← SRC2[447:384]

ELSE TMP_DEST[511:448] ← SRC2[511:448] FI;

FI;

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] ← TMP_DEST[i+63:i]

ELSE
IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+63:i] ← 0
FI

VSHUFPD (EVEX encoded versions when SRC2 is memory)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] ← TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] ← TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
    IF IMM0[3] = 0
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
    IF IMM0[5] = 0
    IF IMM0[6] = 0
    IF IMM0[7] = 0
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
  DEST[i+63:i] ← 0
  Fi;
ENDIF
DEST[MAXVL-1:VL] ← 0

VSHUFPD (VEX.256 encoded version)
IF IMM0[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] Fi;
IF IMM0[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] Fi;
IF IMM0[2] = 0
IF IMM0[3] = 0
DEST[MAXVL-1:256] (Unmodified)

VSHUFPD (VEX.128 encoded version)
IF IMM0[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] Fi;
IF IMM0[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] Fi;
DEST[MAXVL-1:128] (Unmodified)

VSHUFPD (128-bit Legacy SSE version)
IF IMM0[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] Fi;
IF IMM0[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] Fi;
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VSHUFDP __m512d __mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFDP __m512d __mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFDP __m512d __mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFDP __m256d __mm256_shuffle_pd(__m256d a, __m256d b, __m256d c, __mmask8 k, __m256d d, __m256d e);
VSHUFDP __m256d __mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, __m256d c, __m256d d, __m256d e);
VSHUFDP __m256d __mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, __m256d c, __m256d d, __m256d e);
VSHUFDP __m128d __mm_shuffle_pd(__m128d a, __m128d b, __mmask8 k,
   __m128d c, __m128d d, __m128d e);
VSHUFDP __m128d __mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, __m128d c, __m128d d, __m128d e);
VSHUFDP __m128d __mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, __m128d d, __m128d e);
**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.
SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Select from quadruplet of single-precision floating-point values in xmm1 and xmm3/m128 using imm8, interleaved result pairs are stored in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Select from quadruplet of single-precision floating-point values in ymm1 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 C6 /r ib VSHUFPS xmm1{k1}[z], xmm2, xmm3/m128/m32bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 C6 /r ib VSHUFPS ymm1{k1}[z], ymm2, ymm3/m256/m32bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Select from quadruplet of single-precision floating-point values in ymm1 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}[z], zmm2, zmm3/m512/m32bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Select from quadruplet of single-precision floating-point values in zmm1 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Selects a single-precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for each element of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.
128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

**Operation**

```c
Select4(SRC, control) {
  CASE (control[1:0]) OF
    0: TMP ← SRC[31:0];
    1: TMP ← SRC[63:32];
    2: TMP ← SRC[95:64];
    3: TMP ← SRC[127:96];
  ESAC;
  RETURN TMP
}
```

**VPSHUFPS (EVEX encoded versions when SRC2 is a vector register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```c
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
  TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
FI;
IF VL >= 512
  TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
  TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
  TMP_DEST[479:448] ← Select4(SRC2[511:384], imm8[5:4]);
  TMP_DEST[511:480] ← Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
```

**Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result**

---

**SHUFPS**—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values
i ← j * 32
IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPUSHUFPs (EVEX encoded versions when SRC2 is memory)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[479:448] ← Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**VSHUFPS (VEX.256 encoded version)**
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
DEST[MAXVL-1:256] ← 0

**VSHUFPS (VEX.128 encoded version)**
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] ← 0

**SHUFPS (128-bit Legacy SSE version)**
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.
SIDT—Store Interrupt Descriptor Table Register

**Description**
Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

**IA-32 Architecture Compatibility**
The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

**Operation**

IF instruction is SIDT

THEN

IF OperandSize = 16 or OperandSize = 32 (* Legacy or Compatibility Mode *)

THEN

DEST[0:15] ← IDTR(Limit);
DEST[16:47] ← IDTR(Base); Fl; (* Full 32-bit base address stored *)

ELSE (* 64-bit Mode *)

DEST[0:15] ← IDTR(Limit);
DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)

Fl;

Fl;

**Flags Affected**
None.
Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If CR4.UMIP = 1 and CPL > 0.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If CR4.UMIP = 1.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#UD If the LOCK prefix is used.

#GP(0) If the memory address is in a non-canonical form.
If CR4.UMIP = 1 and CPL > 0.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
SLDT—Store Local Descriptor Table Register

**Description**

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

**Operation**

\[ \text{DEST} \leftarrow \text{LDTR(\text{SegmentSelector});} \]

**Flags Affected**

None.

**Protected Mode Exceptions**

- **#GP(0)**
  - If the destination is located in a non-writable segment.
  - If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
  - If CR4.UMIP = 1 and CPL > 0.
- **#SS(0)**
  - If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**
  - If a page fault occurs.
- **#AC(0)**
  - If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
- **#UD**
  - If the LOCK prefix is used.
Real-Address Mode Exceptions
#UD The SLDT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The SLDT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
    If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD If the LOCK prefix is used.
**SMSW—Store Machine Status Word**

**Opcode**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /4</td>
<td>SMSW r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Store machine status word to r/m16.</td>
</tr>
<tr>
<td>0F 01 /4</td>
<td>SMSW r32/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Store machine status word in low-order 16 bits of r32/m16; high-order 16 bits of r32 are undefined.</td>
</tr>
<tr>
<td>REX.W + 0F 01 /4</td>
<td>SMSW r64/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Store machine status word in low-order 16 bits of r64/m16; high-order 16 bits of r32 are undefined.</td>
</tr>
</tbody>
</table>

**Description**

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a general-purpose register or a memory location.

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:

- SMSW r16 operand size 16, store CR0[15:0] in r16
- SMSW r32 operand size 32, zero-extend CR0[31:0], and store in r32
- SMSW r64 operand size 64, zero-extend CR0[63:0], and store in r64
- SMSW m16 operand size 16, store CR0[15:0] in m16
- SMSW m16 operand size 32, store CR0[15:0] in m16 (not m32)
- SMSW m16 operands size 64, store CR0[15:0] in m16 (not m64)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs if CR4.UMIP = 0. It is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on IA-32 and Intel 64 processors beginning with the Intel386 processors should use the MOV CR instruction to load the machine status word.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

\[
\text{DEST} \leftarrow \text{CR0}[15:0];
\]

(* Machine status word *)

**Flags Affected**

None.
Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If CR4.UMIP = 1 and CPL > 0.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If CR4.UMIP = 1.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD If the LOCK prefix is used.
SQRTPD—Square Root of Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 51 /r SQRTPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1G 51 /r VSQRTPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F:W1G 51 /r VSQRTPD ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 51 /r VSQRTPD xmm1[k1]{k1}[z], xmm2/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes Square Roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 51 /r VSQRTPD ymm1[k1]{k1}[z], ymm2/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes Square Roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 51 /r VSQRTPD zmm1[k1]{k1}[z], zmm2/m512/m64bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes Square Roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD computation of the square roots of the two, four or eight packed double-precision floating-point values in the source operand (the second operand) stores the packed double-precision floating-point results in the destination operand (the first operand).

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

**VEX.256 encoded version:** The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

**VSQRTPD (EVEX encoded versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

IF \((VL = 512)\) AND \((EVEX.b = 1)\) AND \((SRC *is register*)\)

THEN

\[SET_RM(EVEX.RC);\]
ELSE

\[SET_RM(MXCSR.RM);\]

FI;

FOR \(j \leftarrow 0\) TO \(KL-1\)

\[i \leftarrow j * 64\]

IF \(k1[j]\) OR \(*no writemask*\) THEN

IF \((EVEX.b = 1)\) AND \((SRC *is memory*)\)

THEN \(DEST[j+63:i] \leftarrow \text{SQRT}(SRC[63:0])\)
ELSE \(DEST[j+63:i] \leftarrow \text{SQRT}(SRC[i+63:i])\)

FI;
ELSE

IF \(*merging-masking*\)

THEN \(*DEST[j+i+63:i] remains unchanged*\)
ELSE \(*zeroing-masking*\)

\[DEST[j+i+63:i] \leftarrow 0\]

FI

ENDFOR

\[DEST[\text{MAXVL}-1:VL] \leftarrow 0\]

**VSQRTPD (VEX.256 encoded version)**

\[DEST[63:0] \leftarrow \text{SQRT}(SRC[63:0])\]
\[DEST[127:64] \leftarrow \text{SQRT}(SRC[127:64])\]
\[DEST[191:128] \leftarrow \text{SQRT}(SRC[191:128])\]
\[DEST[255:192] \leftarrow \text{SQRT}(SRC[255:192])\]
\[DEST[\text{MAXVL}-1:256] \leftarrow 0\]

**VSQRTPD (VEX.128 encoded version)**

\[DEST[63:0] \leftarrow \text{SQRT}(SRC[63:0])\]
\[DEST[127:64] \leftarrow \text{SQRT}(SRC[127:64])\]
\[DEST[\text{MAXVL}-1:128] \leftarrow 0\]

**SQRTPD (128-bit Legacy SSE version)**

\[DEST[63:0] \leftarrow \text{SQRT}(SRC[63:0])\]
\[DEST[127:64] \leftarrow \text{SQRT}(SRC[127:64])\]
\[DEST[\text{MAXVL}-1:128] \text{(Unmodified)}\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[VSQRTPD __m512d _mm512_sqrt_round_pd(__m512d a, int r);\]
\[VSQRTPD __m512d _mm512_mask_sqrt_round_pd(__m512d s, __mmask8 k, __m512d a, int r);\]
\[VSQRTPD __m512d _mm512_maskz_sqrt_round_pd(__mmask8 k, __m512d a, int r);\]
\[VSQRTPD __m256d _mm256_sqrt_pd (__m256d a);\]
\[VSQRTPD __m256d _mm256_mask_sqrt_pd(__m256d s, __mmask8 k, __m256d a, int r);\]
\[VSQRTPD __m256d _mm256_maskz_sqrt_pd(__mmask8 k, __m256d a, int r);\]
\[SQRTPD __m128d _mm_sqrt_pd (__m128d a);\]
\[VSQRTPD __m128d _mm_mask_sqrt_pd(__m128d s, __mmask8 k, __m128d a, int r);\]
\[VSQRTPD __m128d _mm_maskz_sqrt_pd(__mmask8 k, __m128d a, int r);\]
SIMD Floating-Point Exceptions
Invalid, Precision, Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 2; additionally
#UD If VEX.vvv != 111B.
EVEX-encoded instruction, see Exceptions Type E2.
#UD If EVEX.vvv != 111B.
SQRTPS—Square Root of Single-Precision Floating-Point Values

**Opcode/Instruction** | **Op/En** | **CPUID Feature Flag** | **Description**
--- | --- | --- | ---
NP 0F 51 /r SQRTPS xmm1, xmm2/m128 | A | SSE | Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.

VEX.128.0F:WIG 51 /r VSQRTPS xmm1, xmm2/m128 | A | AVX | Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.

VEX.256.0F:WIG 51/r VSQRTPS ymm1, ymm2/m256 | A | AVX | Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1.

EVEX.128.0F:W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst | B | AVX512VL AVX512F | Computes Square Roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.

EVEX.256.0F:W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst | B | AVX512VL AVX512F | Computes Square Roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.

EVEX.512.0F:W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er} | B | AVX512F | Computes Square Roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg(w)</td>
<td>ModRM:r/m(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg(w)</td>
<td>ModRM:r/m(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand second operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

VSQRTPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j \leftarrow 0 TO KL-1
    i \leftarrow j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+31:i] \leftarrow SQRT(SRC[31:0])
        ELSE DEST[i+31:i] \leftarrow SQRT(SRC[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] \leftarrow 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0

VSQRTPS (VEX.256 encoded version)

DEST[31:0] \leftarrow SQRT(SRC[31:0])
DEST[63:32] \leftarrow SQRT(SRC[63:32])
DEST[95:64] \leftarrow SQRT(SRC[95:64])
DEST[127:96] \leftarrow SQRT(SRC[127:96])
DEST[159:128] \leftarrow SQRT(SRC[159:128])
DEST[191:160] \leftarrow SQRT(SRC[191:160])
DEST[223:192] \leftarrow SQRT(SRC[223:192])
DEST[255:224] \leftarrow SQRT(SRC[255:224])

VSQRTPS (VEX.128 encoded version)

DEST[31:0] \leftarrow SQRT(SRC[31:0])
DEST[63:32] \leftarrow SQRT(SRC[63:32])
DEST[95:64] \leftarrow SQRT(SRC[95:64])
DEST[127:96] \leftarrow SQRT(SRC[127:96])
DEST[MAXVL-1:128] \leftarrow 0

SQRTPS (128-bit Legacy SSE version)

DEST[31:0] \leftarrow SQRT(SRC[31:0])
DEST[63:32] \leftarrow SQRT(SRC[63:32])
DEST[95:64] \leftarrow SQRT(SRC[95:64])
DEST[127:96] \leftarrow SQRT(SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

 VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
 VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
 VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
 VSQRTPS __m256 _mm256_sqrt_ps(__m256 a);
 VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
 VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally
#UD If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E2.
#UD If EVEX.vvvv != 1111B.
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 51/r SQRTSD xmm1,xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Computes square root of the low double-precision floating-point value in xmm2/m64 and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F2.0F:WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F:W1 51/r VSQRTSD xmm1 [k1][z], xmm2, xmm3/m64{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Computes the square root of the low double-precision floating-point value in the second source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
**Operation**

**VSQRTSD (EVEX encoded version)**

IF (EVEX.b = 1) AND (SRC2 *is register*)
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;

IF k1[0] or *no writemask*
   THEN DEST[63:0] ← SQRT(SRC2[63:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[63:0] remains unchanged*
           ELSE ; zeroing-masking
               THEN DEST[63:0] ← 0
       FI;
   FI;

DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**VSQRTSD (VEX.128 encoded version)**

DEST[63:0] ← SQRT(SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**SQRSTD (128-bit Legacy SSE version)**

DEST[63:0] ← SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRSTD __m128d _mm_sqrt_sd (__m128d a, __m128d b)

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
**SQRTSS—Compute Square Root of Scalar Single-Precision Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 51 /r SQRTSS xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VSQRTSS (EVEX encoded version)
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SQRT(SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            DEST[31:0] ← 0
        FI;
    FI;
DEST[MAXVL-1:128] ← 0

VSQRTSS (VEX.128 encoded version)
DEST[31:0] ← SQRT(SRC2[31:0])
DEST[MAXVL-1:128] ← 0

SQRSS (128-bit Legacy SSE version)
DEST[31:0] ← SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss( __mmask8 k, __m128 a, __m128 b, int r);
SQRSS __m128 _mm_sqrt_ss(__m128 a)

SIMD Floating-Point Exceptions
Invalid, Precision, Denormal

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.
STAC—Set AC Flag in EFLAGS Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 CB STAC</td>
<td>ZO</td>
<td>V/V</td>
<td>SMAP</td>
<td>Set the AC flag in the EFLAGS register.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register. This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute STAC when CPL > 0 cause #UD.

### Operation

EFLAGS.AC ← 1;

### Flags Affected

AC set. Other flags are unaffected.

### Protected Mode Exceptions

- #UD If the LOCK prefix is used.
- If the CPL > 0.
- If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.
- If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Virtual-8086 Mode Exceptions

- #UD The STAC instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

- #UD If the LOCK prefix is used.
- If the CPL > 0.
- If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### 64-Bit Mode Exceptions

- #UD If the LOCK prefix is used.
- If the CPL > 0.
- If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.
STC—Set Carry Flag

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F9</td>
<td>STC</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set CF flag.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Sets the CF flag in the EFLAGS register. Operation is the same in all modes.

### Operation

CF ← 1;

### Flags Affected

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
STD—Set Direction Flag

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD</td>
<td>STD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set DF flag.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

DF ← 1;

Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
STI—Set Interrupt Flag

### Description

In most cases, STI sets the interrupt flag (IF) in the EFLAGS register. This allows the processor to respond to maskable hardware interrupts.

If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI. (The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure or subroutine. For instance, if an STI instruction is followed by a RET instruction, the RET instruction is allowed to execute before external interrupts are recognized. No interrupts can be recognized if an execution of CLI immediately follow such an execution of STI.) The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and nonmaskable interrupts (NMIs). However, NMIs (and system-management interrupts) may be inhibited on the instruction boundary following an execution of STI that begins with IF = 0.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3, EFLAGS.VIP = 1, and either VME mode or PVI mode is active, STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-19 indicates the action of the STI instruction depending on the processor operating mode, IOPL, CPL, and EFLAGS.VIP.

<table>
<thead>
<tr>
<th>Mode</th>
<th>IOPL</th>
<th>EFLAGS.VIP</th>
<th>STI Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-address</td>
<td>X^1</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>Protected, not PVI^2</td>
<td>≥ CPL</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td></td>
<td>&lt; CPL</td>
<td></td>
<td>#GP fault</td>
</tr>
<tr>
<td>Protected, PVI^3</td>
<td>3</td>
<td>0</td>
<td>VIF = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>#GP fault</td>
</tr>
<tr>
<td>Virtual-8086, not VME^3</td>
<td>3</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>Virtual-8086, VME^3</td>
<td>3</td>
<td>0</td>
<td>VIF = 1</td>
</tr>
</tbody>
</table>

Table 4-19. Decision Table for STI Results

**NOTES:**
1. X = This setting has no effect on instruction operation.
2. For this table, “protected mode” applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.

**Operation**

IF CR0.PE = 0 (* Executing in real-address mode *)
    THEN IF ← 1; (* Set Interrupt Flag *)
ELSE
    IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
        THEN IF ← 1; (* Set Interrupt Flag *)
    ELSE
        IF VME mode OR PVI mode
            THEN
                IF EFLAGS.VIP = 0
                    THEN VIF ← 1; (* Set Virtual Interrupt Flag *)
                ELSE #GP(0);
                    Fl;
                ELSE #GP(0);
                    Fl;
            ELSE
                Fl;
        FI;
    FI;
FI;

**Flags Affected**

Either the IF flag or the VIF flag is set to 1. Other flags are unaffected.

**Protected Mode Exceptions**

#GP(0) If CPL is greater than IOPL and PVI mode is not active.  
      If CPL is greater than IOPL and EFLAGS.VIP = 1.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If IOPL is less than 3 and VME mode is not active.  
      If IOPL is less than 3 and EFLAGS.VIP = 1.

#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
STMXCSR—Store MXCSR Register State

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /3 SXTMXCSR m32</td>
<td>M</td>
<td>V/V</td>
<td>SSE</td>
<td>Store contents of MXCSR register to m32.</td>
</tr>
<tr>
<td>VEX.LZ.0F.wIG AE /3 VSTMXCSR m32</td>
<td>M</td>
<td>V/V</td>
<td>AVX</td>
<td>Store contents of MXCSR register to m32.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

VEX.L must be 0, otherwise instructions will #UD.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation
m32 ← MXCSR;

Intel C/C++ Compiler Intrinsic Equivalent
_mm_getcsr(void)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 5; additionally

#UD
If VEX.L= 1,
If VEX.vvvv ≠ 1111B.
**STOS/STOSB/STOSW/STOSD/STOSQ—Store String**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>STOS m8</td>
<td>NA</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m16</td>
<td>NA</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m32</td>
<td>NA</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.</td>
</tr>
<tr>
<td>REX.W + AB</td>
<td>STOS m64</td>
<td>NA</td>
<td>Valid</td>
<td>N.E.</td>
<td>Store RAX at address RDI or EDI.</td>
</tr>
<tr>
<td>AA</td>
<td>STOSB</td>
<td>NA</td>
<td>Valid</td>
<td>For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.</td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>STOSW</td>
<td>NA</td>
<td>Valid</td>
<td>For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.</td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>STOSD</td>
<td>NA</td>
<td>Valid</td>
<td>For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.</td>
<td></td>
</tr>
<tr>
<td>REX.W + AB</td>
<td>STOSQ</td>
<td>NA</td>
<td>Valid</td>
<td>N.E.</td>
<td>Store RAX at address RDI or EDI.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the ES:(E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).
NOTE: To improve performance, more recent processors support modifications to the processor’s operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for additional information on fast-string operation.

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Operation
Non-64-bit Mode:
IF (Byte store)
THEN
  DEST ← AL;
  THEN IF DF = 0
  THEN (E)DI ← (E)DI + 1;
  ELSE (E)DI ← (E)DI - 1;
  FI;
ELSE IF (Word store)
THEN
  DEST ← AX;
  THEN IF DF = 0
  THEN (E)DI ← (E)DI + 2;
  ELSE (E)DI ← (E)DI - 2;
  FI;
FI;
ELSE IF (Doubleword store)
THEN
  DEST ← EAX;
  THEN IF DF = 0
  THEN (E)DI ← (E)DI + 4;
  ELSE (E)DI ← (E)DI - 4;
  FI;
FI;
64-bit Mode:
IF (Byte store)
THEN
  DEST ← AL;
  THEN IF DF = 0
  THEN (R|E)DI ← (R|E)DI + 1;
  ELSE (R|E)DI ← (R|E)DI - 1;
  FI;
ELSE IF (Word store)
THEN
  DEST ← AX;
THEN IF DF = 0
  THEN (R|E)DI ← (R|E)DI + 2;
ELSE (R|E)DI ← (R|E)DI - 2;
FI;
FI;
ELSE IF (Doubleword store)
THEN
  DEST ← EAX;
  THEN IF DF = 0
    THEN (R|E)DI ← (R|E)DI + 4;
    ELSE (R|E)DI ← (R|E)DI - 4;
  FI;
ELSE IF (Quadword store using REX.W)
THEN
  DEST ← RAX;
  THEN IF DF = 0
    THEN (R|E)DI ← (R|E)DI + 8;
    ELSE (R|E)DI ← (R|E)DI - 8;
  FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the limit of the ES segment.
If the ES register contains a NULL segment selector.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the ES segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the ES segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0)       If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0)       If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD           If the LOCK prefix is used.
STR—Store Task Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /1</td>
<td>STR r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Stores segment selector from TR in r/m16.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operation

DEST ← TR(SegmentSelector);

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If CR4.UMIP = 1 and CPL > 0.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The STR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The STR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
    If CR4.UMIP = 1 and CPL > 0.
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
### SUB—Subtract

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2C ib</td>
<td>SUB AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract imm8 from AL.</td>
</tr>
<tr>
<td>2D iw</td>
<td>SUB AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract imm16 from AX.</td>
</tr>
<tr>
<td>2D id</td>
<td>SUB EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract imm32 from EAX.</td>
</tr>
<tr>
<td>REX.W + 2D id</td>
<td>SUB RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract imm32 sign-extended to 64-bits from RAX.</td>
</tr>
<tr>
<td>80 /5 ib</td>
<td>SUB r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract imm8 from r/m8.</td>
</tr>
<tr>
<td>REX + 80 /5 ib</td>
<td>SUB r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract imm8 from r/m8.</td>
</tr>
<tr>
<td>81 /5 iw</td>
<td>SUB r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract imm16 from r/m16.</td>
</tr>
<tr>
<td>81 /5 id</td>
<td>SUB r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract imm32 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /5 id</td>
<td>SUB r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract imm32 sign-extended to 64-bits from r/m64.</td>
</tr>
<tr>
<td>83 /5 ib</td>
<td>SUB r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract sign-extended imm8 from r/m16.</td>
</tr>
<tr>
<td>83 /5 ib</td>
<td>SUB r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract sign-extended imm8 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 83 /5 ib</td>
<td>SUB r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract sign-extended imm8 from r/m64.</td>
</tr>
<tr>
<td>28 /r</td>
<td>SUB r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract r8 from r/m8.</td>
</tr>
<tr>
<td>REX + 28 /r</td>
<td>SUB r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract r8 from r/m8.</td>
</tr>
<tr>
<td>29 /r</td>
<td>SUB r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract r16 from r/m16.</td>
</tr>
<tr>
<td>29 /r</td>
<td>SUB r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract r32 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 29 /r</td>
<td>SUB r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract r64 from r/m64.</td>
</tr>
<tr>
<td>2A /r</td>
<td>SUB r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract r/m8 from r8.</td>
</tr>
<tr>
<td>REX + 2A /r</td>
<td>SUB r8*, r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract r/m8 from r8.</td>
</tr>
<tr>
<td>2B /r</td>
<td>SUB r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract r/m16 from r16.</td>
</tr>
<tr>
<td>2B /r</td>
<td>SUB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract r/m32 from r32.</td>
</tr>
<tr>
<td>REX.W + 2B /r</td>
<td>SUB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract r/m64 from r64.</td>
</tr>
</tbody>
</table>

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM/r/m (r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM/r/m (r, w)</td>
<td>ModRM/reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM/reg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.
The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

**Operation**

\[ \text{DEST} \leftarrow (\text{DEST} - \text{SRC}); \]

**Flags Affected**

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

**Protected Mode Exceptions**

- #GP(0) If the destination is located in a non-writable segment.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
SUBPD—Subtract Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5C /r SUBPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 5C /r VSUBPD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 5C /r VSUBPD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 5C /r VSUBPD zmm1 (k1){z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Subtract packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD subtract of the two, four or eight packed double-precision floating-point values of the second Source operand from the first Source operand, and stores the packed double-precision floating-point results in the destination operand.

VEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.
**Operation**

**VSUBPD (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

    SET_RM(EVEX.RC);

ELSE

    SET_RM(MXCSR.RM);

FI;

FOR j \(\leftarrow\) 0 TO KL-1

\(i \leftarrow j \times 64\)

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] \(\leftarrow\) SRC1[i+63:i] - SRC2[i+63:i]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

    DEST[63:0] \(\leftarrow\) 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] \(\leftarrow\) 0

**VSUBPD (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j \(\leftarrow\) 0 TO KL-1

\(i \leftarrow j \times 64\)

IF k1[j] OR *no writemask* THEN

IF (EVEX.b = 1)

THEN DEST[i+63:i] \(\leftarrow\) SRC1[i+63:i] - SRC2[63:0];

ELSE EST[i+63:i] \(\leftarrow\) SRC1[i+63:i] - SRC2[i+63:i];

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

    DEST[63:0] \(\leftarrow\) 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] \(\leftarrow\) 0

**VSUBPD (VEX.256 encoded version)**

DEST[63:0] \(\leftarrow\) SRC1[63:0] - SRC2[63:0]

DEST[127:64] \(\leftarrow\) SRC1[127:64] - SRC2[127:64]


DEST[255:192] \(\leftarrow\) SRC1[255:192] - SRC2[255:192]

DEST[MAXVL-1:256] \(\leftarrow\) 0
VSUBPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] - SRC2[127:64]
DEST[MAXVL-1:128] ← 0

SUBPD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VSUBPD __m512d _mm512_sub_pd (__m512d a, __m512d b);
VSUBPD __m512d _mm512_mask_sub_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_maskz_sub_pd (__mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_sub_round_pd (__m512d a, __m512d b, int);
VSUBPD __m512d _mm512_mask_sub_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m512d _mm512_maskz_sub_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);
VSUBPD __m256d _mm256_mask_sub_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VSUBPD __m256d _mm256_maskz_sub_pd (__mmask8 k, __m256d a, __m256d b);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);
VSUBPD __m128d _mm_mask_sub_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBPD __m128d _mm_maskz_sub_pd (__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
SUBPS—Subtract Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5C /r SUBPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 5C /r VSUBPS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 5C /r VSUBPS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 5C /r VSUBPS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

**VSUBPS (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
          DEST[31:0] ← 0
        FI;
      FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

**VSUBPS (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0];
        ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i];
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
          DEST[31:0] ← 0
        FI;
      FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

**VSUBPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[MAXVL-1:256] ← 0
VSUBPS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[MAXVL-1:128] ← 0

SUBPS (128-bit Legacy SSE version)
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VSUBPS __m512_mm512_sub_ps (__m512 a, __m512 b);
VSUBPS __m512_mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VSUBPS __m512_mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);
VSUBPS __m512_mm512_sub_round_ps (__m512 a, __m512 b, int);
VSUBPS __m512_mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m512_mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m256_mm256_sub_ps (__m256 a, __m256 b);
VSUBPS __m256_mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VSUBPS __m256_mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);
SUBPS __m128_mm_sub_ps (__m128 a, __m128 b);
VSUBPS __m128_mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBPS __m128_mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
SUBSD—Subtract Scalar Double-Precision Floating-Point Value

### Opcode/Instruction
- **F2 0F 5C /r SUBSD xmm1, xmm2/m64**: A V/V SSE2 Subtract the low double-precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
- **VEX.NDS.LIG.F2.0F:W1 5C /r VSUBSD xmm1,xmm2, xmm3/m64**: B V/V AVX Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
- **EVEX.NDS.LIG.F2.0F:W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}**: C V/V AVX512F Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMx/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMx/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMx/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description
Subtract the low double-precision floating-point value in the second source operand from the first source operand and stores the double-precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VSUBSD (EVEX encoded version)
IF (SRC2 "is register") AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or "no writemask"
    THEN DEST[63:0] \( \leftarrow \) SRC1[63:0] - SRC2[63:0]
    ELSE
        IF "merging-masking" ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] \( \leftarrow \) 0
        FI;
    FI;
DEST[127:64] \( \leftarrow \) SRC1[127:64]
DEST[MAXVL-1:128] \( \leftarrow \) 0

VSUBSD (VEX.128 encoded version)
DEST[63:0] \( \leftarrow \) SRC1[63:0] - SRC2[63:0]
DEST[127:64] \( \leftarrow \) SRC1[127:64]
DEST[MAXVL-1:128] \( \leftarrow \) 0

SUBSD (128-bit Legacy SSE version)
DEST[63:0] \( \leftarrow \) DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
SUBSS—Subtract Scalar Single-Precision Floating-Point Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 5C /r SUBSS xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
</tr>
<tr>
<td>VEX.NDS.LIG.F3.0F:WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 5C /r VSUBSS xmm1 (k1)[{z}], xmm2, xmm3/m32{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Subtract the low single-precision floating-point value from the second source operand and the first source operand and store the double-precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VSUBSS (EVEX encoded version)
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
            FI;
    FI;
DEST[MAXVL-1:128] ← 0

VSUBSS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[MAXVL-1:128] ← 0

SUBSS (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
**SWAPGS—Swap GS Base Register**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 F8</td>
<td>SWAPGS</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Exchanges the current GS base register value with the value contained in MSR address C0000102H.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

**Operation**

IF CS.L ≠ 1 (* Not in 64-Bit Mode *)

THEN

    #UD; Fl;

IF CPL ≠ 0

THEN #GP(0); Fl;

tmp ← GS.base;
GS.base ← IA32_KERNEL_GS_BASE;
IA32_KERNEL_GS_BASE ← tmp;

**Flags Affected**

None

**Protected Mode Exceptions**

#UD If Mode ≠ 64-Bit.

**Real-Address Mode Exceptions**

#UD If Mode ≠ 64-Bit.

**Virtual-8086 Mode Exceptions**

#UD If Mode ≠ 64-Bit.
Compatibility Mode Exceptions

#UD If Mode ≠ 64-Bit.

64-Bit Mode Exceptions

#GP(0) If CPL ≠ 0.
#UD If the LOCK prefix is used.
**SYSCALL—Fast System Call**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 05</td>
<td>SYSCALL</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Fast call to privilege level 0 system procedures.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

**Operation**

IF (CS.L ≠ 1) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1) (* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)

THEN #UD;

FI;

RCX ← RIP; (* Will contain address of next instruction *)
RIP ← IA32_LSTAR;
R11 ← RFLAGS;
RFLAGS ← RFLAGS AND NOT(IA32_FMASK);

CS.Selector ← IA32_STAR[47:32] AND FFFCH (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0; (* Flat segment *)
CS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11; (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
CS.L ← 1; (* Entry is to 64-bit mode *)
CS.D ← 0; (* Required if CS.L = 1 *)
CS.G ← 1; (* 4-KByte granularity *)
CPL ← 0;
SS.Selector ← IA32_STAR[47:32] + 8; (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0; (* Flat segment *)
SS.Limit ← FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3; (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1; (* 32-bit stack segment *)
SS.G ← 1; (* 4-KByte granularity *)

Flags Affected

All.

Protected Mode Exceptions

#UD The SYSCALL instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The SYSCALL instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The SYSCALL instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The SYSCALL instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD If IA32_EFER.SCE = 0.
If the LOCK prefix is used.
**SYSENTER—Fast System Call**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 34</td>
<td>SYSENTER</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fast call to privilege level 0 system procedures.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment selectors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.
The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

IF CPUID SEP bit is set
  THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
      THEN
        SYSENTER/SYSEXIT_Not_Supported; Fl;
      ELSE
        SYSENTER/SYSEXIT_Supported; Fl;
  FL;

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation
IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); Fl;

RFLAGS.VM ← 0; (* Ensures protected mode execution *)
RFLAGS.IF ← 0; (* Mask interrupts *)
IF in IA-32e mode
  THEN
    RSP ← IA32_SYSENTER_ESP;
    RIP ← IA32_SYSENTER_EIP;
  ELSE
    ESP ← IA32_SYSENTER_ESP[31:0];
    EIP ← IA32_SYSENTER_EIP[31:0];
  FI;

CS.Selector ← IA32_SYSENTER_CS[15:0] AND FFFCH;
(* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0; (* Flat segment *)
CS.Limit ← FFFFFFFH; (* With 4-KByte granularity, implies a 4-Byte limit *)
CS.Type ← 11; (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
IF in IA-32e mode
  THEN
    CS.L ← 1; (* Entry is to 64-bit mode *)
    CS.D ← 0; (* Required if CS.L = 1 *)
  ELSE
    CS.L ← 0;
    CS.D ← 1; (* 32-bit code segment *)
  FI;
CS.G ← 1; (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← CS.Selector + 8; (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0; (* Flat segment *)
SS.Limit ← FFFFFFFH; (* With 4-KByte granularity, implies a 4-Byte limit *)
SS.Type ← 3; (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1; (* 32-bit stack segment*)
SS.G ← 1; (* 4-KByte granularity *)

Flags Affected
VM, IF (see Operation above)

Protected Mode Exceptions
#GP(0) If IA32_SYSENTER_CS[15:2] = 0.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP The SYSENTER instruction is not recognized in real-address mode.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**SYSEXIT—Fast Return from Fast System Call**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 35</td>
<td>SYSEXIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fast return to privilege level 3 user code.</td>
</tr>
<tr>
<td>REX.W + 0F 35</td>
<td>SYSEXIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fast return to 64-bit mode privilege level 3 user code.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENDER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32__SYSENTER__CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32__SYSENTER__CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENDER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENDER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENDER/SYSEXIT instructions are actually present. For example:

```plaintext
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENDER/SYSEXIT_Not_Supported; Fl;
        ELSE
            SYSENDER/SYSEXIT_Supported; Fl;
    Fl;

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENDER/SYSEXIT instructions.
Operation

IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;

IF operand size is 64-bit
   THEN (* Return to 64-bit mode *)
      RSP ← RCX;
      RIP ← RDX;
   ELSE (* Return to protected mode or compatibility mode *)
      RSP ← ECX;
      RIP ← EDX;
   FI;

IF operand size is 64-bit (* Operating system provides CS; RPL forced to 3 *)
   THEN CS.Selector ← IA32_SYSENTER_CS[15:0] + 32;
   ELSE CS.Selector ← IA32_SYSENTER_CS[15:0] + 16;
   FI;
CS.Selector ← CS.Selector OR 3; (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0; (* Flat segment *)
CS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11; (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF operand size is 64-bit (* return to 64-bit mode *)
   THEN (* 64-bit code segment *)
      CS.L ← 1;
      CS.D ← 0; (* Required if CS.L = 1 *)
   ELSE (* return to protected mode or compatibility mode *)
      CS.L ← 0;
      CS.D ← 1; (* 32-bit code segment*)
   FI;
CS.G ← 1; (* 4-KByte granularity *)
CPL ← 3;
SS.Selector ← CS.Selector + 8; (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0; (* Flat segment *)
SS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3; (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1; (* 32-bit stack segment*)
SS.G ← 1; (* 4-KByte granularity *)

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If IA32_SYSENTER_CS[15:2] = 0.
   If CPL ≠ 0.
#UD If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP The SYSEXIT instruction is not recognized in real-address mode.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) The SYSEXIT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If IA32_SYSENTER_CS = 0.
   If CPL ≠ 0.
   If RCX or RDX contains a non-canonical address.
#UD If the LOCK prefix is used.
SYSRET—Return From Fast System Call

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 07</td>
<td>SYSRET</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Return to compatibility mode from fast system call</td>
</tr>
<tr>
<td>REX.W + 0F 07</td>
<td>SYSRET</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Return to 64-bit mode from fast system call</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.¹ With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR. However, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- **External interrupts.** The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.

- **Nonmaskable interrupts (NMIs).** The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, "Interrupt Stack Table," in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

- **General-protection exceptions (#GP).** The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - Confirming that the value of RCX is canonical before executing SYSRET.
  - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - Using the IST mechanism for gate 13 (#GP) in the IDT.

¹ Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.
Operation

IF (CS.L ≠ 1) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
THEN #UD; FI;
IF (CPL ≠ 0) THEN #GP(0); FI;

IF (operand size is 64-bit)
THEN (* Return to 64-Bit Mode *)
   IF (RCX is not canonical) THEN #GP(0);
   RIP ← RCX;
   ELSE (* Return to Compatibility Mode *)
   RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2; (* Clear RF, VM, reserved bits; set bit 2 *)

IF (operand size is 64-bit)
   THEN CS.Selector ← IA32_STAR[63:48]+16;
   ELSE CS.Selector ← IA32_STAR[63:48];
FI:
CS.Selector ← CS.Selector OR 3; (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0; (* Flat segment *)
CS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-Byte limit *)
CS.Type ← 11; (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF (operand size is 64-bit)
   THEN (* Return to 64-Bit Mode *)
      CS.L ← 1; (* 64-bit code segment *)
      CS.D ← 0; (* Required if CS.L = 1 *)
   ELSE (* Return to Compatibility Mode *)
      CS.L ← 0; (* Compatibility mode *)
      CS.D ← 1; (* 32-bit code segment *)
FI;
CS.G ← 1; (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3; (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0; (* Flat segment *)
SS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-Byte limit *)
SS.Type ← 3; (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1; (* 32-bit stack segment*)
SS.G ← 1; (* 4-KByte granularity *)

Flags Affected
All.

Protected Mode Exceptions

#UD The SYSRET instruction is not recognized in protected mode.
Real-Address Mode Exceptions
#UD The SYSRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The SYSRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD The SYSRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions
#UD If IA32_EFER.SCE = 0.
    If the LOCK prefix is used.
#GP(0) If CPL ≠ 0.
    If the return is to 64-bit mode and RCX contains a non-canonical address.
## TEST—Logical Compare

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A8 ib</td>
<td>TEST AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AND imm8 with AL; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>A9 iw</td>
<td>TEST AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AND imm16 with AX; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>A9 id</td>
<td>TEST EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AND imm32 with EAX; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>REX.W + A9 id</td>
<td>TEST RAX, imm32</td>
<td>I</td>
<td>N.E.</td>
<td>Valid</td>
<td>AND imm32 sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>F6 /0 ib</td>
<td>TEST r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>AND imm8 with r/m8; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>REX + F6 /0 ib</td>
<td>TEST r/m8*, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Valid</td>
<td>AND imm8 with r/m8; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>F7 /0 iw</td>
<td>TEST r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>AND imm16 with r/m16; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>F7 /0 id</td>
<td>TEST r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>AND imm32 with r/m32; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>REX.W + F7 /0 id</td>
<td>TEST r/m64, imm32</td>
<td>MI</td>
<td>N.E.</td>
<td>Valid</td>
<td>AND imm32 sign-extended to 64-bits with r/m64; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>84 /r</td>
<td>TEST r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>AND r8 with r/m8; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>REX + 84 /r</td>
<td>TEST r/m8*, r8*</td>
<td>MR</td>
<td>N.E.</td>
<td>Valid</td>
<td>AND r8 with r/m8; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>85 /r</td>
<td>TEST r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>AND r16 with r/m16; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>85 /r</td>
<td>TEST r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>AND r32 with r/m32; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>REX.W + 85 /r</td>
<td>TEST r/m64, r64</td>
<td>MR</td>
<td>N.E.</td>
<td>Valid</td>
<td>AND r64 with r/m64; set SF, ZF, PF according to result.</td>
</tr>
</tbody>
</table>

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.
Operation

TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);

IF TEMP = 0
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI:

PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(* AF is undefined *)

Flags Affected

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
TZCNT — Count the Number of Trailing Zero Bits

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F BC r16, r/m16</td>
<td>A V/V</td>
<td>BMI1</td>
<td>Count the number of trailing zero bits in r/m16, return result in r16.</td>
<td></td>
</tr>
<tr>
<td>F3 0F BC r32, r/m32</td>
<td>A V/V</td>
<td>BMI1</td>
<td>Count the number of trailing zero bits in r/m32, return result in r32.</td>
<td></td>
</tr>
<tr>
<td>F3 REX.W 0F BC r64, r/m64</td>
<td>A V/N.E.</td>
<td>BMI1</td>
<td>Count the number of trailing zero bits in r/m64, return result in r64.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

temp ← 0
DEST ← 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )

        temp ← temp +1
        DEST ← DEST+ 1
OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI

IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI

Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

TZCNT:     unsigned __int32 _tzcnt_u32(unsigned __int32 src);
TZCNT:     unsigned __int64 _tzcnt_u64(unsigned __int64 src);
Protected Mode Exceptions

#GP(0)  For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0)  For an illegal address in the SS segment.

#PF (fault-code)  For a page fault.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)  If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.

#SS(0)  For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions

#GP(0)  If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.

#SS(0)  For an illegal address in the SS segment.

#PF (fault-code)  For a page fault.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)  If the memory address is in a non-canonical form.

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.

#PF (fault-code)  For a page fault.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2E /r UCOMISD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F:WIG 2E /r VUCOMISD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F:W1 2E /r VUCOMISD xmm1, xmm2/m64[sa]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs an unordered compare of the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid numeric exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

(V)UCOMISD (all versions)
RESULT ← UnorderedCompare(DEST[63:0] <> SRC[63:0]) (/* Set EFLAGS */) CASE (RESULT) OF

UNORDERED: ZF, PF, CF ← 111;
GREATER_THAN: ZF, PF, CF ← 000;
LESS_THAN: ZF, PF, CF ← 001;
EQUAL: ZF, PF, CF ← 100;
ESAC;
OF, AF, SF ← 0;
Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISD int _mm_comi_round_sd( __m128d a, __m128d b, int imm, int sae);
VUCOMISD int _mm_ucomieq_sd( __m128d a, __m128d b)
VUCOMISD int _mm_ucomilt_sd( __m128d a, __m128d b)
VUCOMISD int _mm_ucomile_sd( __m128d a, __m128d b)
VUCOMISD int _mm_ucomigt_sd( __m128d a, __m128d b)
VUCOMISD int _mm_ucomige_sd( __m128d a, __m128d b)
VUCOMISD int _mm_ucomineq_sd( __m128d a, __m128d b)

SIMD Floating-Point Exceptions
Invalid (if SNaN operands), Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3; additionally
#UD If VEX.vvvv != 1111B.
EVEX-encoded instructions, see Exceptions Type E3NF.
Comparisone the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

\[(V)UCOMISS (all versions)\]

\[
\text{RESULT } \leftarrow \text{UnorderedCompare}\left(\text{DEST}[31:0] \leftrightarrow \text{SRC}[31:0]\right) \{
(*) \text{Set EFLAGS} \}
\]

\[
\text{CASE (RESULT) OF}
\]

\[
\text{UNORDERED: ZF, PF, CF } \leftarrow 111;
\]

\[
\text{GREATER_THAN: ZF, PF, CF } \leftarrow 000;
\]

\[
\text{LESS_THAN: ZF, PF, CF } \leftarrow 001;
\]

\[
\text{EQUAL: ZF, PF, CF } \leftarrow 100;
\]

\[
\text{ESAC};
\]

\[
\text{OF, AF, SF } \leftarrow 0;\}
\]
Intel C/C++ Compiler Intrinsic Equivalent

UCOMISS  int __mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS  int __mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS  int __mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS  int __mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS  int __mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS  int __mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS  int __mm_ucomineq_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Invalid (if SNaN Operands), Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3; additionally
#UD If VEX.vvv != 111B.
EVEX-encoded instructions, see Exceptions Type E3NF.
**UD—Undefined Instruction**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F FF /r</td>
<td>UD0 (^1) r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Raise invalid opcode exception.</td>
</tr>
<tr>
<td>0F B9 /r</td>
<td>UD1 r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Raise invalid opcode exception.</td>
</tr>
<tr>
<td>0F 0B</td>
<td>UD2</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Raise invalid opcode exception.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Some older processors decode the UD0 instruction without a ModR/M byte. As a result, those processors would deliver an invalid opcode exception instead of a fault on instruction fetch when the instruction with a ModR/M byte (and any implied bytes) would cross a page or segment boundary.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\#UD (* Generates invalid opcode exception *);

**Flags Affected**

None.

**Exceptions (All Operating Modes)**

\#UD Raises an invalid opcode exception in all operating modes.
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 15 /r UNPCKHPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpacks and interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and interleaves double-precision floating-point values from high quadwords of xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and interleaves double-precision floating-point values from high quadwords of ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.W1 15 /r VUNPCKHPD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.W1 15 /r VUNPCKHPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.W1 15 /r VUNPCKHPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpacks and interleaves double precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMrm/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs an interleaved unpack of the high double-precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.
Operation

**VUNPCKHPD (EVEX encoded versions when SRC2 is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL >= 128

\[ \text{TMP\_DEST}[63:0] \leftarrow \text{SRC1}[127:64] \]
\[ \text{TMP\_DEST}[127:64] \leftarrow \text{SRC2}[127:64] \]

FI;

IF VL >= 256

\[ \text{TMP\_DEST}[191:128] \leftarrow \text{SRC1}[255:192] \]
\[ \text{TMP\_DEST}[255:192] \leftarrow \text{SRC2}[255:192] \]

FI;

IF VL >= 512

\[ \text{TMP\_DEST}[319:256] \leftarrow \text{SRC1}[383:320] \]
\[ \text{TMP\_DEST}[383:320] \leftarrow \text{SRC2}[383:320] \]
\[ \text{TMP\_DEST}[447:384] \leftarrow \text{SRC1}[511:448] \]
\[ \text{TMP\_DEST}[511:448] \leftarrow \text{SRC2}[511:448] \]

FI;

FOR \( j \leftarrow 0 \) TO KL-1

\[ i \leftarrow j \times 64 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+63:i] \leftarrow \text{TMP\_DEST}[i+63:i] \)

ELSE

IF *merging-masking* ; merging-masking

THEN *\( \text{DEST}[i+63:i] \) remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\[ \text{DEST}[i+63:i] \leftarrow 0 \]

FI

ENDFOR

\[ \text{DEST}[\text{MAX\_VL}-1:\text{VL}] \leftarrow 0 \]
VUNPCKHPD (EVEX encoded version when SRC2 is memory)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[63:0] ← SRC1[127:64]
  TMP_DEST[127:64] ← TMP_SRC2[127:64]
FI;
IF VL >= 256
FI;
IF VL >= 512
  TMP_DEST[319:256] ← SRC1[383:320]
  TMP_DEST[447:384] ← SRC1[511:448]
  TMP_DEST[511:448] ← TMP_SRC2[511:448]
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VUNPCKHPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[255:192] ← SRC2[255:192]
DEST[MAXVL-1:256] ← 0

VUNPCKHPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[MAXVL-1:128] ← 0

UNPCKHPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKHPD __m512d _mm512_unpackhi_pd(__m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b);
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 15 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F:WIG 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F:WIG 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F:W0 15 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F:W0 15 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F:W0 15 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpacks and Interleaves single-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.
EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPS (EVEX encoded version when SRC2 is a register)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

If \(VL \geq 128\):

\[
\begin{align*}
&\text{TMP}_\text{DEST}[31:0] \leftarrow \text{SRC1}[95:64] \\
&\text{TMP}_\text{DEST}[63:32] \leftarrow \text{SRC2}[95:64] \\
&\text{TMP}_\text{DEST}[95:64] \leftarrow \text{SRC1}[127:96] \\
&\text{TMP}_\text{DEST}[127:96] \leftarrow \text{SRC2}[127:96]
\end{align*}
\]

If:

If \(VL \geq 256\):

\[
\begin{align*}
&\text{TMP}_\text{DEST}[159:128] \leftarrow \text{SRC1}[223:192] \\
&\text{TMP}_\text{DEST}[191:160] \leftarrow \text{SRC2}[223:192] \\
&\text{TMP}_\text{DEST}[223:192] \leftarrow \text{SRC1}[255:224] \\
&\text{TMP}_\text{DEST}[255:224] \leftarrow \text{SRC2}[255:224]
\end{align*}
\]

If:

If \(VL \geq 512\):

\[
\begin{align*}
&\text{TMP}_\text{DEST}[511:384] \leftarrow \text{SRC1}[383:352] \\
&\text{TMP}_\text{DEST}[383:352] \leftarrow \text{SRC2}[383:352] \\
&\text{TMP}_\text{DEST}[319:288] \leftarrow \text{SRC1}[351:320] \\
&\text{TMP}_\text{DEST}[351:320] \leftarrow \text{SRC2}[351:320] \\
&\text{TMP}_\text{DEST}[387:356] \leftarrow \text{SRC1}[479:448] \\
&\text{TMP}_\text{DEST}[415:384] \leftarrow \text{SRC2}[479:448] \\
&\text{TMP}_\text{DEST}[443:412] \leftarrow \text{SRC1}[479:448] \\
&\text{TMP}_\text{DEST}[479:448] \leftarrow \text{SRC2}[479:448] \\
&\text{TMP}_\text{DEST}[511:480] \leftarrow \text{SRC1}[511:480] \\
&\text{TMP}_\text{DEST}[511:480] \leftarrow \text{SRC2}[511:480]
\end{align*}
\]

If:

Figure 4-27. VUNPCKHPS Operation
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+31:i] ← 0
      FI
   FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VUNPCKHPS (EVEX encoded version when SRC2 is memory)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
   i ← j * 32
   IF (EVEX.b = 1)
      THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
      ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
   FI
ENDFOR;
IF VL >= 128
   TMP_DEST[31:0] ← SRC1[95:64]
   TMP_DEST[63:32] ← TMP_SRC2[95:64]
   TMP_DEST[95:64] ← SRC1[127:96]
   FI;
IF VL >= 256
   FI;
IF VL >= 512
   TMP_DEST[287:256] ← SRC1[351:320]
   TMP_DEST[319:288] ← TMP_SRC2[351:320]
   TMP_DEST[415:384] ← SRC1[479:448]
   TMP_DEST[447:416] ← TMP_SRC2[479:448]
   TMP_DEST[479:448] ← SRC1[511:480]
   TMP_DEST[511:480] ← TMP_SRC2[511:480]
   FI;
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+31:i] ← 0
      FI
   FI
FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

**VUNPCKHPS (VEX.256 encoded version)**
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[255:224] ← SRC2[255:224]
DEST[MAXVL-1:128] ← 0

**VUNPCKHPS (VEX.128 encoded version)**
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[MAXVL-1:128] ← 0

**UNPCKHPS (128-bit Legacy SSE version)**
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VUNPCKHPS __m512 _mm512_unpackhi_ps( __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.
### UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 14 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpacks and interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128:66:0F:W1 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and interleaves double-precision floating-point values from low quadwords of xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256:66:0F:W1 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and interleaves double-precision floating-point values from low quadwords of ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>VEX.NDS.512:66:0F:W1 14 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128:66:0F:W1 14 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512:66:0F:W1 14 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpacks and interleaves double precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs an interleaved unpack of the low double-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is an XMM register, conditionally updated using writemask k1.
Operation

VUNPCKLPD (EVEX encoded versions when SRC2 is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← SRC2[63:0]
FI;

IF VL >= 256
FI;

IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[511:448] ← SRC2[447:384]
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0
VUNPCKLPD (EVEX encoded version when SRC2 is memory)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL - 1\)
  \(i \leftarrow j \times 64\)
  IF \((EVEX.b = 1)\)
    THEN \(\text{TMP\_SRC2}[i+63:i] \leftarrow \text{SRC2}[63:0]\)
    ELSE \(\text{TMP\_SRC2}[i+63:i] \leftarrow \text{SRC2}[i+63:i]\)
  \FI;
ENDFOR;

IF \(VL \geq 128\)
  \(\text{TMP\_DEST}[63:0] \leftarrow \text{SRC1}[63:0]\)
  \(\text{TMP\_DEST}[127:64] \leftarrow \text{TMP\_SRC2}[63:0]\)
\FI;

IF \(VL \geq 256\)
  \(\text{TMP\_DEST}[191:128] \leftarrow \text{SRC1}[191:128]\)
  \(\text{TMP\_DEST}[255:192] \leftarrow \text{TMP\_SRC2}[191:128]\)
\FI;

IF \(VL \geq 512\)
  \(\text{TMP\_DEST}[319:256] \leftarrow \text{SRC1}[319:256]\)
  \(\text{TMP\_DEST}[383:320] \leftarrow \text{TMP\_SRC2}[319:256]\)
  \(\text{TMP\_DEST}[447:384] \leftarrow \text{SRC1}[447:384]\)
  \(\text{TMP\_DEST}[511:448] \leftarrow \text{TMP\_SRC2}[447:384]\)
\FI;

FOR \(j \leftarrow 0\) TO \(KL - 1\)
  \(i \leftarrow j \times 64\)
  IF \(k1[j]\) OR *no writemask*
    THEN \(\text{DEST}[i+63:i] \leftarrow \text{TMP\_DEST}[i+63:i]\)
    ELSE
      IF *merging-masking* \(;\) merging-masking
        THEN *DEST[i+63:i] remains unchanged* \(;\)
      ELSE *zeroing-masking* \(;\) zeroing-masking
        \(\text{DEST}[i+63:i] \leftarrow 0\)
      FI
  \FI
ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

**VUNPCKLPD (VEX.256 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]\)
\(\text{DEST}[127:64] \leftarrow \text{SRC2}[63:0]\)
\(\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128]\)
\(\text{DEST}[255:192] \leftarrow \text{SRC2}[191:128]\)
\(\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0\)

**VUNPCKLPD (VEX.128 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]\)
\(\text{DEST}[127:64] \leftarrow \text{SRC2}[63:0]\)
\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0\)

**UNPCKLPD (128-bit Legacy SSE version)**

\(\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0]\)
\(\text{DEST}[127:64] \leftarrow \text{SRC2}[63:0]\)
\(\text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)
**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKLKD _m512d _mm512_unpacklo_pd(_m512d a, _m512d b);
VUNPCKLDP __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLDP __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLDP __m256d _mm256_unpacklo_pd(_m256d a, _m256d b)
VUNPCKLDP __m256d _mm256_mask_unpacklo_pd(_m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLDP __m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
VUNPCKLDP __m128d _mm_unpacklo_pd(__m128d a, __m128d b);
VUNPCKLDP __m128d _mm_mask_unpacklo_pd(_m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLDP __m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.
### UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/</th>
<th>Op</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 14 /r UNPCKLPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Unpacks and interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymmm2,ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 14 /r VUNPCKLPS xmm1[k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 14 /r VUNPCKLPS ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 14 /r VUNPCKLPS zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpacks and interleaves single-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be an XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

IF \(VL \geq 128\)

\[\begin{align*}
\text{TMP\_DEST}[31:0] & \leftarrow \text{SRC1}[31:0] \\
\text{TMP\_DEST}[63:32] & \leftarrow \text{SRC2}[31:0] \\
\text{TMP\_DEST}[95:64] & \leftarrow \text{SRC1}[63:32] \\
\text{TMP\_DEST}[127:96] & \leftarrow \text{SRC2}[63:32]
\end{align*}\]

FI;

IF \(VL \geq 256\)

\[\begin{align*}
\text{TMP\_DEST}[159:128] & \leftarrow \text{SRC1}[159:128] \\
\text{TMP\_DEST}[191:160] & \leftarrow \text{SRC2}[159:128] \\
\text{TMP\_DEST}[223:192] & \leftarrow \text{SRC1}[191:160] \\
\text{TMP\_DEST}[255:224] & \leftarrow \text{SRC2}[191:160]
\end{align*}\]

FI;

IF \(VL \geq 512\)

\[\begin{align*}
\text{TMP\_DEST}[287:256] & \leftarrow \text{SRC1}[287:256] \\
\text{TMP\_DEST}[319:288] & \leftarrow \text{SRC2}[287:256] \\
\text{TMP\_DEST}[351:320] & \leftarrow \text{SRC1}[319:288] \\
\text{TMP\_DEST}[383:352] & \leftarrow \text{SRC2}[319:288] \\
\text{TMP\_DEST}[415:384] & \leftarrow \text{SRC1}[341:384] \\
\text{TMP\_DEST}[447:416] & \leftarrow \text{SRC2}[415:384] \\
\text{TMP\_DEST}[479:448] & \leftarrow \text{SRC1}[447:416] \\
\text{TMP\_DEST}[511:480] & \leftarrow \text{SRC2}[447:416]
\end{align*}\]

FI;

FOR \(j \leftarrow 0 \text{ to } KL-1\)

\(i \leftarrow j \times 32\)
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

IF k1[j] OR "no writemask"
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
ELSE
    IF "merging-masking" ; merging-masking
        THEN "DEST[i+31:i] remains unchanged"
        ELSE "zeroing-masking" ; zeroing-masking
            DEST[i+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VUNPCKLPS (EVEX encoded version when SRC2 is memory)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[31:0]
    TMP_DEST[63:32] ← TMP_SRC2[31:0]
    TMP_DEST[95:64] ← SRC1[63:32]
    FI;
IF VL >= 256
    FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← TMP_SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← TMP_SRC2[447:416]
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR "no writemask"
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF "merging-masking" ; merging-masking
            THEN "DEST[i+31:i] remains unchanged"
            ELSE "zeroing-masking" ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI
    FI;

VUNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values
ENDFOR
DEST[MAXVL-1:VL] ← 0

**UNPCKLPS (VEX.256 encoded version)**
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[255:224] ← SRC2[191:160]
DEST[MAXVL-1:256] ← 0

**VUNPCKLPS (VEX.128 encoded version)**
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[MAXVL-1:128] ← 0

**UNPCKLPS (128-bit Legacy SSE version)**
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VUNPCKLPS __m512 __m512_mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 __m512_mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 __m512_mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 __m256_mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256__mm256_mask_unpacklo_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 __m256_mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 __m128_mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 __m128_mm_mask_unpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 __m128_mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.
5.1 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-1 for the column index value between 0:7, followed by Table 5-2 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a shorthand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one of more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator ‘not’ is expressed using ‘!’. Additionally, the conditional expression “A?B:C” expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g. andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g. andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g. norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g. zeros or norCB.

The 3-input expression “majorABC” returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression “minorABC” returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-1 and Table 5-2 include:

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.
Table 5-1. Low 8 columns of the 16x16 Map of VTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>0H</th>
<th>1H</th>
<th>2H</th>
<th>3H</th>
<th>4H</th>
<th>5H</th>
<th>6H</th>
<th>7H</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00H</td>
<td>FALSE</td>
<td>andAnorBC</td>
<td>norBnandAC</td>
<td>andAIB</td>
<td>norCnandBA</td>
<td>andAIC</td>
<td>norAxorBC</td>
<td>andAnandBC</td>
</tr>
<tr>
<td>01H</td>
<td>norABC</td>
<td>norCB</td>
<td>norBxorAC</td>
<td>A?!B: norBC</td>
<td>norCxorBA</td>
<td>A?!C: norBC</td>
<td>A?!xorBC: norBC</td>
<td>A?!nandBC: norBC</td>
</tr>
<tr>
<td>04H</td>
<td>andBnoraC</td>
<td>norCxorBA</td>
<td>B?norAC: and AC</td>
<td>B?norAC: A</td>
<td>andBIC</td>
<td>norCnorBA</td>
<td>B?!C: andAC</td>
<td>B?!CA</td>
</tr>
<tr>
<td>05H</td>
<td>norCA</td>
<td>norCandBA</td>
<td>B?norAC: xor AC</td>
<td>A?!B: IC</td>
<td>B?!C: norAC</td>
<td>IC</td>
<td>A?!xorBC: IC</td>
<td>A?!nandBC: IC</td>
</tr>
<tr>
<td>06H</td>
<td>norAxnorBC</td>
<td>A?!norBC: xor BC</td>
<td>B?norAC: C</td>
<td>xorBorAC</td>
<td>C?norBA: B</td>
<td>xorCorBA</td>
<td>xorCB</td>
<td>B?!CorAC</td>
</tr>
<tr>
<td>07H</td>
<td>norAandBC</td>
<td>minorABC</td>
<td>C?IB: IA</td>
<td>nandBorAC</td>
<td>B?!C: IA</td>
<td>nandCorBA</td>
<td>A?!xorBC: nand BC</td>
<td>nandCB</td>
</tr>
<tr>
<td>08H</td>
<td>norAnandBC</td>
<td>A?!norBC: and BC</td>
<td>andCxorBA</td>
<td>A?!B: andBC</td>
<td>andBxorAC</td>
<td>A?!C: andBC</td>
<td>A?!xorBC: and BC</td>
<td>xorAandBC</td>
</tr>
<tr>
<td>0AH</td>
<td>andCIA</td>
<td>A?!norBC: C</td>
<td>C?IA: andBA</td>
<td>xorCA</td>
<td>xorCandBA</td>
<td>A?!nandBC: C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0CH</td>
<td>andBIA</td>
<td>A?!norBCB</td>
<td>B?!A: andAC</td>
<td>xorBA</td>
<td>andBnandAC</td>
<td>A?!C: B</td>
<td>xorBandAC</td>
<td>A?!nandBC: B</td>
</tr>
<tr>
<td>0EH</td>
<td>norAnorBC</td>
<td>xorAorBC</td>
<td>B?!A: C</td>
<td>A?!B: orBC</td>
<td>C?IA: B</td>
<td>A?!CorBC</td>
<td>B?!nandAC: C</td>
<td>A?!nandBCor BC</td>
</tr>
<tr>
<td>0FH</td>
<td>IA</td>
<td>nandAorBC</td>
<td>C?nandBA: IA</td>
<td>nandBA</td>
<td>B?!nandAC: IA</td>
<td>nandCA</td>
<td>nandAxnorBC</td>
<td>nandABC</td>
</tr>
</tbody>
</table>

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.
Table 5-2. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>[7:4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3:0]</td>
<td></td>
</tr>
<tr>
<td>00H</td>
<td>08H</td>
</tr>
<tr>
<td>andABC</td>
<td>andAXnorBC</td>
</tr>
<tr>
<td>04H</td>
<td>andBxnorAC</td>
</tr>
<tr>
<td>05H</td>
<td>A?andBC:IC</td>
</tr>
<tr>
<td>08H</td>
<td>andCB</td>
</tr>
<tr>
<td>09H</td>
<td>B?CxorAC</td>
</tr>
<tr>
<td>0AH</td>
<td>A?andBC:CC</td>
</tr>
<tr>
<td>0BH</td>
<td>B?C:IA</td>
</tr>
<tr>
<td>0CH</td>
<td>A?andBC:B</td>
</tr>
<tr>
<td>0DH</td>
<td>C7B:IA</td>
</tr>
<tr>
<td>0EH</td>
<td>A?andBC:orBC</td>
</tr>
<tr>
<td>0FH</td>
<td>nandAnandB C</td>
</tr>
</tbody>
</table>

Table 5-1 and Table 5-2 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

Result_imm8 = Table_Lookup_Entry( 0FH0, 0CCH, 0AAH)

Table_Lookup_Entry is the Boolean expression defined in Table 5-1 and Table 5-2.
5.2 INSTRUCTIONS (V-Z)

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).
Operation

**VALIGND (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (SRC2 *is memory*) (AND EVEX.b = 1)
THEN
    FOR j ← 0 TO KL-1
        i ← j * 32
        src[i+31:i] ← SRC2[31:0]
    ENDFOR;
ELSE src ← SRC2
FI
; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
    THEN SHIFT = imm8[1:0]
ELSE
    IF VL = 256
        THEN SHIFT = imm8[2:0]
    ELSE SHIFT = imm8[3:0]
    FI
FI
FI

tmp[2VL-1:0] ← tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← tmp[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0
VALIGNQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (SRC2 *is memory*) (AND EVEX.b = 1)

THEN

FOR j ← 0 TO KL-1
i ← j * 64
src[i+63:i] ← SRC2[63:0]
ENDFOR;
ELSE src ← SRC2
FI

; Concatenate sources

tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]

; Shift right quadword elements
IF VL = 128
THEN SHIFT = imm8[0]
ELSE
IF VL = 256
THEN SHIFT = imm8[1:0]
ELSE SHIFT = imm8[2:0]
FI
FI;

tmp[2VL-1:0] ← tmp[2VL-1:0] >> (64*SHIFT)

; Apply writemask

FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR *no writemask*
THEN DEST[i+63:i] ← tmp[i+63:i]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] ← 0
FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VALIGND __m512i _mm512_alignr_epi32(__m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32(__m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32(__m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_alignr_epi64(__m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i _mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i _mm256_maskz_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i _mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i _mm_maskz_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);

Exceptions

See Exceptions Type E4NF.
VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 65 /r VBLENDMPD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend double-precision vector xmm2 and double-precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 65 /r VBLENDMPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend double-precision vector ymm2 and double-precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 65 /r VBLENDMPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 65 /r VBLENDMPS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend single-precision vector xmm2 and single-precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 65 /r VBLENDMPS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend single-precision vector ymm2 and single-precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 65 /r VBLENDMPS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.
Operation

**VBLENDMPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC2[i+63:i]
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN DEST[i+63:i] ← SRC1[i+63:i]
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI;
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] ← 0

**VBLENDMPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC2[i+31:i]
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN DEST[i+31:i] ← SRC1[i+31:i]
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
                FI;
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 _mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.
## VBROADCAST—Load with Broadcast Floating-Point Data

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast single-precision floating-point element in mem to four locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast single-precision floating-point element in mem to eight locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast double-precision floating-point element in mem to four locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 18/r VBROADCASTSS xmm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast the low single-precision floating-point element in the source operand to four locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast low double-precision floating-point element in the source operand to four locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 19 /r VBROADCAST ymm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 19 /r VBROADCAST ymm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 (k1)[z], xmm2/m64</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Broadcast two single-precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 (k1)[z], xmm2/m64</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast two single-precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 (k1)[z], xmm2/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Broadcast low single-precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 (k1)[z], xmm2/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Broadcast low single-precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 (k1)[z], xmm2/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 (k1)[z], m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 (k1)[z], m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 (k1)[z], m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1.</td>
</tr>
</tbody>
</table>
**Description**

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF64X4 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Figure 5-1. Vbroadcastss Operation (VEX.256 encoded version)

Figure 5-2. Vbroadcastss Operation (VEX.128-bit version)

Figure 5-3. Vbroadcastsd Operation (VEX.256-bit version)

Figure 5-4. Vbroadcastf128 Operation (VEX.256-bit version)
Operation

**VBROADCASTSS (128 bit version VEX and legacy)**

\( \text{temp} \leftarrow \text{SRC}[31:0] \)
\( \text{DEST}[31:0] \leftarrow \text{temp} \)
\( \text{DEST}[63:32] \leftarrow \text{temp} \)
\( \text{DEST}[95:64] \leftarrow \text{temp} \)
\( \text{DEST}[127:96] \leftarrow \text{temp} \)
\( \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \)

**VBROADCASTSS (VEX.256 encoded version)**

\( \text{temp} \leftarrow \text{SRC}[31:0] \)
\( \text{DEST}[31:0] \leftarrow \text{temp} \)
\( \text{DEST}[63:32] \leftarrow \text{temp} \)
\( \text{DEST}[95:64] \leftarrow \text{temp} \)
\( \text{DEST}[127:96] \leftarrow \text{temp} \)
\( \text{DEST}[159:128] \leftarrow \text{temp} \)
\( \text{DEST}[191:160] \leftarrow \text{temp} \)
\( \text{DEST}[223:192] \leftarrow \text{temp} \)
\( \text{DEST}[255:224] \leftarrow \text{temp} \)
\( \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \)

**VBROADCASTSS (EVEX encoded versions)**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)
\( i \leftarrow j \times 32 \)
\( \text{IF k1}[j] \text{ OR } \text{no writemask} \)
\( \quad \text{THEN } \text{DEST}[i+31:i] \leftarrow \text{SRC}[31:0] \)
\( \quad \text{ELSE} \)
\( \quad \text{IF *merging-masking} \)
\( \quad \quad \text{THEN } *\text{DEST}[i+31:i] \text{ remains unchanged} \)
\( \quad \quad \text{ELSE } \text{zeroing-masking} \)
\( \quad \quad \text{DEST}[i+31:i] \leftarrow 0 \)
\( \quad \text{FI} \)
\( \text{FI}; \)
ENDFOR
\( \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \)
VBROADCASTSD (VEX.256 encoded version)

```plaintext
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTSD (EVEX encoded versions)

```plaintext
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[63:0]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF32x2 (EVEX encoded versions)

```plaintext
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  n ← (j mod 2) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF128 (VEX.256 encoded version)

```plaintext
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAXVL-1:256] ← 0
```
VBROADCASTF32X4 (EVEX encoded versions)
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  n ← (j modulo 4) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VBROADCASTF64X2 (EVEX encoded versions)
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  n ← (j modulo 2) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] = 0
      FI
  FI;
ENDFOR;

VBROADCASTF32X8 (EVEX.U1.512 encoded version)
FOR j ← 0 TO 15
  i ← j * 32
  n ← (j modulo 8) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VBROADCASTF64x4 (EVEX.512 encoded version)
FOR j ← 0 TO 7
    i ← j * 64
    n ← (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
        IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2(__mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m256 _mm256_mask_broadcast_f32x2(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_maskz_broadcast_f32x2(__mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m256 _mm256_mask_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBroadcastF32x4 __m256 _mm256_maskz_broadcast_f32x4(__mmask8 k, __m128 a);
VBROADCASTF64x2 __m512d _mm512d_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512d _mm512d_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512d_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256d_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m256d _mm256d_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256d_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512d_broadcast_f64x4(__m128d a);
VBROADCASTF64x4 __m512d _mm512d_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512d_maskz_broadcast_f64x4(__mmask8 k, __m128d a);
VBROADCASTF64x4 __m256d _mm256d_broadcast_f64x4(__m128d a);
VBROADCASTF64x4 __m256d _mm256d_mask_broadcast_f64x4(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x4 __m256d _mm256d_maskz_broadcast_f64x4(__mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512d_broadcastsd_pd( __m128d a);
VBROADCASTSD __m512d _mm512d_mask_broadcastsd_pd( __m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512d_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256d_broadcastsd_pd( __m128d a);
VBROADCASTSD __m256d _mm256d_mask_broadcastsd_pd( __m256d s, __mmask8 k, __m128d a);
VBroadcastSD __m256d _mm256d_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSS __m512d _mm512d_broadcastss_ps( __m128d a);
VBroadcastSS __m512d _mm512d_mask_broadcastss_ps( __m512d s, __mmask8 k, __m128d a);
VBroadcastSS __m512d _mm512d_maskz_broadcastss_ps( __mmask8 k, __m128d a);
VBROADCASTSS __m256d _mm256d_broadcastss_ps( __m128d a);
VBroadcastSS __m256d _mm256d_mask_broadcastss_ps( __m256d s, __mmask8 k, __m128d a);
VBroadcastSS __m256d _mm256d_maskz_broadcastss_ps( __mmask8 k, __m128d a);
VBROADCASTSS __m128 _mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 _mm_mask_broadcastss_ps(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_maskz_broadcastss_ps( __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);

Exceptions
VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6.
#UD If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.
If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.
VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory

**Opcode/Instruction** | **Op / En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 {k1}{z}, xmm2 | A | V/V | AVX512VL AVX512F | Compress packed double-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 {k1}{z}, ymm2 | A | V/V | AVX512VL AVX512F | Compress packed double-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 {k1}{z}, zmm2 | A | V/V | AVX512F | Compress packed double-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compress (store) up to 8 double-precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

**Memory destination version:** Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

**Register destination version:** If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VCOMPRESSPD (EVEX encoded versions) store form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

\[
\begin{align*}
\text{SIZE} & \leftarrow 64 \\
\text{k} & \leftarrow 0 \\
\text{FOR} j & \leftarrow 0 \text{ TO } KL-1 \\
& \quad i \leftarrow j \times 64 \\
& \quad \text{IF} \ k1[j] \text{ OR } \text{no writemask} \\
& \quad \quad \text{THEN} \\
& \quad \quad \quad \text{DEST}[k+\text{SIZE}-1:k] \leftarrow \text{SRC}[i+63:i] \\
& \quad \quad \quad \text{k} \leftarrow \text{k} + \text{SIZE} \\
& \quad \text{FI;}
\end{align*}
\]

ENDFOR
VCOMPRESSPD (EVEX encoded versions) reg-reg form

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\(SIZE \leftarrow 64\)
\(k \leftarrow 0\)

\(\text{FOR } j \leftarrow 0 \text{ TO } KL-1\)
\(\quad i \leftarrow j \times 64\)
\(\quad \text{IF } k1[j] \text{ OR } *\text{no writemask}*\)
\(\quad \quad \text{THEN}\)
\(\quad \quad \quad \quad \text{DEST}[k+SIZE-1:k] \leftarrow \text{SRC}[i+63:i]\)
\(\quad \quad \quad \quad k \leftarrow k + SIZE\)
\(\quad \text{FI};\)
\(\text{ENDFOR}\)

\(\text{IF } *\text{merging-masking}*\)
\(\quad \text{THEN } \text{DEST}[VL-1:k] \text{ remains unchanged*}\)
\(\quad \text{ELSE } \text{DEST}[VL-1:k] \leftarrow 0\)
\(\text{FI}\)

\(\text{DEST}[MAXVL-1:VL] \leftarrow 0\)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPD _m512d _mm512_mask_compress_pd( _m512d s, _mmask8 k, _m512d a);
VCOMPRESSPD _m512d _mm512_maskz_compress_pd( _mmask8 k, _m512d a);
VCOMPRESSPD void _mm512_mask_compressstoreu_pd( void * d, _mmask8 k, _m512d a);
VCOMPRESSPD _m256d _mm256_mask_compress_pd( _m256d s, _mmask8 k, _m256d a);
VCOMPRESSPD _m256d _mm256_maskz_compress_pd( _mmask8 k, _m256d a);
VCOMPRESSPD void _mm256_mask_compressstoreu_pd( void * d, _mmask8 k, _m256d a);
VCOMPRESSPD _m128d _mm_mask_compress_pd( _m128d s, _mmask8 k, _m128d a);
VCOMPRESSPD _m128d _mm_maskz_compress_pd( _mmask8 k, _m128d a);
VCOMPRESSPD void _mm_mask_compressstoreu_pd( void * d, _mmask8 k, _m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory

## Description
Compress (stores) up to 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

**Memory destination version:** Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

**Register destination version:** If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

## Operation
### VCOMPRESSPS (EVEX encoded versions) store form
(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN
    DEST[k+SIZE-1:k]← SRC[i+31:i]
  k ← k + SIZE
ENDFOR;

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed single-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.</td>
</tr>
<tr>
<td>VCOMPRESSPS xmm1/m128 {k1}{z}, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed single-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.</td>
</tr>
<tr>
<td>VCOMPRESSPS ymm1/m256 {k1}{z}, ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed single-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.</td>
</tr>
<tr>
<td>VCOMPRESSPS zmm1/m512 {k1}{z}, zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:rr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

## Operation

```c
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN
    DEST[k+SIZE-1:k] ← SRC[i+31:i]
  k ← k + SIZE
ENDFOR;
```
VCOMPRESSPS (EVEX encoded versions) reg-reg form

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN
        DEST[k+SIZE-1:k] ← SRC[i+31:i]
        k ← k + SIZE
    FI;
ENDFOR
IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
    ELSE DEST[VL-1:k] ← 0
FI
DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPS __m512 _mm512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 _mm512_maskz_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void _mm512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 _mm256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 _mm256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void _mm256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 _mm_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 _mm_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void _mm_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VCVTPD2QQ—Convert Packed Double-Precision Floating-Point Values to Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7B /r VCVTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ($2^{w-1}$, where w represents the number of bits in the destination format) is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTPD2QQ (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

    SET_RM(EVEX.RC);
ELSE

    SET_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

    i ← j * 64

    IF k1[j] OR *no writemask*

    THEN DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
    ELSE

        IF *merging-masking* ; merging-masking

            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0

        FI

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPD2QQ (EVEX encoded version) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j * 64

    IF k1[j] OR *no writemask*

    THEN

        IF (EVEX.b == 1)

            THEN

                DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])
            ELSE

                DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])

        FI;

    ELSE

        IF *merging-masking* ; merging-masking

            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0

        FI

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2QQ __m512i _mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i _mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i _mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i _mm512_maskz_cvt_roundpd_epi64( __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i _mm256_mask_cvtpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_maskz_cvtpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_cvt_roundpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_mask_cvt_roundpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_maskz_cvt_roundpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_cvtpd_epi64 ( __m256d src)
VCVTPD2QQ __m128i _mm_cvtpd_epi64 ( __m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.
VCVTPD2UDQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W1 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td></td>
<td></td>
<td>AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W1 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td></td>
<td></td>
<td>AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W1 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value 2^w – 1 is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTDP2UDQ (EVEX encoded versions) when src2 operand is a register**

KL, VL = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_RM(EVEX.RC);

ELSE

SET_RM(MXCSR.RM);

Fi;

FOR j ← 0 TO KL-1

i ← j * 32

k ← j * 64

IF k1[j] OR *no writemask*

THEN

DEST[i+31:i] ←

Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*;

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

Fi;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0

**VCVTDP2UDQ (EVEX encoded versions) when src operand is a memory source**

KL, VL = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j * 32

k ← j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])

ELSE

DEST[i+31:i] ←

Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*;

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

Fi;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPD2UDQ __m256i __m512_cvt_roundpd_epu32(__m512d a);
VCVTPD2UDQ __m256i __m512_mask_cvt_roundpd_epu32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i __m512_maskz_cvt_roundpd_epu32(__mmask8 k, __m512d a);
VCVTPD2UDQ __m256i __m512_cvt_roundpd_epu32(__m512d a, int r);
VCVTPD2UDQ __m256i __m512_mask_cvt_roundpd_epu32(__m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i __m256_mask_cvt_roundpd_epu32(__mmask8 k, __m512d a);
VCVTPD2UDQ __m128i __m256_maskz_cvt_roundpd_epu32(__mmask8 k, __m256d a);
VCVTPD2UDQ __m128i __m256_mask_cvt_roundpd_epu32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i __m256_maskz_cvt_roundpd_epu32(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2.
#UD If EVEX.vvvv != 1111B.
VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 79 / r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 79 / r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert fourth packed double-precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 79 / r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTPD2UQQ (EVEX encoded versions) when src operand is a register**

(\(KL, VL\) = (2, 128), (4, 256), (8, 512))

IF (\(VL == 512\)) AND (EVEX.b == 1)

THEN

\[ \text{SET_RM(EVEX.RC);} \]

ELSE

\[ \text{SET_RM(MXCSR.RM);} \]

FI;

\[ \text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]

\[ i \leftarrow j \times 64 \]

IF \(k1[j]\) OR *no writemask*

THEN \[ \text{DEST}[i+63:i] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])} \]

ELSE

IF *merging-masking* ; merging-masking

THEN \[ \text{DEST}[i+63:i] \text{ remains unchanged} \]

ELSE ; zeroing-masking

\[ \text{DEST}[i+63:i] \leftarrow 0 \]

FI;

\[ \text{ENDIF} \]

\[ \text{ENDFOR} \]

\[ \text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0 \]

**VCVTPD2UQQ (EVEX encoded versions) when src operand is a memory source**

(\(KL, VL\) = (2, 128), (4, 256), (8, 512))

\[ \text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]

\[ i \leftarrow j \times 64 \]

IF \(k1[j]\) OR *no writemask*

THEN

IF (EVEX.b == 1)

THEN

\[ \text{DEST}[i+63:i] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])} \]

ELSE

\[ \text{DEST}[i+63:i] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])} \]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN \[ \text{DEST}[i+63:i] \text{ remains unchanged} \]

ELSE ; zeroing-masking

\[ \text{DEST}[i+63:i] \leftarrow 0 \]

FI;

\[ \text{ENDIF} \]

\[ \text{ENDFOR} \]

\[ \text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0 \]
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPD2UQQ __m512i _mm512_cvtpd_epu64(__m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_epu64(__mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvt_roundpd_epu64(__m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64(__mmask8 k, __m256d a);
VCVTPD2UQQ __m128i _mm_mask_cvtpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i _mm_maskz_cvtpd_epu64(__mmask8 k, __m128d a);
VCVTPD2UQQ __m256i _mm256_cvt_roundpd_epu64 (__m256d src)
VCVTPD2UQQ __m128i _mm_cvt_roundpd_epu64 __m128d src)

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2
#UD If EVEX.vvvv != 1111B.
VCVPH2PS—Convert 16-bit FP values to Single-Precision FP values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 {k1}[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}[z], xmm2/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}[z], ymm2/m256 (sae)</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Half Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single-precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).
VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

**Operation**

\[
vCvt\_h2s(SRC1[15:0])\
\{
\text{RETURN Cvt\_Half\_Precision\_To\_Single\_Precision}(SRC1[15:0]);
\}
\]

**VCVTPH2PS (EVEX encoded versions)**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

\(\text{FOR } j \leftarrow 0 \text{ TO } KL-1\)

\(i \leftarrow j \times 32\)

\(k \leftarrow j \times 16\)

\(\text{IF } k1[j] \text{ OR } *\text{no writemask}*\)

\(\text{THEN } \text{DEST}[i+31:i] \leftarrow vCvt\_h2s(SRC[k+15:k])\)

\(\text{ELSE}\)

\(\text{IF } *\text{merging-masking}* ; \text{merging-masking}\)

\(\text{THEN } *\text{DEST}[i+31:i] \text{ remains unchanged}*\)

\(\text{ELSE } ; \text{zeroing-masking}\)

\(\text{DEST}[i+31:i] \leftarrow 0\)

\(\text{FI}\)

\(\text{FI};\)

\(\text{ENDFOR}\)

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

**VCVTPH2PS (VEX.256 encoded version)**

\(\text{DEST}[31:0] \leftarrow vCvt\_h2s(SRC1[15:0])\)

\(\text{DEST}[63:32] \leftarrow vCvt\_h2s(SRC1[31:16])\)

\(\text{DEST}[95:64] \leftarrow vCvt\_h2s(SRC1[47:32])\)

\(\text{DEST}[127:96] \leftarrow vCvt\_h2s(SRC1[63:48])\)

\(\text{DEST}[159:128] \leftarrow vCvt\_h2s(SRC1[79:64])\)

\(\text{DEST}[191:160] \leftarrow vCvt\_h2s(SRC1[95:80])\)

\(\text{DEST}[223:192] \leftarrow vCvt\_h2s(SRC1[111:96])\)

\(\text{DEST}[255:224] \leftarrow vCvt\_h2s(SRC1[127:112])\)

\(\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0\)
VCVTH2PS (VEX.128 encoded version)
DEST[31:0] ← vCvt_h2s(SRC1[15:0]);
DEST[63:32] ← vCvt_h2s(SRC1[31:16]);
DEST[95:64] ← vCvt_h2s(SRC1[47:32]);
DEST[127:96] ← vCvt_h2s(SRC1[63:48]);
DEST[MAXVL-1:128] ← 0

Flags Affected
None

Intel C/C++ Compiler Intrinsic Equivalent
VCVTH2PS __m512 __m512_cvtph_ps(__m256i a);
VCVTH2PS __m512 __m512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTH2PS __m512 __m512_maskz_cvtph_ps(__mmask16 k, __m512 a);
VCVTH2PS __m512 __m512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTH2PS __m512 __m512_maskz_cvt_roundph_ps(__mmask16 k, __m512 a, int sae);
VCVTH2PS __m256 __m256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTH2PS __m256 __m256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTH2PS __m128 __m128_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTH2PS __m128 __m128_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTH2PS __m128 __m128_cvtph_ps (__m128i m1);
VCVTH2PS __m256 __m256_cvtph_ps (__m128i m1)

SIMD Floating-Point Exceptions
Invalid

Other Exceptions
VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);
EVEX-encoded instructions, see Exceptions Type E11.
#UD If VEX.W=1.
#UD If VEX.vvv != 1111B or EVEX.vvv != 1111B.
VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F3A.W0 1D / r ib</td>
<td>A</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 1D / r ib</td>
<td>A</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 1D / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 1D / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1D / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Half Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Convert packed single-precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e., tiny results) are converted to denormal. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

![Figure 5-7. VCVTPS2PH (128-bit Version)](image)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field is listed in Table 5-3.
VVCVTSPS2PH—Convert Single-Precision FP value to 16-bit FP value

INSTRUCTION SET REFERENCE, V-Z
Vol. 2C 5-37

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAXVL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAXVL-1:256/128/64) of the corresponding destination register are zeroed.

Operation

\[
vCvt_s2h(SRC1[31:0])
\]

\[
\begin{align*}
\text{IF } & \text{Imm}[2] = 0 \\
\text{THEN } & \text{using Imm}[1:0] \text{ for rounding control, see Table 5-3} \\
& \text{return } Cvt\_Single\_Precision\_To\_Half\_Precision\_FP\_Imm(SRC1[31:0]); \\
\text{ELSE } & \text{using MXCSR.RC for rounding control} \\
& \text{return } Cvt\_Single\_Precision\_To\_Half\_Precision\_FP\_Mxcsr(SRC1[31:0]); \\
\end{align*}
\]

VCVTPS2PH (EVEX encoded versions) when dest is a register

\( (KL, VL) = (4, 128), (8, 256), (16, 512) \)

\[
\begin{align*}
\text{FOR } j & \leftarrow 0 \text{ TO } KL-1 \\
i & \leftarrow j \times 16 \\
k & \leftarrow j \times 32 \\
\text{IF } k1[j] \text{ OR *no writemask*} \\
\text{THEN } \text{DEST}[i+15:i] & \leftarrow vCvt_s2h(SRC[k+31:k]) \\
\text{ELSE } & \text{merging-masking*} \\
\text{THEN } \text{DEST}[i+15:i] & \text{remains unchanged*} \\
\text{ELSE } & \text{zeroing-masking} \\
& \text{DEST}[i+15:i] \leftarrow 0 \\
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL-1:VL/2}] \leftarrow 0
\]

Table 5-3. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field Name/value</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm[1:0]</td>
<td>RC=00B</td>
<td>Round to nearest even</td>
<td>If Imm[2] = 0</td>
</tr>
<tr>
<td></td>
<td>RC=01B</td>
<td>Round down</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=10B</td>
<td>Round up</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=11B</td>
<td>Truncate</td>
<td></td>
</tr>
<tr>
<td>Imm[2]</td>
<td>MS1=0</td>
<td>Use imm[1:0] for rounding</td>
<td>Ignore MXCSR.RC</td>
</tr>
<tr>
<td></td>
<td>MS1=1</td>
<td>Use MXCSR.RC for rounding</td>
<td></td>
</tr>
<tr>
<td>Imm[7:3]</td>
<td>Ignored</td>
<td>Ignored by processor</td>
<td></td>
</tr>
</tbody>
</table>
VCVTPS2PH (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR \( j \) \( \leftarrow 0 \) TO KL-1
  \( i \leftarrow j \times 16 \)
  \( k \leftarrow j \times 32 \)
  IF \( k1[j] \) OR *no writemask*
    THEN \( \text{DEST}[i+15:i] \leftarrow \text{vCvt_s2h}(	ext{SRC}[k+31:k]) \)
    ELSE
      *\( \text{DEST}[i+15:i] \) remains unchanged* ; merging-masking
  FI;
ENDFOR

VCVTPS2PH (VEX.256 encoded version)
\( \text{DEST}[15:0] \leftarrow \text{vCvt_s2h}(	ext{SRC}[31:0]) \);
\( \text{DEST}[31:16] \leftarrow \text{vCvt_s2h}(	ext{SRC}[63:32]) \);
\( \text{DEST}[47:32] \leftarrow \text{vCvt_s2h}(	ext{SRC}[95:64]) \);
\( \text{DEST}[63:48] \leftarrow \text{vCvt_s2h}(	ext{SRC}[127:96]) \);
\( \text{DEST}[79:64] \leftarrow \text{vCvt_s2h}(	ext{SRC}[159:128]) \);
\( \text{DEST}[95:80] \leftarrow \text{vCvt_s2h}(	ext{SRC}[191:160]) \);
\( \text{DEST}[111:96] \leftarrow \text{vCvt_s2h}(	ext{SRC}[223:192]) \);
\( \text{DEST}[127:112] \leftarrow \text{vCvt_s2h}(	ext{SRC}[255:224]) \);
\( \text{DEST}[\text{MAXVL-1}:128] \leftarrow 0 \)

VCVTPS2PH (VEX.128 encoded version)
\( \text{DEST}[15:0] \leftarrow \text{vCvt_s2h}(	ext{SRC}[31:0]) \);
\( \text{DEST}[31:16] \leftarrow \text{vCvt_s2h}(	ext{SRC}[63:32]) \);
\( \text{DEST}[47:32] \leftarrow \text{vCvt_s2h}(	ext{SRC}[95:64]) \);
\( \text{DEST}[63:48] \leftarrow \text{vCvt_s2h}(	ext{SRC}[127:96]) \);
\( \text{DEST}[\text{MAXVL-1}:64] \leftarrow 0 \)

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VCVTPS2PH __m256i _mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i _mm512_mask_cvtps_ph(__m256i s, __mmask16 k, __m512 a);
VCVTPS2PH __m256i _mm512_maskz_cvtps_ph(__mmask16 k, __m512 a);
VCVTPS2PH __m512i _mm256_cvtps_ph(__m256i m1, const int imm);
VCVTPS2PH __m512i _mm256_maskz_cvtps_ph(__mmask8 k, __m256i m1, const int imm);
```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);
Other Exceptions
VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);
EVEX-encoded instructions, see Exceptions Type E11.
#UD If VEX.W=1.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VCVTPS2UDQ—Convert Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

### Instruction Set

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}</td>
<td>A/V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts sixteen packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTPS2UDQ (EVEX encoded versions) when src operand is a register**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

IF (VL = 512) AND (EVEX.b = 1)

THEN

\[\text{SET}_RM(\text{EVEX.RC});\]

ELSE

\[\text{SET}_RM(\text{MXCSR.RM});\]

FI;

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\(i \leftarrow j \times 32\)

IF \(k1[j] \text{ OR } \text{no writemask}\)

THEN \(\text{DEST}[i+31:i] \leftarrow \text{Convert Single Precision Floating Point To UInteger} (\text{SRC}[i+31:i])\)

ELSE

IF \(*\text{merging-mask}\) ; merging-mask

THEN \(*\text{DEST}[i+31:i] \text{ remains unchanged}\)

ELSE ; zeroing-mask

\(\text{DEST}[i+31:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\)

**VCVTPS2UDQ (EVEX encoded versions) when src operand is a memory source**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\(i \leftarrow j \times 32\)

IF \(k1[j] \text{ OR } \text{no writemask}\)

THEN

IF (EVEX.b = 1)

THEN

\(\text{DEST}[i+31:i] \leftarrow \text{Convert Single Precision Floating Point To UInteger} (\text{SRC}[31:0])\)

ELSE

\(\text{DEST}[i+31:i] \leftarrow \text{Convert Single Precision Floating Point To UInteger} (\text{SRC}[i+31:i])\)

FI;

ELSE

IF \(*\text{merging-mask}\) ; merging-mask

THEN \(*\text{DEST}[i+31:i] \text{ remains unchanged}\)

ELSE ; zeroing-mask

\(\text{DEST}[i+31:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2UDQ __m512i _mm512_cvtps_epu32(__m512 a);
VCVTPS2UDQ __m512i _mm512_mask_cvtps_epu32(__m512 s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_maskz_cvtps_epu32(__mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_cvt_roundps_epu32(__m512 a, int r);
VCVTPS2UDQ __m512i _mm512_mask_cvt_roundps_epu32(__m512 s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_maskz_cvt_roundps_epu32(__mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i _mm256_cvtps_epu32(__m256d a);
VCVTPS2UDQ __m256i _mm256_mask_cvtps_epu32(__m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i _mm256_maskz_cvtps_epu32(__mmask8 k, __m256 a);
VCVTPS2UDQ __m128i _mm128_cvtps_epu32(__m128 a);
VCVTPS2UDQ __m128i _mm128_mask_cvtps_epu32(__m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i _mm128_maskz_cvtps_epu32(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}[z], xmm2/m64/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td></td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}[z], xmm2/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td></td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}[z], ymm2/m256/m32bcst(er)</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td></td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts eight packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where \(w\) represents the number of bits in the destination format, is returned.

The source operand is a YMM/XMM/XMM (low 64-bit) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VCVTPS2QQ (EVEX encoded versions) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
        ELSE
            SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI
    ELSE
        IF *merging-masking*; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VCVTPS2QQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ← Convert_Single_Precision_To_QuadInteger(SRC[31:0])
                    ELSE
                        DEST[i+63:i] ← Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
                    FI;
            ELSE
                IF *merging-masking*; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
            FI
        ELSE
            IF *merging-masking*; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2QQ __m512i _mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvtp_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvtp_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvtp_roundps_epi64( __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i _mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i _mm_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm_maskz_cvtps_epi64( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3
#UD If EVEX.vvvv != 1111B.
VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VCVTPS2UQQ (EVEX encoded versions) when src operand is a register

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\(\text{IF} \ (VL = 512) \ \text{AND} \ (\text{EVEX.b} = 1)\)

\(\text{THEN}\)

\(\text{SET}_RM(\text{EVEX.RC});\)

\(\text{ELSE}\)

\(\text{SET}_RM(\text{MXCSR.RM});\)

\(\text{FI};\)

\(\text{FOR} \ j \leftarrow 0 \ \text{TO} \ KL-1\)

\(\quad i \leftarrow j * 64\)

\(\quad k \leftarrow j * 32\)

\(\text{IF} \ k1[j] \ \text{OR} \ *\text{no writemask}^*\)

\(\quad \text{THEN} \ DEST[i+63:i] \leftarrow\)

\(\quad \text{Convert\_Single\_Precision\_To\_UQuadInteger}(\text{SRC}[k+31:k])\)

\(\text{ELSE}\)

\(\quad \text{IF} \ *\text{merging-masking}^*; \text{merging-masking}\)

\(\quad \text{THEN} \ *\text{DEST}[i+63:i] \text{remains unchanged}^*\)

\(\quad \text{ELSE}; \text{zeroing-masking}\)

\(\quad \text{DEST}[i+63:i] \leftarrow 0\)

\(\text{FI}\)

\(\text{FI};\)

\(\text{ENDFOR}\)

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

VCVTPS2UQQ (EVEX encoded versions) when src operand is a memory source

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\(\text{FOR} \ j \leftarrow 0 \ \text{TO} \ KL-1\)

\(\quad i \leftarrow j * 64\)

\(\quad k \leftarrow j * 32\)

\(\text{IF} \ k1[j] \ \text{OR} \ *\text{no writemask}^*\)

\(\quad \text{THEN}\)

\(\quad \text{IF} \ (\text{EVEX.b} = 1)\)

\(\quad \text{THEN}\)

\(\quad \text{DEST}[i+63:i] \leftarrow\)

\(\quad \text{Convert\_Single\_Precision\_To\_UQuadInteger}(\text{SRC}[31:0])\)

\(\quad \text{ELSE}\)

\(\quad \text{DEST}[i+63:i] \leftarrow\)

\(\quad \text{Convert\_Single\_Precision\_To\_UQuadInteger}(\text{SRC}[k+31:k])\)

\(\quad \text{FI};\)

\(\text{ELSE}\)

\(\quad \text{IF} \ *\text{merging-masking}^*; \text{merging-masking}\)

\(\quad \text{THEN} \ *\text{DEST}[i+63:i] \text{remains unchanged}^*\)

\(\quad \text{ELSE}; \text{zeroing-masking}\)

\(\quad \text{DEST}[i+63:i] \leftarrow 0\)

\(\text{FI}\)

\(\text{ENDFOR}\)

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2UQQ __m512i _mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i _mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i _mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i _mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i _mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i _mm_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i _mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3
#UD If EVEX.vvvv != 1111B.
## VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 [k1][z], xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert two packed quadword integers from xmm2/m128/m64bcst to packed double-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 [k1][z], ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert four packed quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 [k1][z], zmm2/m512/m64bcst(er)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts packed quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTQQ2PD (EVEX2 encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

```
SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
```

FOR j ← 0 TO KL-1

```
i ← j * 64
IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
    FI
FI;
```

ENDFOR

DEST[MAXVL-1:VL] ← 0
VCVTQQ2PD (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
          THEN
            DEST[i+63:i] ← Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
          ELSE
            DEST[i+63:i] ← Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
          FI
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PD _m512d _mm512_cvtepi64_pd(__m512i a);
VCVTQQ2PD _m512d _mm512_mask_cvtepi64_pd(__m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD _m512d _mm512_maskz_cvtepi64_pd(__mmask16 k, __m512i a);
VCVTQQ2PD _m512d _mm512_cvt_roundepi64_pd(__m512i a, int r);
VCVTQQ2PD _m512d _mm512_mask_cvt_roundepi64_pd(__m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD _m256d _mm256_mask_cvt_roundepi64_pd(__m256d s, __mmask8 k, __m256i a);
VCVTQQ2PD _m256d _mm256_cvt_roundepi64_pd(__mmask8 k, __m256i a);
VCVTQQ2PD _m128d _mm_mask_cvt_roundepi64_pd(__mmask8 k, __m128i a);
VCVTQQ2PD _m128d _mm_cvt_roundepi64_pd(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.
VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed quadword integers from xmm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed quadword integers from ymm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W1 5B /r VCVTQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst(ER)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed quadword integers from zmm2/mem to eight packed single-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts packed quadword integers in the source operand (second operand) to packed single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PS (EVEX encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[1] OR *no writemask*
    THEN DEST[k+31:k] ← Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[k+31:k] remains unchanged*
    ELSE ; zeroing-masking
      DEST[k+31:k] ← 0
  FI
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
VCVTQ2PS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b == 1)
            THEN
                DEST[k+31:k] ← Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
            ELSE
                DEST[k+31:k] ← Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[k+31:k] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[k+31:k] ← 0
                FI
            FI;
    END FOR

DEST[MAXVL-1:VL/2] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTQ2PS __m256 __m512_cvtepi64_ps( __m512i a);
VCVTQ2PS __m256 __m512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQ2PS __m256 __m512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQ2PS __m256 __m512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQ2PS __m256 __m512_mask_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQ2PS __m256 __m256_cvtepi64_ps( __m256i a);
VCVTQ2PS __m256 __m256_mask_cvtepi64_ps( __m256 s, __mmask8 k, __m256i a);
VCVTQ2PS __m256 __m256_maskz_cvtepi64_ps( __mmask8 k, __m256i a);
VCVTQ2PS __m256 __m128_cvtepi64_ps( __m128i a);
VCVTQ2PS __m256 __m128_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQ2PS __m256 __m128_maskz_cvtepi64_ps( __mmask8 k, __m128i a);
VCVTQ2PS __m128 __m128_cvtepi64_ps( __m128i a);
VCVTQ2PS __m128 __m128_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQ2PS __m128 __m128_maskz_cvtepi64_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions
Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2
#UD             If EVEX.vvvv != 1111B.
VCVTSD2USI—Convert Scalar Double-Precision Floating-Point Value to Unsigned Doubleword Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F2.0F.w0 79/r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.</td>
</tr>
<tr>
<td>VCVTSD2USI r32, xmm1/m64[er]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| EVEX.LIG.F2.0F.w1 79/r | A      | V/N.E.¹                | AVX512F           | Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.
| VCVTSD2USI r64, xmm1/m64[er] |         |                       |                    |             |

NOTES:
1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

**Operation**

**VCVTSD2USI (EVEX encoded version)**

IF (SRC *is register*) AND (EVEX.b = 1) THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;

IF 64-Bit Mode and OperandSize = 64 THEN
  DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
ELSE
  DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
FI

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvt_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvt_roundsd_u64(__m128d, int r);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F3.0F.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.</td>
</tr>
<tr>
<td>VCVTSS2USI r32, xmm1/m32[er]</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Converts one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.</td>
</tr>
</tbody>
</table>

NOTES:
1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts a single-precision floating-point value in the source operand (the second operand) to an unsigned double-word integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value \(2^w - 1\) is returned, where \(w\) represents the number of bits in the destination format.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.
Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTSS2USI (EVEX encoded version)

IF (SRC *is register*) AND (EVEX.b = 1)  
THEN  
LET_RM(EVEX,RM);  
ELSE  
LET_RM(MXCSR,RM);  
FI;

IF 64-bit Mode and OperandSize = 64  
THEN  
DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);  
ELSE  
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);  
FI;

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned _mm_cvtss_u32(__m128 a);  
VCVTSS2USI unsigned _mm_cvt_roundss_u32(__m128 a, int r);  
VCVTSS2USI unsigned __int64 _mm_cvtss_u64(__m128 a);  
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64(__m128 a, int r);
**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A/V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 using truncation with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7A /r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A/V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7A /r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>A/V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

**EVEX encoded versions**: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\) where \(w\) represents the number of bits in the destination format is returned.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VCVTTPD2QQ (EVEX encoded version) when src operand is a register**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\(\text{FOR} j \leftarrow 0 \text{ TO } KL-1\)

\(i \leftarrow j \times 64\)

\(\text{IF } k1[j] \text{ OR } \text{no writemask}\)

\(\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])}\)

\(\text{ELSE}\)

\(\text{IF } \text{merging-masking}\)

\(\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged}\)

\(\text{ELSE}

\(\text{DEST}[i+63:i] \leftarrow 0\)

\(\text{FI}\)

\(\text{FI}\)

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\)
VCVTPD2QQ (EVEX encoded version) when src operand is a memory source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2QQ __m512i _mm512_cvttpd_epi64( __m512d a);
VCVTPD2QQ __m512i _mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTPD2QQ __m512i _mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTPD2QQ __m512i _mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTPD2QQ __m256i _mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m128i _mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i _mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2.
#UD If EVEX.vvvv != 1111B.
VCVTPD2UDQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W1 78 /r VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W1 78 02 /r VCVTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W1 78 /r VCVTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Operation

VCVTPD2UDQ (EVEX encoded versions) when src2 operand is a register

(KL, VL) = (2, 128), (4, 256),(8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
      FL
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL/2] ← 0

VCVTPD2UDQ (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (2, 128), (4, 256),(8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
      FL
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
      FL
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL/2] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPD2UDQ __m256i _mm512_cvttpd_epi32( __m512d a);
VCVTTPD2UDQ __m256i _mm512_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_maskz_cvttpd_epi32( __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_cvtt_roundpd_epi32(__m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_mask_cvtt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_maskz_cvtt_roundpd_epi32( __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i _mm256_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvttpd_epi32( __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i __m128d_maskz_cvtt_roundpd_epi32( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2.
#UD If EVEX.vvvv != 1111B.
VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 78 /r VCVTTPD2UQQ xmm1 (k1)[z], xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 78 /r VCVTTPD2UQQ ymm1 (k1)[z], ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 78 /r VCVTTPD2UQQ zmm1 (k1)[z], zmm2/m512/m64bcst(5ae)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value \(2^w - 1\) is returned, where \(w\) represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

**VCVTTPD2UQQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR \(j \leftarrow 0\) TO KL-1

\(i \leftarrow j \times 64\)

IF \(k1[]\) OR *no writemask*

THEN DEST[i+63:i] \(\leftarrow\) Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged* ; zeroing-masking

ELSE

DEST[i+63:i] \(\leftarrow 0\)

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] \(\leftarrow 0\)
VCVTTPD2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b == 1)
        THEN
            DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
        ELSE
            DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2UQQ _mm<size>[_mask[z]]_cvtt[round]pd_epu64
VCVTTPD2UQQ __m512i __mm512_cvtppd_epu64(__m512d a);
VCVTTPD2UQQ __m512i __mm512_mask_cvtppd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_maskz_cvtppd_epu64(__mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_cvt_roundpd_epu64(__m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_mask_cvt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_maskz_cvt_roundpd_epu64(__mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i __mm256_cvtppd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i __mm256_mask_cvtppd_epu64(__mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i __mm256_maskz_cvtppd_epu64(__mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i __mm128_cvtppd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __mm128_mask_cvtppd_epu64(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTTPS2UDQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

**Opcode/Instruction**
<table>
<thead>
<tr>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W0 78 /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 78 /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 78 /r</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single precision floating-point values from zmm2/m312/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts with truncation packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**VCVTTPS2UDQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR $j \leftarrow 0$ TO KL-1

\[ i \leftarrow j \times 32 \]

IF k1[$j]$ OR *no writemask* *\
THEN DEST[i+31:i] \leftarrow Convert_Single_Precision_Floating_Point_To_ULInteger_Truncate(SRC[i+31:i])* 
ELSE \[ i \text{ remains unchanged} * 
ELSE \[ i+31:i \leftarrow 0 \]
\]

Fi

ENDFOR

DEST[MAXVL-1:VL] \leftarrow 0
VCVTTPS2UDQ (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] ← Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
        ELSE
            DEST[i+31:i] ← Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPS2UDQ __m512i _mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i _mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i _mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i _mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i _mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2.
#UD If EVEX.vvvv != 1111B.
VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 7A /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7A /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7A /r</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from xmm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where \(w\) represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2QQ (EVEX encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[i] OR *no writemask*
    THEN DEST[i+63:j] ←
      Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:j] ← 0
      FI
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VCVTTPS2QQ (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j × 64
  k ← j × 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b == 1)
      THEN
        DEST[i+63:i] ← Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
      ELSE
        DEST[i+63:i] ← Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
      FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2QQ __m512i _mm512_cvttps_epi64( __m256 a);
VCVTTPS2QQ __m512i _mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i _mm256_maskz_cvttps_epi64( __m512i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i _mm256_cvtt_roundps_epi64( __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_maskz_cvttps_epi64( __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_maskz_cvtt_roundps_epi64( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.
VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F:w0 78 /r VCVTTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:w0 78 /r VCVTTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:w0 78 /r VCVTTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts with truncation up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTTPS2UQQ (EVEX encoded versions) when src operand is a register**

$(KL, VL) = (2, 128), (4, 256), (8, 512)$

FOR $j \leftarrow 0$ TO $KL-1$

$i \leftarrow j \times 64$

$k \leftarrow j \times 32$

IF $k1[j]$ OR *no writemask*

THEN $DEST[i+63:i] \leftarrow$

    Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])

ELSE

    IF *merging-masking* ; merging-masking

    THEN $DEST[i+63:i]$ remains unchanged*

    ELSE ; zeroing-masking

    $DEST[i+63:i] \leftarrow 0$

FI

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$
VCVTTPS2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
   i ← j * 64
   k ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b == 1)
            THEN
                DEST[i+63:i] ← Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
            ELSE
                DEST[i+63:i] ← Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
         FI;
      ELSE
         IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
         ELSE
            zeroing-masking
               DEST[i+63:i] ← 0
         FI
      FI
   ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPS2UQQ _mm<size>[_mask][z]_cvtt[_round]ps_epu64
VCVTTPS2UQQ _mm512i_mm512_cvttps_epu64(__m512 a);
VCVTTPS2UQQ _mm512i_mm512_mask_cvttpps_epu64(__m512l s, __mmask16 k, __m512 a);
VCVTTPS2UQQ _mm512i_mm512_maskz_cvttpps_epu64(__mmask16 k, __m512 a);
VCVTTPS2UQQ _mm512i_mm512_cvtt_roundps_epu64( __m256 a, int sae);
VCVTTPS2UQQ _mm512i_mm512_mask_cvtt_roundps_epu64( __m512l s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UQQ _mm512i_mm512_maskz_cvtt_roundps_epu64( __mmask16 k, __m512 a, int sae);
VCVTTPS2UQQ _mm256i_mm256_cvtt_roundps_epu64( __m256l s, __mmask8 k, __m256 a);
VCVTTPS2UQQ _mm256i_mm256_mask_cvtt_roundps_epu64( __mmask8 k, __m256 a);
VCVTTPS2UQQ _mm128i_mm_mask_cvtt_roundps_epu64( __m128l s, __mmask8 k, __m128 a);
VCVTTPS2UQQ _mm128i_mm_maskz_cvtt_roundps_epu64( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3.
#UD If EVEX.vvvv != 1111B.

5-68 Vol. 2C VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values
VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F2.0F.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.</td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}</td>
<td>A V/N.E.</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts with truncation a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

**Operation**

**VCVTTSD2USI (EVEX encoded version)**

IF 64-Bit Mode and OperandSize = 64

THEN DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);

ELSE DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTSD2USI unsigned int __m_cvttsd_u32(__m128d);
VCVTTSD2USI unsigned int __m_cvttsd_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 __m_cvttsd_u64(__m128d);
VCVTTSD2USI unsigned __int64 __m_cvttsd_roundsd_u64(__m128d, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F3.0F.W0 78 /r VCVTSS2USI r32, xmm1/m32{sae}</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.</td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W1 78 /r VCVTSS2USI r64, xmm1/m32{sae}</td>
<td>A V/N.E.†</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.</td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts with truncation a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

**EVEX.W1 version:** promotes the instruction to produce 64-bit data in 64-bit mode.

**Note:** EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
**Operation**

**VCVTTSS2USI (EVEX encoded version)**

IF 64-bit Mode and OperandSize = 64
THEN
   DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
ELSE
   DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTSS2USI unsigned _mm_cvttss_u32( __m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32( __m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64( __m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64( __m128 a, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F.W0 7A /r VCVTUDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W0 7A /r VCVTUDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W0 7A /r VCVTUDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUDQ2PD (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:j] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:j] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:j] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0


VCVTUDQ2PD (EVEX encoded versions) when src operand is a memory source
(QL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          Convert_UInteger_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTUDQ2PD __m512d _mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d _mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d _mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d _mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d _mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d _mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d _mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E5.
#UD If EVEX.vvvv != 1111B.
VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed unsigned doubleword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VCVTUDQ2PS (EVEX encoded version) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_RM(EVEX.RC);

ELSE

SET_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0
VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PS __m512 _mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 _mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_mask_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 _mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 _mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 _mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 _mm128_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 _mm128_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 _mm128_maskz_cvtepu32_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F:W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F:W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F:W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUQQ2PD (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

If (VL == 512) AND (EVEX.b == 1)

Then

    SET_RM(EVEX.RC);

Else

    SET_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

    i ← j * 64

    IF k1[j] OR *no writemask*

    THEN DEST[i+63:j] ← Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:j])

    ELSE

    IF *merging-masking*;

        merging-masking

    THEN *DEST[i+63:j] remains unchanged*

    ELSE

        zeroing-masking

        DEST[i+63:j] ← 0

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0
VCVTUQQ2PD (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:j] ← Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:j] ← Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:j])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:j] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:j] ← 0
                FI
            FI
    ENDFOR

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTUQQ2PD __m512d _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm128_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d _mm128_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm128_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2.

#UD IF EVEX.vvvv != 1111B.
VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.0F.W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F.W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F.W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed unsigned quadword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

**Note:** EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VCVTUQQ2PS (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_RM(EVEX.RC);

ELSE

SET_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j * 32

k ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] ← Convert_UQuadInteger_To_Single_Precision_Floating_Point[SRC[k+63:k]]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

ENDFOR

DEST[MAXVL-1:VL/2] ← 0
VCVTUQQ2PS (EVEX encoded version) when src operand is a memory source

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 32\)

\(k \leftarrow j \times 64\)

IF \(k1[j]\) OR "no writemask"

THEN

IF (EVEX.b = 1)

THEN

\(\text{DEST}[i+31:i] \leftarrow \text{Convert}_{U\text{QuadInteger}}_{\to\text{Single-Precision_Floating-Point}}(\text{SRC}[63:0])\)

ELSE

\(\text{DEST}[i+31:i] \leftarrow \text{Convert}_{U\text{QuadInteger}}_{\to\text{Single-Precision_Floating-Point}}(\text{SRC}[k+63:k])\)

FI;

ELSE

IF "merging-masking"

THEN "DEST[i+31:i] remains unchanged"

ELSE "zeroing-masking"

\(\text{DEST}[i+31:i] \leftarrow 0\)

FI

ENDIF

\(\text{DEST}[\text{MAXVL-1:VL/2}] \leftarrow 0\)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTUQQ2PS __m256 _mm512_cvtepu64_ps(__m512i a);

VCVTUQQ2PS __m256 _mm512_mask_cvtepu64_ps(__m256 s, __mmask8 k, __m512i a);

VCVTUQQ2PS __m256 _mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a);

VCVTUQQ2PS __m256 _mm512_cvt_roundepu64_ps(__m512i a, int r);

VCVTUQQ2PS __m128 _mm256_cvtepu64_ps(__m256i a);

VCVTUQQ2PS __m128 _mm256_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m256i a);

VCVTUQQ2PS __m128 _mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W0 7B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one unsigned doubleword integer from r/m32 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W1 7B /r</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
<td>Convert one unsigned quadword integer from r/m64 to one double-precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

### Notes:
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.
EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

### Operation

**VCVTUSI2SD (EVEX encoded version)**

IF (SRC2 *is register*) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
IF 64-Bit Mode And OperandSize = 64
  THEN
    DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
  ELSE
    DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTUSI2SD __m128d _mm_cvtu32_sd(__m128d s, unsigned a);
VCVTUSI2SD __m128d _mm_cvtu64_sd(__m128d s, unsigned __int64 a);
VCVTUSI2SD __m128d __m_cvt_roundu64_sd(__m128d s, unsigned __int64 a, int r);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

See Exceptions Type E3NF if W1, else type E10NF.
VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32[er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F:W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64[er]</td>
<td>A</td>
<td>V/N/E,1</td>
<td>AVX512F</td>
<td>Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

NOTES:
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

Operation

VCVTUSI2SS (EVEX encoded version)

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);
    FI;
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS __m128 _mm_cvtu32_ss( __m128 s, unsigned a);
VCVTUSI2SS __m128 _mm_cvt_roundu32_ss( __m128 s, unsigned a, int r);
VCVTUSI2SS __m128 _mm_cvtu64_ss( __m128 s, unsigned __int64 a);
VCVTUSI2SS __m128 _mm_cvt_roundu64_ss( __m128 s, unsigned __int64 a, int r);
SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type E3NF.
VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 42 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 42 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 42 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.wwvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each superblock of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see “Tmp1” in Figure 5-8, is constructed from the second source operand and the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.

- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2 and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.

- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.
Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW
Operation

**VDBPSADbw (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Selection of quadruplets:

FOR I = 0 to VL step 128
   TMP1[I+31:I] ← select (SRC2[I+127: I], imm8[1:0])
   TMP1[I+63:I+32] ← select (SRC2[I+127: I], imm8[3:2])
   TMP1[I+95:I+64] ← select (SRC2[I+127: I], imm8[5:4])
   TMP1[I+127:I+96] ← select (SRC2[I+127: I], imm8[7:6])
END FOR

SAD of quadruplets:

FOR I =0 to VL step 64
   TMP_DEST[I+15:I] ← \(\text{ABS}(\text{SRC1}[I+7: I] - \text{TMP1}[I+7: I]) +\)
      \(\text{ABS}(\text{SRC1}[I+15: I+8]) +\)
      \(\text{ABS}(\text{SRC1}[I+23: I+16]) +\)
      \(\text{ABS}(\text{SRC1}[I+31: I+24])\)

   TMP_DEST[I+31: I+16] ← \(\text{ABS}(\text{SRC1}[I+7: I] - \text{TMP1}[I+15: I+8]) +\)
      \(\text{ABS}(\text{SRC1}[I+15: I+8]) +\)
      \(\text{ABS}(\text{SRC1}[I+23: I+16]) +\)
      \(\text{ABS}(\text{SRC1}[I+31: I+24])\)

   TMP_DEST[I+47: I+32] ← \(\text{ABS}(\text{SRC1}[I+39: I+32]) - \text{TMP1}[I+23: I+16]) +\)
      \(\text{ABS}(\text{SRC1}[I+47: I+40]) +\)
      \(\text{ABS}(\text{SRC1}[I+55: I+48]) +\)
      \(\text{ABS}(\text{SRC1}[I+63: I+56])\)

   TMP_DEST[I+63: I+48] ← \(\text{ABS}(\text{SRC1}[I+39: I+32]) - \text{TMP1}[I+31: I+24]) +\)
      \(\text{ABS}(\text{SRC1}[I+47: I+40]) - \text{TMP1}[I+31: I+24]) +\)
      \(\text{ABS}(\text{SRC1}[I+55: I+48]) - \text{TMP1}[I+47: I+40]) +\)
      \(\text{ABS}(\text{SRC1}[I+63: I+56]) - \text{TMP1}[I+55: I+48])\)

ENDFOR

FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
   ELSE
      IF *merging-masking*
         THEN *DEST[i+15:i] remains unchanged*
      ELSE
         DEST[i+15:i] ← 0
      FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.nb.
VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Op/En Tuple Type Operand 1 Operand 2 Operand 3 Operand 4
A Tuple1 Scalar ModRM:reg (w) ModRM:r/m (r) NA NA

Description

Expand (load) up to 8/4/2, contiguous, double-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VEXPANDPD (EVEX encoded versions)

(\(KL, VL\)) = (2, 128), (4, 256), (8, 512)

\(k \leftarrow 0\)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 64\)

IF \(k1[j]\) OR *no writemask*

THEN

\(\text{DEST}[i+63:i] \leftarrow \text{SRC}[k+63:k];\)

\(k \leftarrow k + 64\)

ELSE

IF *merging-masking* ; merging-masking

THEN *\(\text{DEST}[i+63:i]\) remains unchanged* ; zeroing-masking

ELSE

\(\text{DEST}[i+63:i] \leftarrow 0\)

FI

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent
VEXPANDPD __m512d __m512_mask_expand_pd(__m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d __m512_maskz_expand_pd(__mmask8 k, __m512d a);
VEXPANDPD __m512d __m512_maskz_expandloadu_pd(__m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d __m512_maskz_expandloadu_pd(__mmask8 k, void * a);
VEXPANDPD __m256d __m256_mask Expand_pd(__m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d __m256_maskz_expand_pd(__mmask8 k, __m256d a);
VEXPANDPD __m256d __m256_maskz_expandloadu_pd(__m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d __m256_maskz_expandloadu_pd(__mmask8 k, void * a);
VEXPANDPD __m128d __m128_mask Expand_pd(__m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d __m128_maskz_expand_pd(__mmask8 k, __m128d a);
VEXPANDPD __m128d __m128_maskz_expandloadu_pd(__m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d __m128_maskz_expandloadu_pd(__mmask8 k, void * a);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E4.nb.
#UD If EVEX.vvvv != 111B.
VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}{z}, xmm2/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}{z}, ymm2/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}{z}, zmm2/m512</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Expand (load) up to 16/8/4, contiguous, single-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPS (EVEX encoded versions)

KL, VL = (4, 128), (8, 256), (16, 512)

k ← 0

FOR j ← 0 TO KL-1

i ← j * 32

IF k1[j] OR *no writemask*

THEN

DEST[i+31:i] ← SRC[k+31:k];

k ← k + 32

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*  

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

ENDFOR

DEST[MAXVL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VEXPANDPS __m512 _mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 _mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 _mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 _mm256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 _mm_maskz_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_maskz_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 _mm_maskz_expandloadu_ps( __mmask8 k, void * a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VERR/VERW—Verify a Segment for Reading or Writing

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /4</td>
<td>VERR r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set ZF=1 if segment specified with r/m16 can be read.</td>
</tr>
<tr>
<td>0F 00 /5</td>
<td>VERW r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set ZF=1 if segment specified with r/m16 can be written.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment’s DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector’s RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector’s value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

**Operation**

IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
    THEN ZF ← 0; Fl;

Read segment descriptor;

IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
    THEN
        ZF ← 0;
    ELSE
        IF ((Instruction = VERR) and (Segment readable))
or ((Instruction = VERW) and (Segment writable))
            THEN
                ZF ← 1;
            Fl;
        Fl;
Flags Affected
The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions
The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

Real-Address Mode Exceptions

- **#UD** The VERR and VERW instructions are not recognized in real-address mode.
- If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- **#UD** The VERR and VERW instructions are not recognized in virtual-8086 mode.
- If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.
VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 19 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 19 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Extract 128 bits of packed single-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 19 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 128 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 19 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Extract 128 bits of packed double-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 19 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 128 bits of packed double-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 19 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m256 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1B /r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of packed double-precision floating-point values from zmm2 and store results in xmm1/m256 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single-precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double-precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 6 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Operation

**VEXTRACTF32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512
IF VL = 256
CASEx (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
Fi;
IF VL = 512
CASEx (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
Fi;
FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*
        THEN DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:128] ← 0

**VEXTRACTF32x4 (EVEX encoded versions) when destination is memory**

VL = 256, 512
IF VL = 256
CASEx (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
Fi;
IF VL = 512
CASEx (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
Fi;
FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
VEXTRACTF64x2 (EVEX encoded versions) when destination is a register
VL = 256, 512
IF VL = 256
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
FI;
IF VL = 512
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
FI;
FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:128] ← 0

VEXTRACTF64x2 (EVEX encoded versions) when destination is memory
VL = 256, 512
IF VL = 256
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
FI;
IF VL = 512
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
FI;
FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  FI;
ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
FI;
ENDFOR

VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged* 
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
    FI
  FI;
ENDFOR
DEST[MAXVL-1:256] ← 0

VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is memory
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged* 
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
    FI
  FI;
ENDFOR

VEXTRACTF64x4 (EVEX.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged* 
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
    FI
  FI;
ENDFOR

VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values
VEXTRACTF128 / VEXTRACTF32x4 / VEXTRACTF64x4 — Extract Packed Floating-Point Values

**VEXTRACTF64x4 (EVEX.512 encoded version) when destination is memory**

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE ; merging-masking
      *DEST[i+63:i] remains unchanged*
    FI;
ENDFOR

**VEXTRACTF128 (memory destination form)**

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

**VEXTRACTF128 (register destination form)**

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXTRACTF32x4 __m128 __m512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 __m512_mask_extractf32x4_ps(__m512 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 __m512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m256 __m512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m256 __m512_mask_extractf32x4_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m256 __m512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 __m512_extractf32x8_ps(__m512 a, const int nidx);
VEXTRACTF32x8 __m256 __m512_mask_extractf32x8_ps(__m512 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 __m512_maskz_extractf32x8_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x2 __m128d __m512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d __m512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d __m512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d __m512_extractf64x2_pd(__m256d a, const int nidx);
VEXTRACTF64x2 __m128d __m512_mask_extractf64x2_pd(__m256d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d __m512_maskz_extractf64x2_pd(__mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d __m512_extractf64x4_pd(__m512d a, const int nidx);
VEXTRACTF64x4 __m256d __m512_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d __m512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d __m512_extractf64x4_pd(__m128d a, const int nidx);
VEXTRACTF64x4 __m256d __m512_mask_extractf64x4_pd(__m128d s, __mmask8 k, __m128d a, const int nidx);
VEXTRACTF64x4 __m256d __m512_maskz_extractf64x4_pd(__mmask8 k, __m128d a, const int nidx);
VEXTRACTF128 __m128 __m256_extractf128_ps(__m256 a, int offset);
VEXTRACTF128 __m128d_mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6NF.
#UD IF VEX.L = 0.
#UD IF VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 xmm1/m128, ymm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 39 /r ib VEXTRACTI32X4 xmm1/m128 {k1}{z}, ymm2, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 39 /r ib VEXTRACTI32x4 xmm1/m128 {k1}{z}, zmm2, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, ymm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, zmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3B /r ib VEXTRACTI32X8 ymm1/m256 {k1}{z}, zmm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3B /r ib VEXTRACTI64x4 ymm1/m256 {k1}{z}, zmm2, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at a 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.
VEXTRACTI64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation**

**VEXTRACTI32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.

FI;

FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
  FI
FI

ENDFOR

DEST[MAXVL-1:128] ← 0

**VEXTRACTI32x4 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VEXTRACTI64x2 (EVEX encoded versions) when destination is a register

VL = 256, 512
IF VL = 256
  CASE (imm8[0]) OF
    0:TMP_DEST[127:0] ← SRC1[127:0]
    1:TMP_DEST[127:0] ← SRC1[255:128]
  ESAC.
FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00:TMP_DEST[127:0] ← SRC1[127:0]
    01:TMP_DEST[127:0] ← SRC1[255:128]
    10:TMP_DEST[127:0] ← SRC1[383:256]
    11:TMP_DEST[127:0] ← SRC1[511:384]
  ESAC.
FI;
FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:128] ← 0
VEXTRACTI64x2 (EVEX encoded versions) when destination is memory
VL = 256, 512
IF VL = 256
   CASE (imm8[0]) OF
      0: TMP_DEST[127:0]  SRC1[127:0]
      1: TMP_DEST[127:0]  SRC1[255:128]
   ESAC.
   Ft;
IF VL = 512
   CASE (imm8[1:0]) OF
      00: TMP_DEST[127:0]  SRC1[127:0]
      01: TMP_DEST[127:0]  SRC1[255:128]
      10: TMP_DEST[127:0]  SRC1[383:256]
      11: TMP_DEST[127:0]  SRC1[511:384]
   ESAC.
   Ft;
FOR j  0 TO 1
   i  j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i]  TMP_DEST[i+63:i]
      ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
   FI;
ENDFOR
VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
   0: TMP_DEST[255:0]  SRC1[255:0]
   1: TMP_DEST[255:0]  SRC1[511:256]
ESAC.
FOR j  0 TO 7
   i  j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i]  TMP_DEST[i+31:i]
      ELSE *merging-masking* ; merging-masking
         IF *DEST[i+31:i] remains unchanged*
            THEN DEST[i+31:i]  0
         ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i]  0
         FI
      FI;
ENDFOR
DEST[MAXVL-1:256]  0
VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register

VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR

DEST[MAXVL-1:256] ← 0

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
VEXTRACTI128 (memory destination form)
CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

VEXTRACTI128 (register destination form)
CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m512i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_extracti32x4_epi32(__m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_extracti32x8_epi32(__m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_mask_extracti32x8_epi32(__m512i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm256_extracti32x8_epi32(__m256i a, const int nidx);
VEXTRACTI32x8 __m256i _mm256_mask_extracti32x8_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x8 __m256i _mm256_maskz_extracti32x8_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, int offset);
VEXTRACTI64x4 __m256i _mm512_extracti64x4_epi64(__m512i a, int offset);
SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.
#UD IF VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VFIXUPIMMPD—Fix Up Special Packed Float64 Values

### Instruction Set Reference, V-Z

**Op/ Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 54 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 54 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 54 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

| Op/En Tuple Type Operand 1 Operand 2 Operand 3 Operand 4 |
|----------------|----------------|---------------|---------------|---------------|
| A Full ModRM:reg (r, w) EVEX.vvvv ModRM:rrr/m (r) Imm8 |

**Description**

Perform fix-up of quad-word elements encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.
Operation

enum TOKEN_TYPE
{
    QNAN_TOKEN <= 0,
    SNAN_TOKEN <= 1,
    ZERO_VALUE_TOKEN <= 2,
    POS_ONE_VALUE_TOKEN <= 3,
    NEG_INF_TOKEN <= 4,
    POS_INF_TOKEN <= 5,
    NEG_VALUE_TOKEN <= 6,
    POS_VALUE_TOKEN <= 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0]) {
    tsrc[63:0] <= ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j <= 0;
        SNAN_TOKEN: j <= 1;
        ZERO_VALUE_TOKEN: j <= 2;
        POS_ONE_VALUE_TOKEN: j <= 3;
        NEG_INF_TOKEN: j <= 4;
        POS_INF_TOKEN: j <= 5;
        NEG_VALUE_TOKEN: j <= 6;
        POS_VALUE_TOKEN: j <= 7;
    } ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] <= dest[63:0]; ; preserve content of DEST
        0001: dest[63:0] <= tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] <= QNaN(tsrc[63:0]);
        0011: dest[63:0] <= QNaN_Indefinite;
        0100: dest[63:0] <= -INF;
        0101: dest[63:0] <= +INF;
        0111: dest[63:0] <= -0;
        1000: dest[63:0] <= +0;
        1001: dest[63:0] <= -1;
        1010: dest[63:0] <= +1;
        1011: dest[63:0] <= ½;
        1100: dest[63:0] <= 90.0;
        1101: dest[63:0] <= PI/2;
        1110: dest[63:0] <= MAX_FLOAT;
        1111: dest[63:0] <= -MAX_FLOAT;
    } ; end of token_response CASE
; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[63:0];
ENDFOR
VFIXUPIMMPD
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← FIXUPIMMPD(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8[7:0])
                ELSE
                    DEST[i+63:i] ← FIXUPIMMPD(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8[7:0])
            FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE DEST[i+63:i] ← 0 ; zeroing-masking
                FI
            FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
Immediate Control Description:

![Diagram showing control bits and their effects on various floating-point values.]

**Figure 5-9. VFIXUPIMMPD Immediate Control Description**

**Intel C/C++ Compiler Intrinsic Equivalent**

- VFIXUPIMMPD _m512d _mm512_fixupimm_PD(_m512d a, _m512i tbl, int imm);
- VFIXUPIMMPD _m512d _mm512_mask_fixupimm_PD(_m512d s, _mmask8 k, _m512d a, _m512i tbl, int imm);
- VFIXUPIMMPD _m512d _mm512_maskz_fixupimm_PD(_mmask8 k, _m512d a, _m512i tbl, int imm);
- VFIXUPIMMPD _m512d _mm512_fixupimm_round_PD(_m512d a, _m512i tbl, int imm, int sae);
- VFIXUPIMMPD _m512d _mm512_mask_fixupimm_round_PD(_m512d s, _mmask8 k, _m512d a, _m512i tbl, int imm, int sae);
- VFIXUPIMMPD _m512d _mm512_maskz_fixupimm_round_PD(_mmask8 k, _m512d a, _m512i tbl, int imm, int sae);
- VFIXUPIMMPD _m256d _mm256_fixupimm_PD(_m256d a, _m256i tbl, int imm);
- VFIXUPIMMPD _m256d _mm256_mask_fixupimm_PD(_m256d s, _mmask8 k, _m256d a, _m256i tbl, int imm);
- VFIXUPIMMPD _m256d _mm256_maskz_fixupimm_PD(_mmask8 k, _m256d a, _m256i tbl, int imm);
- VFIXUPIMMPD _m128d _mm128_fixupimm_PD(_m128d a, _m128i tbl, int imm);
- VFIXUPIMMPD _m128d _mm128_mask_fixupimm_PD(_m128d s, _mmask8 k, _m128d a, _m128i tbl, int imm);
- VFIXUPIMMPD _m128d _mm128_maskz_fixupimm_PD(_mmask8 k, _m128d a, _m128i tbl, int imm);

**SIMD Floating-Point Exceptions**

- Zero, Invalid

**Other Exceptions**

- See Exceptions Type E2.
VFIXUPIMMPS—Fix Up Special Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 54 /r</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 54 /r</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 54 /r</td>
<td>A/V/V</td>
<td>AVX512F</td>
<td>Fix up elements of float32 vector in zmm2 using int32 vector in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

Perform fix-up of doubleword elements encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPS can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.
Operation

enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0], tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    } ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
        0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] ← QNaN(tsrc[31:0]);
        0011: dest[31:0] ← QNaN_Indefinite;
        0100: dest[31:0] ← -INF;
        0101: dest[31:0] ← +INF;
        0110: dest[31:0] ← tsrc.sign? -INF : +INF;
        0111: dest[31:0] ← -0;
        1000: dest[31:0] ← +0;
        1001: dest[31:0] ← -1;
        1010: dest[31:0] ← +1;
        1011: dest[31:0] ← ½;
        1100: dest[31:0] ← 90.0;
        1101: dest[31:0] ← Pi/2;
        1110: dest[31:0] ← MAX_FLOAT;
        1111: dest[31:0] ← -MAX_FLOAT;
    } ; end of token_response CASE
; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()

VFIXUPIMMPS (EVEX)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN
      DEST[i+31:i] ← FIXUPIMM_SP(DIST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
    ELSE
      DEST[i+31:i] ← FIXUPIMM_SP(DIST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8 [7:0])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] ← 0 ; zeroing-masking
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
Immediate Control Description:

![Image of immediate control description](image)

**Figure 5-10. VFIXUPIMMPS Immediate Control Description**

### Intel C/C++ Compiler Intrinsic Equivalent

```c
VFIXUPIMMPS __m512 __mm512_fixupimm_ps( __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_ps( __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_fixupimm_round_ps( __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_round_ps( __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m256 __mm256_fixupimm_ps( __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_mask_fixupimm_ps(__m256 s, __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_maskz_fixupimm_ps( __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 __mm128_fixupimm_ps( __m128 a, __mmask8 k, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm128_mask_fixupimm_ps(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm128_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128i tbl, int imm);
```

### SIMD Floating-Point Exceptions

- Zero, Invalid

### Other Exceptions

See Exceptions Type E2.
VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

**Description**

Perform a fix-up of the low quadword element encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set). MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

**Operation**

```c
enum TOKEN_TYPE
{
    QNAN_TOKEN        <  0,
    SNAN_TOKEN        <  1,
    ZERO_VALUE_TOKEN  <  2,
    POS_ONE_VALUE_TOKEN <  3,
    NEG_INF_TOKEN    <  4,
    POS_INF_TOKEN    <  5,
    NEG_VALUE_TOKEN  <  6,
    POS_VALUE_TOKEN  <  7
}
```

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W1 /r ib VFIXUPIMMSD xmm1 [k1][z], xmm2, xmm3/m64 [saq], imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>
FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8[7:0])

\[
\text{tsrc}[63:0] \leftarrow ((\text{src1}[62:52] = 0) \text{ AND } (\text{MXCSR.DAZ} = 1)) ? 0.0 : \text{src1}[63:0]
\]

\begin{align*}
\text{CASE}(\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE}) & \{ \\
\text{QNAN_TOKEN: } j \leftarrow 0; & \\
\text{SNAN_TOKEN: } j \leftarrow 1; & \\
\text{ZERO_VALUE_TOKEN: } j \leftarrow 2; & \\
\text{POS_ONE_VALUE_TOKEN: } j \leftarrow 3; & \\
\text{NEG_INF_TOKEN: } j \leftarrow 4; & \\
\text{POS_INF_TOKEN: } j \leftarrow 5; & \\
\text{NEG_VALUE_TOKEN: } j \leftarrow 6; & \\
\text{POS_VALUE_TOKEN: } j \leftarrow 7; & \\
\} & \text{; end source special CASE(tsrc...)}
\end{align*}

; The required response from src3 table is extracted
\[
\text{token_response}[3:0] = \text{tbl3}[3+4*\text{\text{4}}*];
\]

\begin{align*}
\text{CASE}(\text{token_response}[3:0]) & \{ \\
0000: & \text{dest}[63:0] \leftarrow \text{dest}[63:0] \text{; preserve content of DEST} \\
0001: & \text{dest}[63:0] \leftarrow \text{tsrc}[63:0]; \text{; pass through src1 normal input value, denormal as zero} \\
0010: & \text{dest}[63:0] \leftarrow \text{QNaN}([\text{tsrc}[63:0]]);
0011: & \text{dest}[63:0] \leftarrow \text{QNaN}_\text{Indefinite}; \\
0100: & \text{dest}[63:0] \leftarrow -\text{INF}; \\
0101: & \text{dest}[63:0] \leftarrow +\text{INF}; \\
0110: & \text{dest}[63:0] \leftarrow \text{tsrc}.\text{sign} ? -\text{INF} : +\text{INF}; \\
0111: & \text{dest}[63:0] \leftarrow 0; \\
1000: & \text{dest}[63:0] \leftarrow +0; \\
1001: & \text{dest}[63:0] \leftarrow -1; \\
1010: & \text{dest}[63:0] \leftarrow +1; \\
1011: & \text{dest}[63:0] \leftarrow \frac{1}{2}; \\
1100: & \text{dest}[63:0] \leftarrow 90.0; \\
1101: & \text{dest}[63:0] \leftarrow \text{PI/2}; \\
1110: & \text{dest}[63:0] \leftarrow \text{MAX_FLOAT}; \\
1111: & \text{dest}[63:0] \leftarrow -\text{MAX_FLOAT}; \\
\} & \text{; end of token_response CASE}
\end{align*}

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
\begin{align*}
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: ZERO_VALUE_TOKEN}) \text{ AND } \text{imm8}[0] \text{ then set } \#\text{ZE}; \\
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: ZERO_VALUE_TOKEN}) \text{ AND } \text{imm8}[1] \text{ then set } \#\text{IE}; \\
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: ONE_VALUE_TOKEN}) \text{ AND } \text{imm8}[2] \text{ then set } \#\text{ZE}; \\
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: ONE_VALUE_TOKEN}) \text{ AND } \text{imm8}[3] \text{ then set } \#\text{IE}; \\
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: SNAN_TOKEN}) \text{ AND } \text{imm8}[4] \text{ then set } \#\text{IE}; \\
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: NEG_INF_TOKEN}) \text{ AND } \text{imm8}[5] \text{ then set } \#\text{IE}; \\
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: NEG_VALUE_TOKEN}) \text{ AND } \text{imm8}[6] \text{ then set } \#\text{IE}; \\
\text{IF } (\text{tsrc}[63:0] \text{ of } \text{TOKEN_TYPE: POS_INF_TOKEN}) \text{ AND } \text{imm8}[7] \text{ then set } \#\text{IE}; \\
\} & \text{; end fault reporting}
\end{align*}

return dest[63:0];

\} & \text{; end of FIXUPIMM_DP()}

VFIXUPIMMSD (EVEX encoded version)

IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
        ELSE DEST[63:0] ← 0 ; zeroing-masking
    FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Immediate Control Description:

![Figure 5-11. VFIXUPIMMSD Immediate Control Description](image)

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSD __m128d _mm_fixupimm_sd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_fixupimm_round_sd(__m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E3.
VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 /r ib VFIXUPIMMSS xmm1 (k1)[z], xmm2, xmm3/m32{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fix up nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set). MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```c
enum TOKEN_TYPE
{
    QNAN_TOKEN  <-  0,
    SNAN_TOKEN  <-  1,
    ZERO_VALUE_TOKEN  <-  2,
    POS_ONE_VALUE_TOKEN  <-  3,
    NEG_INF_TOKEN  <-  4,
    POS_INF_TOKEN  <-  5,
    NEG_VALUE_TOKEN  <-  6,
    POS_VALUE_TOKEN  <-  7
}
```
FIXUPIMM_SP (dest[31:0], src1[31:0], tbl3[31:0], imm8[7:0]) {
    tsrc[31:0] <- ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j <- 0;
        SNAN_TOKEN: j <- 1;
        ZERO_VALUE_TOKEN: j <- 2;
        POS_ONE_VALUE_TOKEN: j <- 3;
        NEG_INF_TOKEN: j <- 4;
        POS_INF_TOKEN: j <- 5;
        NEG_VALUE_TOKEN: j <- 6;
        POS_VALUE_TOKEN: j <- 7;
    } ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] <- dest[31:0]; ; preserve content of DEST
        0001: dest[31:0] <- tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] <- QNaN(tsrc[31:0]);
        0011: dest[31:0] <- QNAN_Indefinite;
        0100: dest[31:0] <- -INF;
        0101: dest[31:0] <- +INF;
        0111: dest[31:0] <- -0;
        1000: dest[31:0] <- +0;
        1001: dest[31:0] <- -1;
        1010: dest[31:0] <- +1;
        1011: dest[31:0] <- ½;
        1100: dest[31:0] <- 90.0;
        1101: dest[31:0] <- Pi/2;
        1110: dest[31:0] <- MAX_FLOAT;
        1111: dest[31:0] <- -MAX_FLOAT;
    } ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;

    ; end fault reporting
    return dest[31:0];
} ; end of FIXUPIMM_SP()
VFIXUPIMMSS (EVEX encoded version)

IF k1[0] OR *no writemask*
   THEN DEST[31:0] ← FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[31:0] remains unchanged*
         ELSE DEST[31:0] ← 0 ; zeroing-masking
      FI
   FI;
DEST[MAXVL-1:128] ← 0

Immediate Control Description:

![Figure 5-12. VFIXUPIMMSS Immediate Control Description](image)

**Intel C/C++ Compiler Intrinsic Equivalent**

VFIXUPIMMSS _m128 _mm_fixupimm_ss( _m128 a, _m128i tbl, int imm);
VFIXUPIMMSS _m128 _mm_mask_fixupimm_ss(_m128 s, _mmask8 k, _m128 a, _m128i tbl, int imm);
VFIXUPIMMSS _m128 _mm_maskz_fixupimm_ss( _mmask8 k, _m128 a, _m128i tbl, int imm);
VFIXUPIMMSS _m128 _mm_fixupimm_round_ss(_m128 a, _m128i tbl, int imm, int sae);
VFIXUPIMMSS _m128 _mm_mask_fixupimm_round_ss(_m128 s, _mmask8 k, _m128 a, _m128i tbl, int imm, int sae);
VFIXUPIMMSS _m128 _mm_maskz_fixupimm_round_ss( _mmask8 k, _m128 a, _m128i tbl, int imm, int sae);

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.
### VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst {er}</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst {er}</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst {er}</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>
Description
Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

\[\text{VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)}\]

If (VEX.128) THEN
\[\text{MAXNUM } \leftarrow 2\]
ELSEIF (VEX.256)
\[\text{MAXNUM } \leftarrow 4\]
FI

For \(i = 0\) to \(\text{MAXNUM}-1\) {
\[n \leftarrow 64*i;\]
\[\text{DEST}[n+63:n] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[n+63:n]*\text{SRC3}[n+63:n] + \text{SRC2}[n+63:n])\]
}

If (VEX.128) THEN
\[\text{DEST}[^{\text{MAXVL}-1:128}] \leftarrow 0\]
ELSEIF (VEX.256)
\[\text{DEST}[^{\text{MAXVL}-1:256}] \leftarrow 0\]
FI

\[\text{VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)}\]

If (VEX.128) THEN
\[\text{MAXNUM } \leftarrow 2\]
ELSEIF (VEX.256)
\[\text{MAXNUM } \leftarrow 4\]
FI

For \(i = 0\) to \(\text{MAXNUM}-1\) {
\[n \leftarrow 64*i;\]
\[\text{DEST}[n+63:n] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[n+63:n]*\text{DEST}[n+63:n] + \text{SRC3}[n+63:n])\]
}

If (VEX.128) THEN
\[\text{DEST}[^{\text{MAXVL}-1:128}] \leftarrow 0\]
ELSEIF (VEX.256)
\[\text{DEST}[^{\text{MAXVL}-1:256}] \leftarrow 0\]
FI

\[\text{VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)}\]

If (VEX.128) THEN
\[\text{MAXNUM } \leftarrow 2\]
ELSEIF (VEX.256)
\[\text{MAXNUM } \leftarrow 4\]
FI

For \(i = 0\) to \(\text{MAXNUM}-1\) {
\[n \leftarrow 64*i;\]
\[\text{DEST}[n+63:n] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[n+63:n]*\text{SRC3}[n+63:n] + \text{DEST}[n+63:n])\]
}

If (VEX.128) THEN
\[\text{DEST}[^{\text{MAXVL}-1:128}] \leftarrow 0\]
ELSEIF (VEX.256)
\[\text{DEST}[^{\text{MAXVL}-1:256}] \leftarrow 0\]
FI
VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI;
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ← RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ← RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI;
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_RM(EVEX.RM);
ELSE
  SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63] ← 
          RoundFPControl(SRC2[i+63]*DEST[i+63] + SRC3[i+63])
    ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[i+63] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63] ← 0
          FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63] ← 
            RoundFPControl_MXCSR(SRC2[i+63]*DEST[i+63] + SRC3[63:0])
        ELSE
          DEST[i+63] ← 
            RoundFPControl_MXCSR(SRC2[i+63]*DEST[i+63] + SRC3[i+63])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+63] ← 0
          FI
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j \leftarrow 0 \text{ TO KL-1}
    i \leftarrow j \times 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] \leftarrow
            RoundFPControl(SRC2[i+63:i] \times SRC3[i+63:i] + DEST[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] \leftarrow 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j \leftarrow 0 \text{ TO KL-1}
    i \leftarrow j \times 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] \leftarrow
                        RoundFPControl_MXCSR(SRC2[i+63:i] \times SRC3[63:0] + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] \leftarrow
                        RoundFPControl_MXCSR(SRC2[i+63:i] \times SRC3[i+63:i] + DEST[i+63:i])
                FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] \leftarrow 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| VEX.NDS.128.66.0F38.W0 98 /r  
VFMADD132PS xmm1, xmm2, xmm3/m128  
VEX.NDS.128.66.0F38.W0 A8 /r  
VFMADD213PS xmm1, xmm2, xmm3/m128  
VEX.NDS.128.66.0F38.W0 B8 /r  
VFMADD231PS xmm1, xmm2, xmm3/m128  | A V/V | FMA |  | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1. |
| VEX.NDS.256.66.0F38.W0 98 /r  
VFMADD132PS ymm1, ymm2, ymm3/m256  
VEX.NDS.256.66.0F38.W0 A8 /r  
VFMADD213PS ymm1, ymm2, ymm3/m256  
VEX.NDS.256.66.0F38.W0 B8 /r  
VFMADD231PS ymm1, ymm2, ymm3/m256  | A V/V | FMA |  | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1. |
| EVEX.NDS.128.66.0F38.W0 98 /r  
VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst  
EVEX.NDS.128.66.0F38.W0 A8 /r  
VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst  
EVEX.NDS.128.66.0F38.W0 B8 /r  
VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst  | B V/V | AVX512VL AVX512F |  | Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W0 98 /r  
VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst  
EVEX.NDS.256.66.0F38.W0 A8 /r  
VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst  
EVEX.NDS.256.66.0F38.W0 B8 /r  
VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst  | B V/V | AVX512VL AVX512F |  | Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W0 98 /r  
VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst(er)  
EVEX.NDS.512.66.0F38.W0 A8 /r  
VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst(er)  
EVEX.NDS.512.66.0F38.W0 B8 /r  
VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst(er)  | B V/V | AVX512F |  | Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1. |
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
**Operation**

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3**

If (VEX.128) THEN
   MAXNUM = 4
Elseif (VEX.256)
   MAXNUM = 8
Fi

For i = 0 to MAXNUM-1 {
   n = 32*i;
   DEST[n+31:n] = RoundFPControl_MXCSR(Dest[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}

If (VEX.128) THEN
   DEST[MAXVL-1:128] = 0
Elseif (VEX.256)
   DEST[MAXVL-1:256] = 0
Fi

**VFMADD213PS DEST, SRC2, SRC3**

If (VEX.128) THEN
   MAXNUM = 4
Elseif (VEX.256)
   MAXNUM = 8
Fi

For i = 0 to MAXNUM-1 {
   n = 32*i;
   DEST[n+31:n] = RoundFPControl_MXCSR(SRC2[n+31:n]*Dest[n+31:n] + SRC3[n+31:n])
}

If (VEX.128) THEN
   DEST[MAXVL-1:128] = 0
Elseif (VEX.256)
   DEST[MAXVL-1:256] = 0
Fi

**VFMADD231PS DEST, SRC2, SRC3**

If (VEX.128) THEN
   MAXNUM = 4
Elseif (VEX.256)
   MAXNUM = 8
Fi

For i = 0 to MAXNUM-1 {
   n = 32*i;
   DEST[n+31:n] = RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + Dest[n+31:n])
}

If (VEX.128) THEN
   DEST[MAXVL-1:128] = 0
Elseif (VEX.256)
   DEST[MAXVL-1:256] = 0
Fi
VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
   SET_RM(EVEX.RC);
ELSE
   SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ←
              RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+31:i] ← 0
         FI
      FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1)
            THEN
               DEST[i+31:i] ←
               RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
            ELSE
               DEST[i+31:i] ←
               RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
         FI;
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+31:i] ← 0
         FI
      FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] ←
    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
      THEN
        DEST[i+31:i] ←
          RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
      ELSE
        DEST[i+31:i] ←
          RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI
  ENDIF
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
        RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            ELSE
                DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 __mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 __mm256_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 __mm128_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m256 __mm256_fmadd_ps(__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
### VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1 (k1){z}, xmm2, xmm3/m64{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 (k1){z}, xmm2, xmm3/m64{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 (k1){z}, xmm2, xmm3/m64{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

#### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD multiply-add computation on the low double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

**VFMADD132SD:** Multiplies the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point values in the second source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD213SD:** Multiplies the low double-precision floating-point value from the second source operand to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD231SD:** Multiplies the low double-precision floating-point value from the second source to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

**EVEX encoded version:** The low quadword element of the destination is updated according to the writemask.
Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN
      DEST[63:0] ← RoundFPControl(DEST[63:0] * SRC3[63:0] + SRC2[63:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[63:0] remains unchanged*
      ELSE ; zeroing-masking
         THEN DEST[63:0] ← 0
      FI;
   FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

**VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN
      DEST[63:0] ← RoundFPControl(SRC2[63:0] * DEST[63:0] + SRC3[63:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[63:0] remains unchanged*
      ELSE ; zeroing-masking
         THEN DEST[63:0] ← 0
      FI;
   FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
                ELSE ; zeroing-masking
                    THEN DEST[63:0] ← 0
                FI;
    FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← MAXVL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0

VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0

VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDxxxSD __m128d _mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1, xmm2, xmm3/m32</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1, xmm2, xmm3/m32</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1, xmm2, xmm3/m32</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:rr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD multiply-add computation on single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

VFMADD132SS: Multiplies the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvv/ EVEX.vvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “*“ and “+“ symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN DEST[31:0] \(\leftarrow\) RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
         THEN DEST[31:0] \(\leftarrow\) 0
      FI;
   FI;
DEST[MAXVL-1:128] \(\leftarrow\) 0

**VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN DEST[31:0] \(\leftarrow\) RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
         THEN DEST[31:0] \(\leftarrow\) 0
      FI;
   FI;
DEST[MAXVL-1:128] \(\leftarrow\) 0
VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

IF k1[0] or *no writemask*
    THEN
        DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;

DEST[MAXVL-1:128] ← 0

VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] ← RoundFPControl_MXCSR(Dest[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[MAXVL-1:128] ← 0

VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*Dest[31:0] + SRC3[31:0])
DEST[MAXVL-1:128] ← 0

VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + Dest[31:0])
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __m128_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 __m128_mask2_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 __m128_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 __m128_mask4_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __m128_mask5_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __m128_mask6_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __m128_mask7_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, __m128 a, __m128 b, __m128 c, int r);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
### VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB231PD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB231PD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB231PD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB231PD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMADDSUB132PD zmm1 {k1}{z} zmm2, zmm3/m512/m64bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VFMADDSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VFMADDSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.
VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PD** DEST, SRC2, SRC3

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
```

FI

**VFMADDSUB213PD** DEST, SRC2, SRC3

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
```

FI

**VFMADDSUB231PD** DEST, SRC2, SRC3

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
```

FI
VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j is even
                THEN DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR
DEST[MAXVL-1:VL] ← 0
**VFMADDSUB132PD** DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(\(KL, VL\) = (2, 128), (4, 256), (8, 512))

FOR \(j \leftarrow 0\) TO \(KL - 1\)
  \(i \leftarrow j \times 64\)
  IF \(k1[j] \text{ OR *no writemask*} \)
      THEN
        IF \(j \text{ *is even*} \)
            THEN
              IF \(\text{EVEX.b = 1} \)
                  THEN
                    \(\text{DEST}[i+63:i]\) \leftarrow
                    \text{RoundFPControl_MXCSR(DEST}[i+63:i]*SRC3[63:0] - SRC2[i+63:i])}
                  ELSE
                    \(\text{DEST}[i+63:i]\) \leftarrow
                    \text{RoundFPControl_MXCSR(DEST}[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])}
            FI;
            ELSE
              IF \(\text{EVEX.b = 1} \)
                  THEN
                    \(\text{DEST}[i+63:i]\) \leftarrow
                    \text{RoundFPControl_MXCSR(DEST}[i+63:i]*SRC3[63:0] + SRC2[i+63:i])}
                  ELSE
                    \(\text{DEST}[i+63:i]\) \leftarrow
                    \text{RoundFPControl_MXCSR(DEST}[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])}
              FI;
              ELSE
                \(\text{DEST}[i+63:i]\) \leftarrow 0
              FI
        FI
      ELSE
        IF \(*\text{merging-masking*} \)
          THEN \(*\text{DEST}[i+63:i] \text{ remains unchanged*} \)
          ELSE \(*\text{zeroing-masking*} \)
            \(\text{DEST}[i+63:i]\) \leftarrow 0
          FI
      FI
  Fi
ENDFOR
\(\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\)
VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(SRC2[i+63:i] * DEST[i+63:i] - SRC3[63:0])
            ELSE
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(SRC2[i+63:i] * DEST[i+63:i] - SRC3[i+63:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i] * DEST[i+63:i] + SRC3[63:0])
              ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i] * DEST[i+63:i] + SRC3[i+63:i])
            FI;
          ELSE
            IF *merging-masking* ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
              DEST[i+63:i] ← 0
            FI
        FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN
      ENDIF
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN DEST[i+63:i] ←
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        ELSE DEST[i+63:i] ←
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
      FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(\(KL, VL\) = (2, 128), (4, 256), (8, 512))

\(\text{FOR } j \leftarrow 0 \text{ TO } KL-1\)  
\(i \leftarrow j \times 64\)
\(\text{IF } k1[j] \text{ OR } \text{no writemask}\)  
\(\text{THEN}\)
\(\text{IF } j \text{ is even}\)  
\(\text{THEN}\)
\(\text{IF } (\text{EVEX.b} = 1)\)  
\(\text{THEN}\)
\(\text{DEST}[i+63:i] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[i+63:i] \times \text{SRC3}[63:0] - \text{DEST}[i+63:i])\)
\(\text{ELSE}\)
\(\text{DEST}[i+63:i] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[i+63:i] \times \text{SRC3}[i+63:i] - \text{DEST}[i+63:i])\)
\(\text{FI};\)
\(\text{ELSE}\)
\(\text{IF } (\text{EVEX.b} = 1)\)  
\(\text{THEN}\)
\(\text{DEST}[i+63:i] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[i+63:i] \times \text{SRC3}[63:0] + \text{DEST}[i+63:i])\)
\(\text{ELSE}\)
\(\text{DEST}[i+63:i] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[i+63:i] \times \text{SRC3}[i+63:i] + \text{DEST}[i+63:i])\)
\(\text{FI};\)
\(\text{ELSE}\)
\(\text{IF } \text{merging-masking}\)  
\(\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged}\)
\(\text{ELSE}\)
\(\text{DEST}[i+63:i] \leftarrow 0\)
\(\text{FI}\)
\(\text{ENDFOR}\)
\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPD __m512d __m512d_fmadsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d__m512_d_fmadsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d__m512_mask_fmadsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d__m512_maskz_fmadsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d__m512_mask3_fmadsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d__m512_mask_fmadsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d__m512_maskz_fmadsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d__m512_mask3_fmadsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d__m256d_mask_fmadsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d__m256d_maskz_fmadsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d__m256d_mask3_fmadsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d__m128d_mask_fmadsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d__m128d_maskz_fmadsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d__m128d_mask3_fmadsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m256d__m256d_fmadsub_pd(__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
## VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB213PS ymm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB213PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 96 /r VFMADDSUB132PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 B6 /r VFMADDSUB213PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

**VFMADDSUB132PS**: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB213PS**: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB231PS**: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions**: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version**: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version**: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANSs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
Operation
In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
   n ← 64*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI

**VFMADDSUB213PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
   n ← 64*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI
VFMADDSUB231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
  MAXNUM ← 2
ELSEIF (VEX.256)
  MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
  n ← 64*i;
  DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] ← 0
FI

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) (4, 128), (8, 256),= (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN DEST[i+31:i] ← RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE DEST[i+31:i] ← RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
      FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
      FI
    ENDFOR
  DEST[MAXVL-1:VL] ← 0
VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                            ELSE
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                                FI;
                        ELSE
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                            ELSE
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                                FI;
                        FI;
                    ELSE
                        IF *merging-masking* ; merging-masking
                            THEN *DEST[i+31:i] remains unchanged*
                        ELSE ; zeroing-masking
                            DEST[i+31:i] ← 0
                        FI
                    FI
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE
            DEST[i+31:i] ← 0
        FI
    ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    DEST[i+31:i] ← RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                ELSE
                    DEST[i+31:i] ← RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                                DEST[i+31:i] ← RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                        ELSE
                                DEST[i+31:i] ← RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                        FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                                DEST[i+31:i] ← RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                        ELSE
                                DEST[i+31:i] ← RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                        FI;
        ELSE
            FI;
ENDFOR

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision
Vol. 2C 5-155
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
   DEST[i+31:i] ← 0
FI

IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
   DEST[i+31:i] ← 0
FI

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF j *is even*
            THEN DEST[i+31:i] ←
               RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            ELSE DEST[i+31:i] ←
               RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+31:i] ← 0
            FI
      FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

KL, VL = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
            ELSE
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI;
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged* ; zeroing-masking
              DEST[i+31:i] ← 0
            FI
        FI
    ELSE
      ENDFOR
  DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFMADDSUBxxxPS __m512 __mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 __mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m256 __mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __mm128_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __mm128_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __mm128_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __mm128_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256 __mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DS.128.66.0F38.W1 97 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DS.128.66.0F38.W1 A7 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DS.128.66.0F38.W1 B7 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DS.256.66.0F38.W1 97 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DS.256.66.0F38.W1 A7 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DS.256.66.0F38.W1 B7 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DS.128.66.0F38.W1 97 /r</td>
<td>B/V/V</td>
<td>AVX512VL/AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DS.128.66.0F38.W1 A7 /r</td>
<td>B/V/V</td>
<td>AVX512VL/AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DS.128.66.0F38.W1 B7 /r</td>
<td>B/V/V</td>
<td>AVX512VL/AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DS.256.66.0F38.W1 97 /r</td>
<td>B/V/V</td>
<td>AVX512VL/AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DS.256.66.0F38.W1 A7 /r</td>
<td>B/V/V</td>
<td>AVX512VL/AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DS.256.66.0F38.W1 B7 /r</td>
<td>B/V/V</td>
<td>AVX512VL/AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
VFMSUBADD132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.
VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

IF (VEX.128) THEN

\[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[63:0] \times \text{SRC3}[63:0] + \text{SRC2}[63:0]) \]

\[ \text{DEST}[127:64] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[127:64] \times \text{SRC3}[127:64] - \text{SRC2}[127:64]) \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

ELSEIF (VEX.256)

\[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[63:0] \times \text{SRC3}[63:0] + \text{SRC2}[63:0]) \]

\[ \text{DEST}[127:64] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[127:64] \times \text{SRC3}[127:64] - \text{SRC2}[127:64]) \]

\[ \text{DEST}[191:128] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[191:128] \times \text{SRC3}[191:128] + \text{SRC2}[191:128]) \]

\[ \text{DEST}[255:192] \leftarrow \text{RoundFPControl\_MXCSR}(\text{DEST}[255:192] \times \text{SRC3}[255:192] - \text{SRC2}[255:192]) \]

FI

VFMSUBADD213PD DEST, SRC2, SRC3

IF (VEX.128) THEN

\[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0] \times \text{DEST}[63:0] + \text{SRC3}[63:0]) \]

\[ \text{DEST}[127:64] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[127:64] \times \text{DEST}[127:64] - \text{SRC3}[127:64]) \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

ELSEIF (VEX.256)

\[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0] \times \text{SRC3}[63:0] + \text{DEST}[63:0]) \]

\[ \text{DEST}[127:64] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[127:64] \times \text{SRC3}[127:64] - \text{DEST}[127:64]) \]

\[ \text{DEST}[191:128] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[191:128] \times \text{SRC3}[191:128] + \text{DEST}[191:128]) \]

\[ \text{DEST}[255:192] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[255:192] \times \text{SRC3}[255:192] - \text{DEST}[255:192]) \]

FI

VFMSUBADD231PD DEST, SRC2, SRC3

IF (VEX.128) THEN

\[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0] \times \text{SRC3}[63:0] + \text{DEST}[63:0]) \]

\[ \text{DEST}[127:64] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[127:64] \times \text{SRC3}[127:64] - \text{DEST}[127:64]) \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

ELSEIF (VEX.256)

\[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[63:0] \times \text{SRC3}[63:0] + \text{DEST}[63:0]) \]

\[ \text{DEST}[127:64] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[127:64] \times \text{SRC3}[127:64] - \text{DEST}[127:64]) \]

\[ \text{DEST}[191:128] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[191:128] \times \text{SRC3}[191:128] + \text{DEST}[191:128]) \]

\[ \text{DEST}[255:192] \leftarrow \text{RoundFPControl\_MXCSR}(\text{SRC2}[255:192] \times \text{SRC3}[255:192] - \text{DEST}[255:192]) \]

FI
VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN DEST[i+63:j] ←
            RoundFPControl(DEST[i+63:j]*SRC3[i+63:j] + SRC2[i+63:j])
        ELSE DEST[i+63:j] ←
            RoundFPControl(DEST[i+63:j]*SRC3[i+63:j] - SRC2[i+63:j])
      FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:j] ← 0
      FI
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
          FI
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:j] ←
                    RoundFPControl(SRC2[i+63:j]*DEST[i+63:j] + SRC3[i+63:j])
                ELSE DEST[i+63:j] ←
                    RoundFPControl(SRC2[i+63:j]*DEST[i+63:j] - SRC3[i+63:j])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:j] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:j] ← 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                            ELSE
                                DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                                FI;
                            ELSE
                                IF (EVEX.b = 1)
                                    THEN
                                        DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                                        ELSE
                                            DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                                            FI;
                                    ELSE
                                        FI
                                ELSE
                                    IF *merging-masking* ; merging-masking
                                        THEN *DEST[i+63:i] remains unchanged*
                                        ELSE ; zeroing-masking
                                            DEST[i+63:i] ← 0
                                        FI
                                    FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:j] ←
                    RoundFPControl(SRC2[i+63:j]*SRC3[i+63:j] + DEST[i+63:j])
                ELSE DEST[i+63:j] ←
                    RoundFPControl(SRC2[i+63:j]*SRC3[i+63:j] - DEST[i+63:j])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:j] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:j] ← 0
            FI
        FI
    ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                    FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
        FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBADDxxxPD __m512d __mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d __mm256_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm128_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm128_mask_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm128_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm128_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm128_fmsubadd_round_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm128_mask_fmsubadd_round_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBADDxxxPD __m128d __mm128_maskz_fmsubadd_round_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBADDxxxPD __m128d __mm128_mask3_fmsubadd_round_pd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 97 /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 A7 /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 B7 /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 97 /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 A7 /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 B7 /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 97 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 A7 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 B7 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 97 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 A7 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 B7 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 97 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 A7 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 B7 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
**Operation**

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM \(\leftarrow\) 2
ELSEIF (VEX.256)
   MAXNUM \(\leftarrow\) 4
FI

For \(i = 0\) to MAXNUM -1{
   \(n \leftarrow 64*i;\)
   \(\text{DEST}[n+31:n] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[n+31:n]\times\text{SRC3}[n+31:n] + \text{SRC2}[n+31:n])\)
   \(\text{DEST}[n+63:n+32] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[n+63:n+32]\times\text{SRC3}[n+63:n+32] -\text{SRC2}[n+63:n+32])\)
}

IF (VEX.128) THEN
   \(\text{DEST}[\text{MAXVL-1:128}] \leftarrow 0\)
ELSEIF (VEX.256)
   \(\text{DEST}[\text{MAXVL-1:256}] \leftarrow 0\)
FI

**VFMSUBADD213PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM \(\leftarrow\) 2
ELSEIF (VEX.256)
   MAXNUM \(\leftarrow\) 4
FI

For \(i = 0\) to MAXNUM -1{
   \(n \leftarrow 64*i;\)
   \(\text{DEST}[n+31:n] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[n+31:n]\times\text{DEST}[n+31:n] + \text{SRC3}[n+31:n])\)
   \(\text{DEST}[n+63:n+32] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[n+63:n+32]\times\text{DEST}[n+63:n+32] -\text{SRC3}[n+63:n+32])\)
}

IF (VEX.128) THEN
   \(\text{DEST}[\text{MAXVL-1:128}] \leftarrow 0\)
ELSEIF (VEX.256)
   \(\text{DEST}[\text{MAXVL-1:256}] \leftarrow 0\)
FI

**VFMSUBADD231PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM \(\leftarrow\) 2
ELSEIF (VEX.256)
   MAXNUM \(\leftarrow\) 4
FI

For \(i = 0\) to MAXNUM -1{
   \(n \leftarrow 64*i;\)
   \(\text{DEST}[n+31:n] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[n+31:n]\times\text{SRC3}[n+31:n] + \text{DEST}[n+31:n])\)
   \(\text{DEST}[n+63:n+32] \leftarrow \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[n+63:n+32]\times\text{SRC3}[n+63:n+32] -\text{DEST}[n+63:n+32])\)
}

IF (VEX.128) THEN
   \(\text{DEST}[\text{MAXVL-1:128}] \leftarrow 0\)
ELSEIF (VEX.256)
   \(\text{DEST}[\text{MAXVL-1:256}] \leftarrow 0\)
FI
VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI
    ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                            ELSE
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                            FI;
                        ELSE
                    IF (EVEX.b = 1)
                        THEN
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                            ELSE
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                            FI;
                    ELSE
                        FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

((KL, VL) = (4, 128), (8, 256), (16, 512))

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                        ELSE
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                    DEST[i+31:i] ←
                                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                            ELSE
                                    DEST[i+31:i] ←
                                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                            FI;
                        ELSE
                            ELSE
                                IF *merging-masking* ; merging-masking
                                    THEN *DEST[i+31:i] remains unchanged*
                                ELSE ; zeroing-masking
                                    DEST[i+31:i] ← 0
                                FI
                        FI
        ELSE
            ENDFOR
    DEST[MAXVL-1:VL] ← 0
VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 32

    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                        RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                ELSE DEST[i+31:i] ←
                        RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                            ELSE
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                            FI;
                        ELSE
                            IF (EVEX.b = 1)
                                THEN
                                    DEST[i+31:i] ←
                                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                                ELSE
                                    DEST[i+31:i] ←
                                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                                FI;
                            ELSE
                                IF *merging-masking* ; merging-masking
                                    THEN *DEST[i+31:i] remains unchanged*
                                ELSE ; zeroing-masking
                                    DEST[i+31:i] ← 0
                                FI
                            FI
            FI
    ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBADDxxxPS __m512 _mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m256 _mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 9A /r VFM SUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 AA /r VFM SUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 BA /r VFM SUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 9A /r VFM SUB132PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 AA /r VFM SUB213PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 BA /r VFM SUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 9A /r VFM SUB132PD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 AA /r VFM SUB213PD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 BA /r VFM SUB231PD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 9A /r VFM SUB132PD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 AA /r VFM SUB213PD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 BA /r VFM SUB231PD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 9A /r VFM SUB132PD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 AA /r VFM SUB213PD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 BA /r VFM SUB231PD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>
VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

**Description**

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
**Operation**

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
   n ← 64*i;
   DEST[n+63:n] ← RoundFPControl_MXCSR(Dest[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}

IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI

**VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
   n ← 64*i;
}

IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI

**VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
   n ← 64*i;
   DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - Dest[n+63:n])
}

IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI
VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

    SET_Rm(EVEX.RC);

ELSE

    SET_Rm(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

    i ← j * 64

    IF k1[j] OR *no writemask*

        THEN DEST[i+63:i] ←

            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])

        ELSE

            IF *merging-masking* ; merging-masking

                THEN *DEST[i+63:i] remains unchanged* 

            ELSE ; zeroing-masking

                DEST[i+63:i] ← 0

            FI

        FI

ENDFOR

DEST[MAXVL-1:VL] ← 0

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j * 64

    IF k1[j] OR *no writemask*

        THEN

            IF (EVEX.b = 1)

                THEN

                    DEST[i+63:i] ←

                        RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])

                ELSE

                    DEST[i+63:i] ←

                        RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])

                FI;

        ELSE

            IF *merging-masking* ; merging-masking

                THEN *DEST[i+63:i] remains unchanged* 

            ELSE ; zeroing-masking

                DEST[i+63:i] ← 0

            FI

        FI

ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] ← RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        ELSE
            IF *merging-masking*  ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE  ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0

VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE
                    DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                FI;
        ELSE
            IF *merging-masking*  ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE  ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0
VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR "no writemask"
THEN
    DEST[i+63:i] ← RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
ELSE
    IF "merging-masking" ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR "no writemask"
THEN
    IF (EVEX.b = 1)
        THEN
            DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i:63] - DEST[i+63:i])
        ELSE
            DEST[i+63:i] ← RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        FI;
    ELSE
        IF "merging-masking" ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_fmsub_pd(__m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_round_pd(__mmask8 k, __m256d a, __m256d b, __m256d c, int r);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_round_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_fmsub_pd(__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm256_fmsub_pd(__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_mask_fmsub_round_pd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_round_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_round_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
### VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 9A /r VFMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 AA /r VFMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 BA /r VFMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Operation
In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**
IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
   n ← 32*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI

**VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**
IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
   n ← 32*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI

**VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**
IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
   n ← 32*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI
VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] ←
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j \leftarrow 0 \text{ TO } KL-1
    i \leftarrow j \times 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] \leftarrow RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] \leftarrow 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j \leftarrow 0 \text{ TO } KL-1
    i \leftarrow j \times 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] \leftarrow RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                ELSE
                    DEST[i+31:i] \leftarrow RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+31:i] \leftarrow 0
                FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0
VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
  ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
      THEN
        DEST[i+31:i] ←
        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:i] - DEST[i+31:i])
    ELSE
        DEST[i+31:i] ←
        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
    DEST[i+31:i] ← 0
  FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxPS __m512 __m512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __m512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __m512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __m512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __m512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 __m512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __m512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __m512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 __m256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __m256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __m256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 __m128_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __m128_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __m128_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m256 __m256_fmsub_ps(__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM or a 64-bit memory location.

VFMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low quadword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation
In the operations below, “*“ and “-“ symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
THEN DEST[63:0]  RoundFPControl(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[63:0]  0
    FI;
FI;
DEST[127:64]  DEST[127:64]
DEST[MAXVL-1:128]  0

VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
THEN DEST[63:0]  RoundFPControl(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[63:0]  0
    FI;
FI;
DEST[127:64]  DEST[127:64]
DEST[MAXVL-1:128]  0
VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(Dest[63:0]* SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

**VFMSUB132SS**: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFMSUB213SS**: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFMSUB231SS**: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

If (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_RMVEX(RC);
ELSE

SET_RMMXCSR.RM;
FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] \( \leftarrow \) RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] \( \leftarrow \) 0

FI;

FI;

DEST[127:32] \( \leftarrow \) DEST[127:32]

DEST[MAXVL-1:128] \( \leftarrow \) 0

**VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

If (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_RMVEX(RC);
ELSE

SET_RMMXCSR.RM;
FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] \( \leftarrow \) RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] \( \leftarrow \) 0

FI;

FI;

DEST[127:32] \( \leftarrow \) DEST[127:32]

DEST[MAXVL-1:128] \( \leftarrow \) 0
VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
            FI;
    FI;
DEST[MAXVL-1:128] ← 0

VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(Dest[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[MAXVL-1:128] ← 0

VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*Dest[31:0] - SRC3[31:0])
DEST[MAXVL-1:128] ← 0

VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**
VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask2_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask2_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);

**SIMD Floating-Point Exceptions**
Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm0 {k1}{z}, ymm1, ymm2/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm0 {k1}{z}, ymm1, ymm2/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm0 {k1}{z}, ymm1, ymm2/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 9C /r VFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst[er]</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 AC /r VFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst[er]</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 BC /r VFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst[er]</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>
**VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values**

**INSTRUCTION SET REFERENCE, V-Z**

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VFNMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).
VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

    SET_RM(EVEX.RC);
ELSE

    SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR *no writemask*

THEN DEST[i+63:i] ←
     RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
ELSE

    IF *merging-masking* ; merging-masking

    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking

    DEST[i+63:i] ← 0

FI

ENDFOR

DEST[MAXVL-1:VL] ← 0

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR *no writemask*

THEN

    IF (EVEX.b = 1)

    THEN

        DEST[i+63:i] ←
        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
    ELSE

        DEST[i+63:i] ←
        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])

    FI;
ELSE

    IF *merging-masking* ; merging-masking

    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking

    DEST[i+63:i] ← 0

    FI

ENDFOR

DEST[MAXVL-1:VL] ← 0
VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)  
(KL, VL) = (2, 128), (4, 256), (8, 512)  
IF (VL = 512) AND (EVEX.b = 1)  
THEN  
    SET_RM(EVEX.RC);  
ELSE  
    SET_RM(MXCSR.RM);  
FI;  
FOR j ← 0 TO KL-1  
i ← j * 64  
IF k1[j] OR *no writemask*  
THEN DEST[i+63:i] ←  
    RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])  
ELSE  
    IF *merging-masking* ; merging-masking  
    THEN *DEST[i+63:i] remains unchanged*  
    ELSE ; zeroing-masking  
        DEST[i+63:i] ← 0  
    FI  
FI;  
ENDFOR  
DEST[MAXVL-1:VL] ← 0  

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)  
(KL, VL) = (2, 128), (4, 256), (8, 512)  
FOR j ← 0 TO KL-1  
i ← j * 64  
IF k1[j] OR *no writemask*  
THEN  
    IF (EVEX.b = 1)  
    THEN  
        DEST[i+63:i] ←  
        RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])  
    ELSE  
        DEST[i+63:i] ←  
        RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])  
    FI;  
ELSE  
    IF *merging-masking* ; merging-masking  
    THEN *DEST[i+63:i] remains unchanged*  
    ELSE ; zeroing-masking  
        DEST[i+63:i] ← 0  
    FI  
ENDFOR  
DEST[MAXVL-1:VL] ← 0
VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
        RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
            ELSE
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPD _m512d _mm512_fnmadd_pd(_m512d a, _m512d b, _m512d c);
VFNMADDxxxPD _m512d _mm512_fnmadd_round_pd(_m512d a, _m512d b, _m512d c, int r);
VFNMADDxxxPD _m512d _mm512_mask_fnmadd_pd(_m512d a, _mmask8 k, _m512d b, _m512d c);
VFNMADDxxxPD _m512d _mm512_maskz_fnmadd_pd(_mmask8 k, _m512d a, _m512d b, _m512d c);
VFNMADDxxxPD _m512d _mm512_mask3_fnmadd_pd(_m512d a, _m512d b, _m512d c, _mmask8 k);
VFNMADDxxxPD _m512d _mm512_mask_fnmadd_round_pd(_m512d a, _mmask8 k, _m512d b, _m512d c, int r);
VFNMADDxxxPD _m512d _mm512_maskz_fnmadd_round_pd(_mmask8 k, _m512d a, _m512d b, _m512d c, int r);
VFNMADDxxxPD _m512d _mm512_mask3_fnmadd_round_pd(_m512d a, _m512d b, _m512d c, _mmask8 k, int r);
VFNMADDxxxPD _m256d _mm256_fnmadd_pd(_m256d a, _m256d b, _m256d c);
VFNMADDxxxPD _m256d _mm256_mask_fnmadd_pd(_m256d a, _mmask8 k, _m256d b, _m256d c);
VFNMADDxxxPD _m256d _mm256_maskz_fnmadd_pd(_mmask8 k, _m256d a, _m256d b, _m256d c);
VFNMADDxxxPD _m256d _mm256_mask3_fnmadd_pd(_m256d a, _m256d b, _m256d c, _mmask8 k);
VFNMADDxxxPD _m128d _mm128_fnmadd_pd(_m128d a, _m128d b, _m128d c);
VFNMADDxxxPD _m128d _mm128_mask_fnmadd_pd(_mmask8 k, _m128d a, _m128d b, _m128d c);
VFNMADDxxxPD _m128d _mm128_maskz_fnmadd_pd(_mmask8 k, _m128d a, _m128d b, _m128d c);
VFNMADDxxxPD _m128d _mm128_mask3_fnmadd_pd(_m128d a, _m128d b, _m128d c, _mmask8 k);
VFNMADDxxxPD _m128d _mm128_fnmadd_pd(_m128d a, _m128d b, _m128d c);
VFNMADDxxxPD _m256d _mm256_fnmadd_pd(_m256d a, _m256d b, _m256d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
## VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
</tr>
</tbody>
</table>

### EVEX Version

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 BC /r VFNMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
</tr>
</tbody>
</table>

### EVEX.512 Version

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 9C /r VFNMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 AC /r VFNMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 BC /r VFNMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
VFNMADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
   n ← 32*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] ← 0
FI

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values
VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
MAXNUM ← 2
ELSEIF (VEX.256)
MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(-(SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMADD231PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
MAXNUM ← 2
ELSEIF (VEX.256)
MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(-(SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
        RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ← RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
  ENDFOR
DEST[MAXVL-1:VL] ← 0

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])
                ELSE
                    DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
                FI
            FI;
    ENDFOR
DEST[MAXVL-1:VL] ← 0

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] ← 0
VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPS __m512 _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps(__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DS.L66.0F38.W1 9D /r VFNMADD132SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DS.L66.0F38.W1 AD /r VFNMADD213SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DS.L66.0F38.W1 BD /r VFNMADD231SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DS.L66.0F38.W1 9D /r VFNMADD132SD xmm1 (k1)(z), xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DS.L66.0F38.W1 AD /r VFNMADD213SD xmm1 (k1)(z), xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DS.L66.0F38.W1 BD /r VFNMADD231SD xmm1 (k1)(z), xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

**VFNMADD132SD**: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMADD213SD**: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMADD231SD**: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version**: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvv/ EVEX.vvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "+" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

\[ \text{SET\_RM} \left( EVEX.RC \right) ; \]

ELSE

\[ \text{SET\_RM} \left( \text{MXCSR.RM} \right) ; \]

FI;

IF k1[0] or *no writemask*

THEN \[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl} \left( -(\text{DEST}[63:0] \times \text{SRC3}[63:0]) + \text{SRC2}[63:0] \right) \]

ELSE

\[ \text{DEST}[63:0] \text{ remains unchanged}; \]

THEN DEST[63:0] \leftarrow 0

FI;

FI;

\[ \text{DEST}[127:64] \leftarrow \text{DEST}[127:64] \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]

**VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

\[ \text{SET\_RM} \left( EVEX.RC \right) ; \]

ELSE

\[ \text{SET\_RM} \left( \text{MXCSR.RM} \right) ; \]

FI;

IF k1[0] or *no writemask*

THEN \[ \text{DEST}[63:0] \leftarrow \text{RoundFPControl} \left( -(\text{SRC2}[63:0] \times \text{DEST}[63:0]) + \text{SRC3}[63:0] \right) \]

ELSE

\[ \text{DEST}[63:0] \text{ remains unchanged}; \]

THEN DEST[63:0] \leftarrow 0

FI;

FI;

\[ \text{DEST}[127:64] \leftarrow \text{DEST}[127:64] \]

\[ \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \]
VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN
       DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[63:0] remains unchanged*
           ELSE ; zeroing-masking
               THEN DEST[63:0] ← 0
               FI;
   FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxSD __m128d _mm_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEV-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 9D /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 AD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 BD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 9D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 AD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 BD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMz/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMz/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “*“ and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN DEST[31:0]  RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[31:0] remains unchanged*
           ELSE ; zeroing-masking
               THEN DEST[31:0]  0
       FI;
   FI;
DEST[MAXVL-1:128]  0

**VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN DEST[31:0]  RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[31:0] remains unchanged*
           ELSE ; zeroing-masking
               THEN DEST[31:0]  0
       FI;
   FI;
DEST[MAXVL-1:128]  0
VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        Dest[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + Dest[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *Dest[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN Dest[31:0] ← 0
        FI;
    FI;
Dest[MAXVL-1:128] ← 0

VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)
Dest[31:0] ← RoundFPControl_MXCSR( (Dest[31:0]*SRC3[31:0]) + SRC2[31:0])
Dest[MAXVL-1:128] ← 0

VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)
Dest[31:0] ← RoundFPControl_MXCSR( (SRC2[31:0]*Dest[31:0]) + SRC3[31:0])
Dest[MAXVL-1:128] ← 0

VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)
Dest[31:0] ← RoundFPControl_MXCSR( (SRC2[31:0]*SRC3[31:0]) + Dest[31:0])
Dest[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxSS __m128 __m128_fnmadd_round_ss (__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 __m128_mask_fnmadd_ss (__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 __m128_maskz_fnmadd_ss (__m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 __m128_mask3_fnmadd_ss (__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 __m128_mask_fnmadd_round_ss (__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 __m128_maskz_fnmadd_round_ss (__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 __m128_mask3_fnmadd_round_ss (__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 __m128_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
### VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 9E /r VFNMSUB132PD zmm1 (k1){z}, zmm2, zmm3/m512/m64bcst(er)</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 AE /r VFNMSUB213PD zmm1 (k1){z}, zmm2, zmm3/m512/m64bcst(er)</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 BE /r VFNMSUB231PD zmm1 (k1){z}, zmm2, zmm3/m512/m64bcst(er)</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
Description

VFNMSUB132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(- (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(- (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] ← RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
          FI
        FI;
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
    RoundFPControl(-SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
    ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAXVL-1:VL] ← 0
VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
                ELSE
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
            FI
    ENDFOR

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
        RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR

DEST[MAXVL-1:VL] ← 0
VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(- (SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(- (SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged* 
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxPD __m512d_mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d_mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d_mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d_mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d_mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d_mm512_mask_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __m512d k);
VFNMSUBxxxPD __m512d_mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d_mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m256d_mm256_fnmsub_pd(__m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d_mm256_fnmsub_round_pd(__m256d a, __m256d b, __m256d c, int r);
VFNMSUBxxxPD __m256d_mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d_mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d_mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d_mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d_mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d_mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d_mm_fnmsub_pd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m256d_mm256_fnmsub_pd(__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
### VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 BE /r VFNMSUB231PS ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.0F.0E 9E /r VFNMSUB132PS ymm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.0F.0E AE /r VFNMSUB213PS ymm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.0F.0E BE /r VFNMSUB231PS ymm2 (k1){z}, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.0F.0E 9E /r VFNMSUB132PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.0F.0E AE /r VFNMSUB213PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.0F.0E BE /r VFNMSUB231PS ymm2 (k1){z}, ymm3/m256/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.0F.0E 9E /r VFNMSUB132PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.0F.0E AE /r VFNMSUB213PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.0F.0E BE /r VFNMSUB231PS zmm2 (k1){z}, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
VFNMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation
In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version)**

```assembly
IF (VEX.128) THEN
    MAXNUM <- 2
ELSEIF (VEX.256)
    MAXNUM <- 4
FI
For i = 0 to MAXNUM-1 {
    n <- 32*i;
    DEST[n+31:n] <- RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] <- 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] <- 0
FI
```

**VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values**
VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM ← 2
ELSEIF (VEX.256)
  MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
  n ← 32*i;
  DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] ← 0
FI

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM ← 2
ELSEIF (VEX.256)
  MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
  n ← 32*i;
  DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] ← 0
FI

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(\(KL, VL\) = \(4, 128\), \(8, 256\), \(16, 512\))

FOR \(j \leftarrow 0\) TO \(KL-1\)
    \(i \leftarrow j \times 32\)
    IF \(k1[j]\) OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[\(i+31:i\)] \leftarrow \text{RoundFPControl\_MXCSR}(-(DEST[\(i+31:i\)]*SRC3[\(31:0\)]) - SRC2[\(i+31:i\)])
                ELSE
                    DEST[\(i+31:i\)] \leftarrow \text{RoundFPControl\_MXCSR}(-(DEST[\(i+31:i\)]*SRC3[\(i+31:i\)]) - SRC2[\(i+31:i\)])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN \(\text{DEST}[i+31:i]\) remains unchanged*
                ELSE ; zeroing-masking
                    DEST[\(i+31:i\)] \leftarrow 0
                FI
            FI;
    ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\)

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(\(KL, VL\) = \(4, 128\), \(8, 256\), \(16, 512\))

IF (\(VL = 512\)) AND (EVEX.b = 1)
    THEN
        \(\text{SET\_RM}(\text{EVEX.RC});\)
    ELSE
        \(\text{SET\_RM}(\text{MXCSR.RM});\)
    FI;

FOR \(j \leftarrow 0\) TO \(KL-1\)
    \(i \leftarrow j \times 32\)
    IF \(k1[j]\) OR *no writemask*
        THEN \(\text{DEST}[i+31:i] \leftarrow \text{RoundFPControl\_MXCSR}(-(SRC2[\(i+31:i]\)*DEST[\(i+31:i\)]) - SRC3[\(i+31:i\)])\)
        ELSE
            IF *merging-masking* ; merging-masking
                THEN \(\text{DEST}[i+31:i]\) remains unchanged*
                ELSE ; zeroing-masking
                    DEST[\(i+31:i\)] \leftarrow 0
                FI
        FI;
    ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0\)
VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
          ELSE
          DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        FI;
      ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
    DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
    ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0
VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

KL, VL = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
        ELSE
          DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
      FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
        FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPS __m512 _mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m256 _mm256_fnmsub_ps(__m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_fnmsub_round_ps(__m256 a, __m256 b, __m256 c, int r);
VFNMSUBxxxPS __m256 _mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm128_fnmsub_ps(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm128_fnmsub_round_ps(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxPS __m128 _mm128_mask_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm128_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm128_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
## VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 9F /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 AF /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 BF /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 9F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 AF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 BF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

**VFNMSUB132SD**: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB213SD**: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB231SD**: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version**: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low quadword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation
In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_RM(EVEX.RC);
ELSE

SET_RM(MXCSR.RM);
FI;

IF k1[0] or *no writemask*

THEN DEST[63:0] ← RoundFPControl(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*
ELSE ; zeroing-masking

THEN DEST[63:0] ← 0
FI;
FI;

DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

**VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_RM(EVEX.RC);
ELSE

SET_RM(MXCSR.RM);
FI;

IF k1[0] or *no writemask*

THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*
ELSE ; zeroing-masking

THEN DEST[63:0] ← 0
FI;
FI;

DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ←RoundFPControl_MXCSR(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ←RoundFPControl_MXCSR(-(SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ← 0

VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ←RoundFPControl_MXCSR(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_sd(__mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __mmask8 k);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.8B 9F /r VFNMSUB132SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.8B AF /r VFNMSUB213SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.8B BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.8B BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANS are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_RM(EVEX.RC);

ELSE

SET_RM(MXCSR.RM);

FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] \leftarrow \text{RoundFPControl}(-\text{DEST}[31:0]*\text{SRC3}[31:0]) - \text{SRC2}[31:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] \leftarrow 0

FI;

FI;

DEST[127:32] \leftarrow DEST[127:32]

DEST[MAXVL-1:128] \leftarrow 0

**VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_RM(EVEX.RC);

ELSE

SET_RM(MXCSR.RM);

FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] \leftarrow \text{RoundFPControl}(-(\text{SRC2}[31:0]*\text{DEST}[31:0]) - \text{SRC3}[31:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] \leftarrow 0

FI;

FI;

DEST[127:32] \leftarrow DEST[127:32]

DEST[MAXVL-1:128] \leftarrow 0
VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[MAXVL-1:128] ← 0

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(-(SRC2[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[MAXVL-1:128] ← 0

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[MAXVL-1:128] ← 0

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(-(SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFPCLASSPD—Tests Types Of a Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 [k1], xmm2/m128/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 [k1], ymm2/m256/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 [k1], zmm2/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The FCLASSPD instruction checks the packed double precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-4.

![Figure 5-13. Imm8 ByteSpecifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS](image)

Table 5-4. Classifier Operations for VFPCLASSPD/SD/PS/SS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>QNaN</td>
<td>PosZero</td>
<td>NegZero</td>
<td>PosINF</td>
<td>NegINF</td>
<td>Denormal</td>
<td>Negative</td>
<td>SNAN</td>
</tr>
<tr>
<td>Classifier</td>
<td>Checks for QNaN</td>
<td>Checks for +0</td>
<td>Checks for -0</td>
<td>Checks for +INF</td>
<td>Checks for -INF</td>
<td>Checks for Denormal</td>
<td>Checks for Negative finite</td>
<td>Checks for SNan</td>
</tr>
</tbody>
</table>

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    //* Start checking the source operand for special type */
    NegNum ← tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSEIF (tsrc[51:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber ← ExpAllZeros AND MantAllZeros
    SignalingBit ← tsrc[51];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    Pinf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Ninf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
    Return bResult;
} //* end of CheckFPClassDP() */

VFCLASSPD (EVEX Encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] ← CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                    ELSE
                        DEST[j] ← CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
                    FI;
            ELSE DEST[j] ← 0 ; zeroing-masking only
            FI;
        ENDFOR
    DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSPD __mmask8 __mm512_fpclass_pd_mask(__m512d a, int c);
VFPCLASSPD __mmask8 __mm512_mask_fpclass_pd_mask(__mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 __mm256_fpclass_pd_mask(__m256d a, int c)
VFPCLASSPD __mmask8 __mm256_mask_fpclass_pd_mask(__mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 __mm_fpclass_pd_mask(__m128d a, int c)
VFPCLASSPD __mmask8 __mm_mask_fpclass_pd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4
#UD If EVEX.vvvv != 1111B.
VFCLASSPS—Tests Types Of a Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The FPCLASSPS instruction checks the packed single-precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-4.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0])

```c
    //** Start checking the source operand for special type **/
    NegNum ← tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; Fl;
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros ← 1;
        Fl;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];
    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
```
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR

Return bResult;
} /* end of CheckSPClassSP() */

**VFCLASSPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                        DEST[j] ← CheckFPClassDP(SRC1[31:0], imm8[7:0]);
                ELSE
                        DEST[j] ← CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
                FI;
            ELSE DEST[j] ← 0 ; zeroing-masking only
        FI;
    ENDFOR

DEST[MAX_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFCLASSPS __m128 _mm_fpclass_ps_mask( __m128 a, int c);
VFCLASSPS __m16 _mm512_fpclass_ps_mask( __m512 a, int c);
VFCLASSPS __m16 _mm512_mask_fpclass_ps_mask( __m16 m, __m512 a, int c);
VFCLASSPS __m256 _mm256_fpclass_ps_mask( __m256 a, int c);
VFCLASSPS __m256 _mm256_mask_fpclass_ps_mask( __m256 m, __m256 a, int c);
VFCLASSPS __m128 _mm128_fpclass_ps_mask( __m128 a, int c);
VFCLASSPS __m128 _mm128_mask_fpclass_ps_mask( __m128 m, __m128 a, int c);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4

#UD If EVEX.vvvv != 1111B.
VFPCLASSSD—Tests Types Of a Scalar Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFPCLASSSD k2</td>
<td>A/V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

Description

The FCLASSSD instruction checks the low double precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1:1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-4.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

NegNum ← tsrc[63];
IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; Fi;
IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
  MantAllZeros ← 1;
ELSIF (tsrc[51:0]=0h) Then
  MantAllZeros ← 1;
FI;
ZeroNumber ← ExpAllZeros AND MantAllZeros
SignalingBit ← tsrc[51];

sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
Pinf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
Ninf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR

Return bResult;
} //* end of CheckFPClassDP() *//
VFPCLASSSD (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[0] ← CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE DEST[0] ← 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFPCLASSSD __mmask8 _mm_fpclass_sd_mask(__m128d a, int c)
VFPCLASSSD __mmask8 _mm_mask_fpclass_sd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E6
#UD If EVEX.vvvv != 1111B.
VFPCLASSSSS—Tests Types Of a Scalar Float32 Values

**Description**

The VFPCLASSSS instruction checks the low single-precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1:1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-4.

**Operation**

CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type */
    NegNum ← tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    Pinf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Ninf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR

    Return bResult;

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LG.66.0F3A.W0 67 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.**
VFPCCLASSSS (EVEX encoded version)
IF k1[0] OR *no writemask*
     THEN DEST[0] ←
         CheckFPClassSP(SRC1[31:0], imm8[7:0])
     ELSE DEST[0] ← 0 ; zeroing-masking only
Fl;
DEST[MAX_KL-1:1] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFPCCLASSSS __mmask8 _mm_fpclass_ss_mask(__m128 a, int c)
VFPCCLASSSS __mmask8 _mm_mask_fpclass_ss_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E6
#UD If EVEX.vvvv != 1111B.
VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 2-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W1 92 /r VGATHERDPD xmm1, vm32x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 93 /r VGATHERQPD xmm1, vm64x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 92 /r VGATHERDPD ymm1, vm32x, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 93 /r VGATHERQPD ymm1, vm64y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (r, w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.
VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

• If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
• The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• This instruction will cause a #UD if the address size attribute is 16-bit.
• This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
• This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
• The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.
**Operation**

DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

**VGATHERDPD (VEX.128 version)**

FOR j ← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 1
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR

MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

**VGATHERQPD (VEX.128 version)**

FOR j ← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 1
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR

MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
VGATHERQPD (VEX.256 version)
FOR j ← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERDPD (VEX.256 version)
FOR j ← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 3
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR

(non-masked elements of the mask register have the content of respective element cleared)
**Intel C/C++ Compiler Intrinsic Equivalent**

VGATHERDPD: ```__m128d __mm_i32gather_pd (double const * base, __m128i index, const int scale);```  
VGATHERDPD: ```__m128d __mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);```  
VGATHERDPD: ```__m256d __mm256_i32gather_pd (double const * base, __m128i index, const int scale);```  
VGATHERDPD: ```__m256d __mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);```  
VGATHERQPD: ```__m128d __mm_i64gather_pd (double const * base, __m128i index, const int scale);```  
VGATHERQPD: ```__m128d __mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);```  
VGATHERQPD: ```__m256d __mm256_i64gather_pd (double const * base, __m256i index, const int scale);```  
VGATHERQPD: ```__m256d __mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 12.
VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 -bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS xmm1, vm32x, xmm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64x, xmm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS ymm1, vm32y, ymm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather single-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64y, xmm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero’ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.
VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:

• If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
• The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• This instruction will cause a #UD if the address size attribute is 16-bit.
• This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
• This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
• The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
VGATHERDPS (VEX.128 version)
FOR j ← 0 to 3
  \( i \leftarrow j \times 32; \)
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 3
  \( i \leftarrow j \times 32; \)
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.128 version)
FOR j ← 0 to 3
  \( i \leftarrow j \times 32; \)
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 1
  \( k \leftarrow j \times 64; \)
  \( i \leftarrow j \times 32; \)
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
**VGATHERDPS (VEX.256 version)**

FOR $j \leftarrow 0$ to $7$
  
  $i \leftarrow j \times 32$;
  
  IF $\text{MASK}[31+i]$ THEN
    
    $\text{MASK}[i+31:i] \leftarrow FFFFFFFFH$; // extend from most significant bit
  
  ELSE
    
    $\text{MASK}[i+31:i] \leftarrow 0$
  
  FI;

ENDFOR

FOR $j \leftarrow 0$ to $7$
  
  $i \leftarrow j \times 32$;
  
  $\text{DATA_ADDR} \leftarrow \text{BASE_ADDR} + (\text{SignExtend(VINDEX1}[i+31:i]) \times \text{SCALE} + \text{DISP}$;
  
  IF $\text{MASK}[31+i]$ THEN
    
    $\text{DEST}[i+31:i] \leftarrow \text{FETCH_32BITS}(\text{DATA_ADDR})$; // a fault exits the instruction
  
  FI;

ENDFOR

(non-masked elements of the mask register have the content of respective element cleared)

**VGATHERQPS (VEX.256 version)**

FOR $j \leftarrow 0$ to $7$
  
  $i \leftarrow j \times 32$;
  
  IF $\text{MASK}[31+i]$ THEN
    
    $\text{MASK}[i+31:i] \leftarrow FFFFFFFFH$; // extend from most significant bit
  
  ELSE
    
    $\text{MASK}[i+31:i] \leftarrow 0$
  
  FI;

ENDFOR

FOR $j \leftarrow 0$ to $3$
  
  $k \leftarrow j \times 64$;
  
  $i \leftarrow j \times 32$;
  
  $\text{DATA_ADDR} \leftarrow \text{BASE_ADDR} + (\text{SignExtend(VINDEX1}[k+63:k]) \times \text{SCALE} + \text{DISP}$;
  
  IF $\text{MASK}[31+i]$ THEN
    
    $\text{DEST}[i+31:i] \leftarrow \text{FETCH_32BITS}(\text{DATA_ADDR})$; // a fault exits the instruction
  
  FI;

ENDFOR

$\text{MASK}[\text{MAXVL}-1:128] \leftarrow 0$;

$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$;

(non-masked elements of the mask register have the content of respective element cleared)
Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPS: __m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);
VGATHERDPS: __m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);
VGATHERDPS: __m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);
VGATHERDPS: __m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);
VGATHERQPS: __m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);
VGATHERQPS: __m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);
VGATHERQPS: __m256 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);
VGATHERQPS: __m256 _mm256_mask_i64gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 12.
VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R): VSIB:index</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

A set of single-precision/double-precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
This instruction does not perform AC checks, and so will never deliver an AC fault.

- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VGATHERDPS (EVEX encoded version)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j \times 32\]

\[\text{IF } k1[j] \]

\[\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + \text{SignExtend}(\text{VINDEX}[i+31:i]) \times \text{SCALE} + \text{DISP}]\]

\[k1[j] \leftarrow 0\]

\[\text{ELSE } \text{DEST}[i+31:i] \leftarrow \text{remains unchanged}\]

\[\text{FI; ENDFORE}\]

\[k1[\text{MAX}_K_1:KL] \leftarrow 0\]

\[\text{DEST}[\text{MAX}_{VL}:VL] \leftarrow 0\]

**VGATHERDPD (EVEX encoded version)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j \times 64\]

\[k \leftarrow j \times 32\]

\[\text{IF } k1[j] \]

\[\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + \text{SignExtend}(\text{VINDEX}[k+31:k]) \times \text{SCALE} + \text{DISP}]\]

\[k1[j] \leftarrow 0\]

\[\text{ELSE } \text{DEST}[i+63:i] \leftarrow \text{remains unchanged}\]

\[\text{FI; ENDFORE}\]

\[k1[\text{MAX}_K_1:KL] \leftarrow 0\]

\[\text{DEST}[\text{MAX}_{VL}:VL] \leftarrow 0\]
Intel C/C++ Compiler Intrinsic Equivalent
VGATHERDPD __m512d _mm512_i32gather_pd(__m256i vdx, void * base, int scale);
VGATHERDPD __m512d _mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d _mm256_mask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d _mm_mask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 __mm256_mask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPS __m128 __mm_mask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12.
VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x</td>
<td>A /vsib</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y</td>
<td>A /vsib</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z</td>
<td>A /vsib</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x</td>
<td>A /vsib</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y</td>
<td>A /vsib</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z</td>
<td>A /vsib</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

A set of 8 single-precision/double-precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

This instruction does not perform AC checks, and so will never deliver an AC fault.

Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

**VGATHERQPS (EVEX encoded version)**

(KL, VL) = (2, 64), (4, 128), (8, 256)
FOR j ← 0 TO KL-1

\[
i \leftarrow j \times 32 \\
k \leftarrow j \times 64 \\
\text{IF } k1[j] \text{ OR *no writemask*} \\
\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + (\text{VINDEX}[k+63:k] \times \text{SCALE} + \text{DISP}] \\
\quad k1[j] \leftarrow 0 \\
\text{ELSE } \text{DEST}[i+31:i] \leftarrow \text{remains unchanged*} \\
\text{FI;}
\]

ENDFOR

k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL/2] ← 0

**VGATHERQPD (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1

\[
i \leftarrow j \times 64 \\
k \leftarrow j \times 64 \\
\text{IF } k1[j] \text{ OR *no writemask*} \\
\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + (\text{VINDEX}[i+63:i] \times \text{SCALE} + \text{DISP}] \\
\quad k1[j] \leftarrow 0 \\
\text{ELSE } \text{DEST}[i+63:i] \leftarrow \text{remains unchanged*} \\
\text{FI;}
\]

ENDFOR

k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VGATHERQPD __m512d _mm512_i64gather_pd(__m512i vdx, void * base, int scale);
VGATHERQPD __m512d _mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d _mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d _mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);

VGATHERQPS __m256 _mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 _mm_mask_i64gather_ps(__m128 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.
VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 42 /r VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-5.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for the greatest integer not exceeding real number \( x \).

<table>
<thead>
<tr>
<th>Input Operand</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = NaN</td>
<td>QNaN(src1)</td>
<td>If (SRC = SNaN) then #IE</td>
</tr>
<tr>
<td>0 &lt;</td>
<td>src1</td>
<td>&lt; INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= +INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= 0</td>
</tr>
</tbody>
</table>
**Operation**

NormalizeExpTinyDPFP(SRC[63:0])

```
[ 
   // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
   Src.Jbit ← 0;
   Dst.exp ← 1;
   Dst.fraction ← SRC[51:0];
   WHILE(Src.Jbit = 0)
   {
      Src.Jbit ← Dst.fraction[51]; // Get the fraction MSB
      Dst.fraction ← Dst.fraction << 1; // One bit shift left
      Dst.exp--; // Decrement the exponent
   }
   Dst.fraction ← 0; // zero out fraction bits
   Dst.sign ← 1; // Return negative sign
   TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
   Return (TMP[63:0]);
]}
```

ConvertExpDPFP(SRC[63:0])

```
[
   Src.sign ← 0; // Zero out sign bit
   Src.exp ← SRC[62:52];
   Src.fraction ← SRC[51:0];
   // Check for NaN
   IF (SRC = NaN)
   {
      IF ( SRC = SNAN ) SET IE;
      Return QNAN(SRC);
   }
   // Check for +INF
   IF (SRC = +INF) Return (SRC);
   // check if zero operand
   IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
   ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
   {
      IF ((Src.exp = 0) AND (Src.fraction != 0))
      {
         TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]); // Get Normalized Exponent
         Set #DE
      }
      ELSE // exponent value is correct
      {
         TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
      }
   }
   TMP ← SAR(TMP, 52); // Shift Arithmetic Right
   TMP ← TMP - 1023; // Subtract Bias
   Return CvtI2D(TMP); // Convert INT to Double-Precision FP number
]}
```
VGETEXPPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC is memory*)
                THEN
                    DEST[i+63:i] ← ConvertExpDPFP(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← ConvertExpDPFP(SRC[i+63:i])
                FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd(__mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd(__mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd(__mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Table 5-6. VGETEXPPS/SS Special Cases

<table>
<thead>
<tr>
<th>Input Operand</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = NaN</td>
<td>QNaN(src1)</td>
<td>If (SRC = SNaN) then #IE</td>
</tr>
<tr>
<td>src1 = 0</td>
<td>-INF</td>
<td>If (SRC = denormal) then #DE</td>
</tr>
<tr>
<td>src1 &lt; 0</td>
<td>floor(log_2(</td>
<td>src1</td>
</tr>
<tr>
<td>src1 = +INF</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>src1 = -INF</td>
<td>-INF</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

**Figure 5-14. VGETEXPPS Functionality On Normal Input values**

**Operation**

- **NormalizeExpTinySPFP**(SRC[31:0])
  
  ```
  // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
  Src.Jbit ← 0;
  Dst.exp ← 1;
  Dst.fraction ← SRC[22:0];
  WHILE(Src.Jbit = 0)
  {
    Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
    Dst.fraction ← Dst.fraction << 1 ; // One bit shift left
    Dst.exp-- ; // Decrement the exponent
  }
  Dst.fraction ← 0; // zero out fraction bits
  Dst.sign ← 1; // Return negative sign
  TMP[31:0] ← MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
  Return (TMP[31:0]);
  ```

- **ConvertExpSPFP**(SRC[31:0])
  
  ```
  Src.sign ← 0; // Zero out sign bit
  Src.exp ← SRC[30:23];
  Src.fraction ← SRC[22:0];
  // Check for NaN
  IF (SRC = NaN)
  {
    IF ( SRC = SNAN ) SET IE;
    Return QNAN(SRC);
  }
  // Check for +INF
  IF (SRC = +INF) Return (SRC);
  // check if zero operand
  IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
  ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
  ```
IF ((Src.exp = 0) AND (Src.fraction != 0))
{
    TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]);  // Get Normalized Exponent
    Set #DE
}
ELSE  // exponent value is correct
{
    TMP[31:0] ← (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction);
}
TMP ← SAR(TMP, 23);  // Shift Arithmetic Right
TMP ← TMP − 127;  // Subtract Bias
Return CvtI2D(TMP);  // Convert INT to Single-Precision FP number

VGETEXPPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN
            DEST[i+31:i] ← ConvertExpSPFP(SRC[31:0])
        ELSE
            DEST[i+31:i] ← ConvertExpSPFP(SRC[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPS __m512 __mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 __mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 __mm512_maskz_getexp_ps(__m512 a);
VGETEXPPS __m512 __mm512_getexp_round_ps(__m512 a, int sae);
VGETEXPPS __m512 __mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 __mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 __mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 __mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 __mm256_maskz_getexp_ps(__mmask8 k, __m256 a);
VGETEXPPS __m128 __mm128_getexp_ps(__m128 a);
VGETEXPPS __m128 __mm128_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a);
VGETEXPPS __m128 __mm128_maskz_getexp_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 43 /r</td>
<td>V</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the biased exponent (bits 62:52) of the low double-precision floating-point value in xmm3/m64 to a DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-5.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

Operation

// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

VGETEXPSD (EVEX encoded version)

IF k1[0] OR *no writemask*  
THEN DEST[63:0] ← ConvertExpDPFP(SRC2[63:0])
ELSE  
IF *merging-masking*  
THEN *DEST[63:0] remains unchanged*  
ELSE  
DEST[63:0] ← 0
FI

FI:
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSD __m128d _mm_getexp_sd(__m128d a, __m128d b);
VGETEXPSD __m128d __mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d __mm_maskz_getexp_sd(__mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d __mm_getexp_round_sd(__m128d a, __m128d b, int sae);
VGETEXPSD __m128d __mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d __mm_maskz_getexp_round_sd(__mmask8 k, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Exceptions Type E3.
**VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W0 43 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a SP FP value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

\[
\text{GETEXP}(x) = \text{floor}(\log_2(|x|))
\]

Notation \(\text{floor}(x)\) stands for maximal integer not exceeding real number \(x\).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

**Operation**

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

**VGETEXPSS (EVEX encoded version)**

IF k1[0] OR *no writemask*

THEN DEST[31:0] \(\leftarrow\) ConvertExpDPFP(SRC2[31:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*  
ELSE  ; zeroing-masking

DEST[31:0] \(\leftarrow\) 0

FI

ENDFOR

DEST[127:32] \(\leftarrow\) SRC1[127:32]

DEST[MAXVL-1:128] \(\leftarrow\) 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSS __m128 _mm_getexp_ss( __m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss( __m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss( __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss( __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss( __mmask8 k, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Exceptions Type E3.
VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 [k1]{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 [k1]{z}, ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 [k1]{z}, zmm2/m512/m64bcst{saе}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

![Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS](image)

For each input DP FP value x, The conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}|
\]

where:

\[
1 <= |x.\text{significand}| < 2
\]

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.
If interv \(!= 0\) then \(k = -1\), otherwise \(K = 0\). The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent \(k\) (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register \(k1\) are computed and stored into the destination. Elements in \(zmm1\) with the corresponding bit clear in \(k1\) retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b; otherwise instructions will #UD.

### Table 5-7. GetMant() Special Float Values Behavior

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
<th>Exceptions / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(SRC)</td>
<td>Ignore interv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (SRC = SNaN) then #IE</td>
</tr>
<tr>
<td>+∞</td>
<td>1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>+0</td>
<td>1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>-0</td>
<td>IF (SC[0]) THEN +1.0 ELSE -1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>-∞</td>
<td>IF (SC[1]) THEN (QNaN_Indefinite) ELSE { IF (SC[0]) THEN +1.0 ELSE -1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (SC[1]) then #IE</td>
</tr>
<tr>
<td>negative</td>
<td>SC[1] ? QNaN_Indefinite : Getmant(SRC)</td>
<td>If (SC[1]) then #IE</td>
</tr>
</tbody>
</table>

### Operation

GetNormalizeMantissaDP\(\text{SRC[63:0]}, \text{SignCtrl[1:0]}, \text{Interv[1:0]}\)

```c
// Extracting the SRC sign, exponent and mantissa fields
Dst.sign \(\leftarrow\) SignCtrl[0] ? 0 : Src[63];  // Get sign bit
Dst.exp \(\leftarrow\) SRC[62:52];  // Get original exponent value
Dst.fraction \(\leftarrow\) SRC[51:0];  // Get original fraction value
ZeroOperand \(\leftarrow (Dst.exp = 0) \text{ AND (Dst.fraction} = 0));
DenormOperand \(\leftarrow (Dst.exp = 0h) \text{ AND (Dst.fraction} = 0));
InfiniteOperand \(\leftarrow (Dst.exp = 07FFh) \text{ AND (Dst.fraction} = 0));
NaNOperand \(\leftarrow (Dst.exp = 07FFh) \text{ AND (Dst.fraction} = 0));
// Check for NAN operand
IF (NaNOperand)
{ IF (SRC = SNaN) \{Set #IE;  
    Return QNaN(SRC); 
  
}
// Check for Zero and Infinite operands
IF ((ZeroOperand) OR (InfiniteOperand))
{ Dst.exp \(\leftarrow\) 03FFh;  // Override exponent with BIAS
  Return ((Dst.sign<<63) | (Dst.exp<<52) | (Dst.fraction));
}
// Check for negative operand (including -0.0)
IF ((Src[63] = 1) AND SignCtrl[1])
{ Set #IE;  
  Return QNaN_Indefinite; 
}
```
// Checking for denormal operands
IF (DenormOperand)
{ IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
  ELSE
    { // Jbit is the hidden integral bit. Zero in case of denormal operand.
      Src.bit ← 0; // Zero Src Jbit
      Dst.exp ← 03FFh; // Override exponent with BIAS
      WHILE (Src.bit = 0) { // normalize mantissa
        Src.bit ← Dst.fraction[51]; // Get the fraction MSB
        Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
        Dst.exp--; // Adjust the exponent
      }
      SET #DE; // Set DE bit
    }
  }

// At this point, Dst.fraction is normalized.
// Checking for exponent response
Unbiased.exp ← Dst.exp - 03FFh; // subtract the bias from exponent
IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[51];
CASE (interv[1:0])
  00: Dst.exp ← 03FFh; // This is the bias
  01: Dst.exp ← (IsOddExp ? 03FEh : 03FFh); // either bias-1, or bias
  10: Dst.exp ← 03FEh; // bias-1
  11: Dst.exp ← (SignalingBit ? 03FEh : 03FFh); // either bias-1, or bias
ESAC
// At this point Dst.exp has the correct result. Form the final destination
DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
Return (DEST);
}

VGETMANTPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN
          DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[i+63:i], SignCtrl, Interv)
        ELSE
          DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[i+63:i], SignCtrl, Interv)
        FI;
    ELSE
      IF *merging-masking* // merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE // zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPD __m512d _mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd( __m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd( __m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd( __m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd( __m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VGETMANTPS—Extract float32 vector of normalized mantissas from float32 vector

Description

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input SP FP value x, The conversion operation is:

\[ \text{GetMant}(x) = \pm 2^k \times |x.\text{significand}| \]

where:

\[ 1 \leq |x.\text{significand}| < 2 \]

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.
Operation

GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0])
{
  // Extracting the SRC sign, exponent and mantissa fields
  Dst.sign <- SignCtrl[0] ? 0 : Src[31];  // Get sign bit
  Dst.exp <- SRC[30:23];  // Get original exponent value
  Dst.fraction <- SRC[22:0];  // Get original fraction value
  ZeroOperand <- (Dst.exp = 0) AND (Dst.fraction = 0);
  DenormOperand <- (Dst.exp = 0h) AND (Dst.fraction != 0);
  InfiniteOperand <- (Dst.exp = 0FFh) AND (Dst.fraction = 0);
  NaNOperand <- (Dst.exp = 0FFh) AND (Dst.fraction != 0);
  // Check for NAN operand
  IF (NaNOperand)
  { IF (SRC = SNaN) {Set #IE;}
    Return QNAN(SRC);
  }
  // Check for Zero and Infinite operands
  IF ((ZeroOperand) OR (InfiniteOperand))
  { Dst.exp <- 07Fh;  // Override exponent with BIAS
    Return ((Dst.sign<<31) | (Dst.exp<<23) | (Dst.fraction));
  }
  // Check for negative operand (including -0.0)
  IF ((Src[31] = 1) AND SignCtrl[1])
  { Set #IE;
    Return QNaN_Indefinite;
  }
  // Checking for denormal operands
  IF (DenormOperand)
  { IF (MXCSR.DAZ=1) Dst.fraction <- 0;  // Zero out fraction
    ELSE
    { // Jbit is the hidden integral bit. Zero in case of denormal operand.
      Src.Jbit <- 0;  // Zero Src Jbit
      Dst.exp <- 07Fh;  // Override exponent with BIAS
      WHILE (Src.Jbit = 0) {
        // normalize mantissa
        Src.Jbit <- Dst.fraction[22];  // Get the fraction MSB
        Dst.fraction <- (Dst.fraction << 1);  // Start normalizing the mantissa
        Dst.exp--;  // Adjust the exponent
      }
      SET #DE;  // Set DE bit
    }
  }  // At this point, Dst.fraction is normalized.
  // Checking for exponent response
  Unbiased.exp <- Dst.exp - 07Fh;  // subtract the bias from exponent
  IsOddExp <- Unbiased.exp[0];  // recognized unbiased ODD exponent
  SignalingBit <- Dst.fraction[22];
  CASE (interv[1:0])
  00: Dst.exp <- 07Fh;  // This is the bias
  01: Dst.exp <- (IsOddExp) ? 07Eh : 07Fh;  // either bias-1, or bias
  10: Dst.exp <- 07Eh;  // bias-1
  11: Dst.exp <- (SignalingBit) ? 07Eh : 07Fh;  // either bias-1, or bias
  ESAC

  // Form the final destination
  DEST[31:0] <- (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
}

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector
Return (DEST);
}

**VGETMANTPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN
                DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[31:0], SignCtrl, Interv)
            ELSE
                DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[i+31:i], SignCtrl, Interv)
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETMANTPS __m512 __mm512_getmant_ps( __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 __mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 __mm512_maskz_getmant_ps( __mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 __mm512_getmant_round_ps( __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 __mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 __mm512_maskz_getmant_round_ps( __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m256 __mm256_getmant_ps( __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 __mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 __mm256_maskz_getmant_ps( __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m128 __mm128_getmant_ps( __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 __mm128_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 __mm128_maskz_getmant_ps( __mmask8 k, __m128 a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

### Opcode/Description

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W1 27 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract the normalized mantissa of the low float64 element in xmm3/m64 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}| 
\]

where:

\[
1 \leq |x.\text{significand}| < 2 
\]

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

If interv \( \neq 0 \) then \( k = -1 \), otherwise \( K = 0 \). The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.
Operation
// GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

VGETMANTSD (EVEX encoded version)
SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ←
        GetNormalizedMantissaDP(SRC2[63:0], SignCtrl, Interv)
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            DEST[63:0] ← 0
        FI
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

SIMD Floating-Point Exceptions
Denormal, Invalid

Other Exceptions
See Exceptions Type E3.
VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 (k1)[(x)], xmm2, xmm3/m32{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}|
\]

where:

\[1 \leq |x.\text{significand}| < 2\]

Unbiased exponent \(k\) depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then \(k = -1\), otherwise \(K = 0\). The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted SP FP result is encoded according to the sign control, the unbiased exponent \(k\) (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.
**Operation**

// GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

**VGETMANTSS (EVEX encoded version)**

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ←
        GetNormalizedMantissaSP(SRC2[31:0], SignCtrl, Interv)
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] ← 0
        FI
    FI;
DEST[127:32] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E3.
### VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 18 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 18 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 18 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 18 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 1A /r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 1A /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed double-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

**VINSERTF128/VINSERTF32x4 and VINSERTF64x2** insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

**VINSERTF32x4**: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

**VINSERTF64x2**: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

**VINSERTF32x8 and VINSERTF64x4** inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.
Operation

**VINSERTF32x4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)
TEMP_dest[Vl-1:0] ← SRC1[Vl-1:0]
IF VL = 256
   CASE (imm8[0]) OF
      0: TEMP_dest[127:0] ← SRC2[127:0]
      1: TEMP_dest[255:128] ← SRC2[127:0]
   ESAC.
   FI;
IF VL = 512
   CASE (imm8[1:0]) OF
      00: TEMP_dest[127:0] ← SRC2[127:0]
      01: TEMP_dest[255:128] ← SRC2[127:0]
      10: TEMP_dest[383:256] ← SRC2[127:0]
      11: TEMP_dest[511:384] ← SRC2[127:0]
   ESAC.
   FI;
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TEMP_dest[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] ← 0
   FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**VINSERTF64x2 (EVEX encoded versions)**

(KL, VL) = (4, 256), (8, 512)
TEMP_dest[Vl-1:0] ← SRC1[Vl-1:0]
IF VL = 256
   CASE (imm8[0]) OF
      0: TEMP_dest[127:0] ← SRC2[127:0]
      1: TEMP_dest[255:128] ← SRC2[127:0]
   ESAC.
   FI;
IF VL = 512
   CASE (imm8[1:0]) OF
      00: TEMP_dest[127:0] ← SRC2[127:0]
      01: TEMP_dest[255:128] ← SRC2[127:0]
      10: TEMP_dest[383:256] ← SRC2[127:0]
      11: TEMP_dest[511:384] ← SRC2[127:0]
   ESAC.
   FI;
FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ← TEMP_dest[i+63:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+63:i] ← 0
   FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
IF *merging-masking*  ; merging-masking
  THEN *DEST[i+63:i] remains unchanged*
ELSE  ; zeroing-masking
  DEST[i+63:i] \leftarrow 0
FI

FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0

VINSERTF32x8 (EVEX.U1.512 encoded version)
TEMP_DEST[VL-1:0] \leftarrow SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] \leftarrow SRC2[255:0]
  1: TMP_DEST[511:256] \leftarrow SRC2[255:0]
ESAC.

FOR j \leftarrow 0 TO 15
  i \leftarrow j \times 32
  IF k1[j] OR *no writemask*  
    THEN DEST[i+31:i] \leftarrow TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*  ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE  ; zeroing-masking
      DEST[i+31:i] \leftarrow 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0

VINSERTF64x4 (EVEX.512 encoded version)
VL = 512
TEMP_DEST[VL-1:0] \leftarrow SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] \leftarrow SRC2[255:0]
  1: TMP_DEST[511:256] \leftarrow SRC2[255:0]
ESAC.

FOR j \leftarrow 0 TO 7
  i \leftarrow j \times 64
  IF k1[j] OR *no writemask*  
    THEN DEST[i+63:i] \leftarrow TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking*  ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE  ; zeroing-masking
      DEST[i+63:i] \leftarrow 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] \leftarrow 0
VINSERTF128 (VEX encoded version)
TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] ← SRC2[127:0]
  1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ← TEMP

Intel C/C++ Compiler Intrinsic Equivalent
VINSERTF32x4 __m512 _mm512_insertf32x4( __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_mask_insertf32x4( __m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_maskz_insertf32x4( __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_insertf32x4( __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_mask_insertf32x4( __m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_maskz_insertf32x4( __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 __m512_insertf32x8( __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_mask_insertf32x8( __m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_maskz_insertf32x8( __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8x2 __m512d _mm512_insertf64x2( __m512d a, __m128d b, int imm);
VINSERTF32x8x2 __m512d _mm512_mask_insertf64x2( __m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF32x8x2 __m512d _mm512_maskz_insertf64x2( __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF32x4 __m256d _mm256_insertf64x4( __m256d a, __m128d b, int imm);
VINSERTF32x4 __m256d _mm256_mask_insertf64x4( __m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF32x4 __m256d _mm256_maskz_insertf64x4( __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF128 __m256i _mm256_insertf128_ps( __m256 a, __m128 b, int offset);
VINSERTF128 __m256d __m256d_insertf128_pd( __m256d a, __m128d b, int offset);
VINSERTF128 __m256d __m256i_insertf128_si256( __m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instruction, see Exceptions Type 6; additionally
#UD If VEX.L = 0.
EVEX-encoded instruction, see Exceptions Type E6NF.
**VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.6F3A.W0 38 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.6F3A.W0 38 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.6F3A.W0 38 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.6F3A.W1 38 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.6F3A.W1 38 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.6F3A.W0 3A /r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.6F3A.W1 3A /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

**VINSERTI32x4 and VINSERTI64x2** inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7 bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

**VINSERTI32x8 and VINSERTI64x4** inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

**VINSERTI128** inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.
Operation

**VINSERTI32x4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256
    CASE (imm8[0]) OF
        0: TEMP_DEST[127:0] ← SRC2[127:0]
        1: TEMP_DEST[255:128] ← SRC2[127:0]
    ESAC.

FI;

IF VL = 512
    CASE (imm8[1:0]) OF
        00: TEMP_DEST[127:0] ← SRC2[127:0]
        01: TEMP_DEST[255:128] ← SRC2[127:0]
        10: TEMP_DEST[383:256] ← SRC2[127:0]
        11: TEMP_DEST[511:384] ← SRC2[127:0]
    ESAC.

FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VINSERTI64x2 (EVEX encoded versions)**

(KL, VL) = (4, 256), (8, 512)

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256
    CASE (imm8[0]) OF
        0: TEMP_DEST[127:0] ← SRC2[127:0]
        1: TEMP_DEST[255:128] ← SRC2[127:0]
    ESAC.

FI;

IF VL = 512
    CASE (imm8[1:0]) OF
        00: TEMP_DEST[127:0] ← SRC2[127:0]
        01: TEMP_DEST[255:128] ← SRC2[127:0]
        10: TEMP_DEST[383:256] ← SRC2[127:0]
        11: TEMP_DEST[511:384] ← SRC2[127:0]
    ESAC.

FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
IF *merging-masking* ; merging-masking
   THEN *DEST[i+63:i] remains unchanged*
   ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
   FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**VINSERTI32x8 (EVEX.U1.512 encoded version)**
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
   0: TMP_DEST[255:0] ← SRC2[255:0]
   1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 15
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+31:i] ← 0
            FI
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**VINSERTI64x4 (EVEX.512 encoded version)**
VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
   0: TMP_DEST[255:0] ← SRC2[255:0]
   1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 7
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+63:i] ← 0
            FI
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
**VINSERTI128**

TEMP[255:0] ← SRC1[255:0]

CASE (imm8[0]) OF

0: TEMP[127:0] ← SRC2[127:0]
1: TEMP[255:128] ← SRC2[127:0]

ESAC

DEST ← TEMP

---

**Intel C/C++ Compiler Intrinsic Equivalent**

- VINSERTI32x4 _mm512i _inserti32x4(__m512i a, __m128i b, int imm);
- VINSERTI32x4 _mm512i _mask_inserti32x4(__m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
- VINSERTI32x4 _mm512i _maskz_inserti32x4(__mmask16 k, __m512i a, __m128i b, int imm);
- VINSERTI32x4 _mm256i _mm256_inserti32x4(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
- VINSERTI32x4 _mm256i _mm256_mask_inserti32x4(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
- VINSERTI32x4 _mm256i _mm256_maskz_inserti32x4(__mmask8 k, __m256i a, __m128i b, int imm);

---

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD  If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.
VMASKMOV—Conditional SIMD Packed Loads and Stores

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2C /r</td>
<td>RVM V/V AVX</td>
<td>Conditionally load packed single-precision values from m128 using mask in xmm2 and store in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2C /r</td>
<td>RVM V/V AVX</td>
<td>Conditionally load packed single-precision values from m256 using mask in ymm2 and store in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2D /r</td>
<td>RVM V/V AVX</td>
<td>Conditionally load packed double-precision values from m128 using mask in xmm2 and store in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2D /r</td>
<td>RVM V/V AVX</td>
<td>Conditionally load packed double-precision values from m256 using mask in ymm2 and store in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2E /r</td>
<td>MVR V/V AVX</td>
<td>Conditionally store packed single-precision values from xmm2 using mask in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2E /r</td>
<td>MVR V/V AVX</td>
<td>Conditionally store packed single-precision values from ymm2 using mask in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2F /r</td>
<td>MVR V/V AVX</td>
<td>Conditionally store packed double-precision values from xmm2 using mask in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2F /r</td>
<td>MVR V/V AVX</td>
<td>Conditionally store packed double-precision values from ymm2 using mask in ymm1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRM:r/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.
In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

**Operation**

**VMASKMOVP - 128-bit load**

DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] ← 0

**VMASKMOVP - 128-bit store**

IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

**VMASKMOVP - 256-bit load**

DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0

**VMASKMOVP - 256-bit store**

IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

**VMASKMOVPD - 128-bit load**

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] ← 0

**VMASKMOVPD - 128-bit store**

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

**VMASKMOVPD - 256-bit load**

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0

**VMASKMOVPD - 256-bit store**

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
VMASKMOVDPD - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

VMASKMOVDPD - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent
__m256 _mm256_maskload_ps(float const *a, __m256i mask)
void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)
__m256d _mm256_maskload_pd(double *a, __m256i mask);
void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);
__m128 _mm_maskload_ps(float const *a, __m128i mask)
void _mm_maskstore_ps(float *a, __m128i mask, __m128 b)
__m128d _mm_maskload_pd(double const *a, __m128i mask);
void _mm_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 6 (No AC# reported for any mask bit combinations);
additionally
#UD If VEX.W = 1.
**VPBLENDD — Blend Packed Dwords**

**Description**

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Operation**

**VPBLENDD (VEX.256 encoded version)**

IF (imm8[0] == 1) THEN DEST[31:0] \( \leftarrow \) SRC2[31:0]
ELSE DEST[31:0] \( \leftarrow \) SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] \( \leftarrow \) SRC2[63:32]
ELSE DEST[63:32] \( \leftarrow \) SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] \( \leftarrow \) SRC2[95:64]
ELSE DEST[95:64] \( \leftarrow \) SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] \( \leftarrow \) SRC2[127:96]
ELSE DEST[127:96] \( \leftarrow \) SRC1[127:96]
IF (imm8[4] == 1) THEN DEST[159:128] \( \leftarrow \) SRC2[159:128]
ELSE DEST[159:128] \( \leftarrow \) SRC1[159:128]
ELSE DEST[191:160] \( \leftarrow \) SRC1[191:160]
ELSE DEST[223:192] \( \leftarrow \) SRC1[223:192]
IF (imm8[7] == 1) THEN DEST[255:224] \( \leftarrow \) SRC2[255:224]
ELSE DEST[255:224] \( \leftarrow \) SRC1[255:224]
VPBLEND (VEX.128 encoded version)
IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
ELSE DEST[95:64] ← SRC1[95:64]
ELSE DEST[127:96] ← SRC1[127:96]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VPBLEND: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)
VPBLEND: __m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 4; additionally
#UD If VEX.W = 1.
VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 66 /r VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 66 /r VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 66 /r VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 66 /r VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 66 /r VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).
**Operation**

**VPBLENDMB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC2[i+7:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN DEST[i+7:i] ← SRC1[i+7:i]
        ELSE ; zeroing-masking
            DEST[i+7:i] ← 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;

**VPBLENDMW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC2[i+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN DEST[i+15:i] ← SRC1[i+15:i]
        ELSE ; zeroing-masking
            DEST[i+15:i] ← 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBLENDMB __m512i _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.
VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 64 /r VPBLENDMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.
Operation

VPBLENDMD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\( i \leftarrow j \times 32 \)

IF \( k1[j] \) OR *no controlmask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

\( \text{DEST}[i+31:i] \leftarrow \text{SRC2}[31:0] \)

ELSE

\( \text{DEST}[i+31:i] \leftarrow \text{SRC2}[i+31:i] \)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN \( \text{DEST}[i+31:i] \leftarrow \text{SRC1}[i+31:i] \)

ELSE ; zeroing-masking

\( \text{DEST}[i+31:i] \leftarrow 0 \)

FI;

FI;

ENDFOR

\( \text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0; \)

VPBLENDMD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\( i \leftarrow j \times 32 \)

IF \( k1[j] \) OR *no controlmask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

\( \text{DEST}[i+31:i] \leftarrow \text{SRC2}[31:0] \)

ELSE

\( \text{DEST}[i+31:i] \leftarrow \text{SRC2}[i+31:i] \)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN \( \text{DEST}[i+31:i] \leftarrow \text{SRC1}[i+31:i] \)

ELSE ; zeroing-masking

\( \text{DEST}[i+31:i] \leftarrow 0 \)

FI;

FI;

ENDFOR

\( \text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0; \)
Intel C/C++ Compiler Intrinsic Equivalent
VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E4.
VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Description</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 {k1}{z}, r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64</td>
<td>A</td>
<td>V.N.E.</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64</td>
<td>A</td>
<td>V.N.E.</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64</td>
<td>A</td>
<td>V.N.E.</td>
<td>AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Notes:
1. EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:rr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPBROADCASTB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
   i ← j * 8
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] ← SRC[7:0]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+7:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+7:i] ← 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPBROADCASTW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] ← SRC[15:0]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+15:i] ← 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPBROADCASTD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← SRC[31:0]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] ← 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPBROADCASTQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ← SRC[63:0]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+63:i] ← 0
   FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB __m512i _mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i _mm512_maskz_set1_epi8( __mmask64 k, int a);
VPBROADCASTB __m256i _mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i _mm256_maskz_set1_epi8( __mmask32 k, int a);
VPBROADCASTB __m128i _mm_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i _mm_maskz_set1_epi8( __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_mask_set1_epi32(__m512i s, __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_maskz_set1_epi32( __mmask16 k, int a);
VPBROADCASTD __m256i _mm256_mask_set1_epi32(__m256i s, __mmask8 k, int a);
VPBROADCASTD __m256i _mm256_maskz_set1_epi32( __mmask8 k, int a);
VPBROADCASTD __m128i _mm_mask_set1_epi32(__m128i s, __mmask8 k, int a);
VPBROADCASTD __m128i _mm_maskz_set1_epi32( __mmask8 k, int a);
VPBROADCASTQ __m512i _mm512_mask_set1_epi64(__m512i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m512i _mm512_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_mask_set1_epi64(__m256i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_mask_set1_epi64(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_maskz_set1_epi64( __mmask8 k, __int64 a);

Exceptions

EVEX-encoded instructions, see Exceptions Type E7NM.

#UD If EVEX.vvvv != 1111B.
## VPBROADCAST—Load Integer and Broadcast

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 78 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a byte integer in the source operand to sixteen locations in xmm1.</td>
</tr>
<tr>
<td>VPBROADCASTB xmm1, xmm2/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 78 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a byte integer in the source operand to thirty-two locations in ymm1.</td>
</tr>
<tr>
<td>VPBROADCASTB ymm1, xmm2/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 78 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTB xmm1[k1]{z}, xmm2/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 78 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTB ymm1[k1]{z}, xmm2/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 78 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTB zmm1{k1}{z}, xmm2/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a word integer in the source operand to eight locations in xmm1.</td>
</tr>
<tr>
<td>VPBROADCASTW xmm1, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a word integer in the source operand to sixteen locations in ymm1.</td>
</tr>
<tr>
<td>VPBROADCASTW ymm1, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 79 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTW xmm1[k1]{z}, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 79 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTW ymm1[k1]{z}, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 79 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTW zmm1{k1}{z}, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 58 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a dword integer in the source operand to four locations in xmm1.</td>
</tr>
<tr>
<td>VPBROADCASTD xmm1, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 58 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a dword integer in the source operand to eight locations in ymm1.</td>
</tr>
<tr>
<td>VPBROADCASTD ymm1, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 58 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTD xmm1[k1]{z}, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 58 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTD ymm1[k1]{z}, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 58 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTD zmm1[k1]{z}, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 59 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a qword element in source operand to two locations in xmm1.</td>
</tr>
<tr>
<td>VPBROADCASTQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 59 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a qword element in source operand to four locations in ymm1.</td>
</tr>
<tr>
<td>VPBROADCASTQ ymm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 59 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTQ xmm1[k1]{z}, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 59 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTQ ymm1[k1]{z}, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 59 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTQ zmm1[k1]{z}, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 59 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VPBROADCASTQ12x2 xmm1 [k1]{z}, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5-304 Vol. 2C  VPBROADCAST—Load Integer and Broadcast
Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

**VEX256-encoded VPBROADCASTB/W/D/Q:** The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register.

**VPBROADCASTI128 support:** The source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

**EVEX-encoded VPBROADCASTD/Q:** The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

**VPBROADCASTI32X4 and VPBROADCASTI64X4:** The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.

---

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple2</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple4</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>Tuple8</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

**VEX256-encoded VPBROADCASTB/W/D/Q:** The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

**EVEX-encoded VPBROADCASTD/Q:** The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

**VPBROADCASTI32X4 and VPBROADCASTI64X4:** The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.
Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD. If VPBROADCAST128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)**

**Figure 5-17. VPBROADCASTD Operation (128-bit version)**

**Figure 5-18. VPBROADCASTQ Operation (256-bit version)**
Operation

VPBROADCASTB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[7:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPBROADCAST (EVEX encoded versions)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \( j \leftarrow 0 \) TO \( KL-1 \)
\[
  i \leftarrow j \times 16
\]
IF \( k1[j] \) OR *no writemask*
THEN \( \text{DEST}[i+15:i] \leftarrow \text{SRC}[15:0] \)
ELSE
  IF *merging-masking* ; merging-masking
    THEN *\text{DEST}[i+15:i] remains unchanged* 
    ELSE ; zeroing-masking
      \( \text{DEST}[i+15:i] \leftarrow 0 \)
    FI
  FI;
ENDFOR
\( \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \)

VPBROADCAST (128 bit version)

\( \text{temp} \leftarrow \text{SRC}[31:0] \)
\( \text{DEST}[31:0] \leftarrow \text{temp} \)
\( \text{DEST}[63:32] \leftarrow \text{temp} \)
\( \text{DEST}[95:64] \leftarrow \text{temp} \)
\( \text{DEST}[127:96] \leftarrow \text{temp} \)
\( \text{DEST}[\text{MAXVL}-1:128] \leftarrow 0 \)

VPBROADCAST (VEX.256 encoded version)

\( \text{temp} \leftarrow \text{SRC}[31:0] \)
\( \text{DEST}[31:0] \leftarrow \text{temp} \)
\( \text{DEST}[63:32] \leftarrow \text{temp} \)
\( \text{DEST}[95:64] \leftarrow \text{temp} \)
\( \text{DEST}[127:96] \leftarrow \text{temp} \)
\( \text{DEST}[191:160] \leftarrow \text{temp} \)
\( \text{DEST}[223:192] \leftarrow \text{temp} \)
\( \text{DEST}[255:224] \leftarrow \text{temp} \)
\( \text{DEST}[\text{MAXVL}-1:256] \leftarrow 0 \)

VPBROADCAST (EVEX encoded versions)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \( j \leftarrow 0 \) TO \( KL-1 \)
\[
  i \leftarrow j \times 32
\]
IF \( k1[j] \) OR *no writemask*
THEN \( \text{DEST}[i+31:i] \leftarrow \text{SRC}[31:0] \)
ELSE
  IF *merging-masking* ; merging-masking
    THEN *\text{DEST}[i+31:i] remains unchanged* 
    ELSE ; zeroing-masking
      \( \text{DEST}[i+31:i] \leftarrow 0 \)
    FI
  FI;
ENDFOR
\( \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \)
VPBROADCASTQ (VEX.256 encoded version)

\[
\text{temp} \leftarrow \text{SRC}[63:0] \\
\text{DEST}[63:0] \leftarrow \text{temp} \\
\text{DEST}[127:64] \leftarrow \text{temp} \\
\text{DEST}[191:128] \leftarrow \text{temp} \\
\text{DEST}[255:192] \leftarrow \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0
\]

VPBROADCASTQ (EVEX encoded versions)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
\quad \text{i} \leftarrow j * 64 \\
\quad \text{IF } k1[j] \text{ OR *no writemask*} \\
\quad \quad \text{THEN DEST}[i+63:i] \leftarrow \text{SRC}[63:0] \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{IF *merging-masking*; merging-masking} \\
\quad \quad \quad \quad \quad \text{THEN *DEST}[i+63:i] \text{ remains unchanged*} \\
\quad \quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \quad \text{DEST}[i+63:i] \leftarrow 0 \\
\quad \quad \quad \text{FI} \\
\quad \quad \text{FI}; \\
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0
\]

VBROADCASTI32x2 (EVEX encoded versions)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
\quad \text{i} \leftarrow j * 32 \\
\quad \text{n} \leftarrow (j \text{ mod 2}) * 32 \\
\quad \text{IF } k1[j] \text{ OR *no writemask*} \\
\quad \quad \text{THEN DEST}[i+31:i] \leftarrow \text{SRC}[n+31:n] \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{IF *merging-masking*; merging-masking} \\
\quad \quad \quad \quad \quad \text{THEN *DEST}[i+31:i] \text{ remains unchanged*} \\
\quad \quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \quad \text{DEST}[i+31:i] \leftarrow 0 \\
\quad \quad \quad \text{FI} \\
\quad \quad \text{FI}; \\
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0
\]

VBROADCASTI128 (VEX.256 encoded version)

\[
\text{temp} \leftarrow \text{SRC}[127:0] \\
\text{DEST}[127:0] \leftarrow \text{temp} \\
\text{DEST}[255:128] \leftarrow \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0
\]
VPBROADCASTI32X4 (EVEX encoded versions)
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    n ← (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPBROADCASTI64X2 (EVEX encoded versions)
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    n ← (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR;

VPBROADCASTI32X8 (EVEX.U1.512 encoded version)
FOR j ← 0 TO 15
    i ← j * 32
    n ← (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VBROADCASTI64X4 (EVEX.512 encoded version)

FOR j ← 0 TO 7
    i ← j * 64
    n ← (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
        IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB __m512i _mm512_broadcastb_epi8( __m128i a);
VPBROADCASTB __m512i _mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i _mm512_maskz_broadcastb_epi8( __mmask64 k, __m128i a);
VPBROADCASTB __m256i _mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i _mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i _mm256_maskz_broadcastb_epi8( __mmask32 k, __m128i a);
VPBROADCASTB __m128i _mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_maskz_broadcastb_epi8( __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_broadcastb_epi8(__m128i a);

VPBROADCASTD __m512i _mm512_broadcastd_epi32( __m128i a);
VPBROADCASTD __m512i _mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i _mm512_maskz_broadcastd_epi32( __mmask16 k, __m128i a);
VPBROADCASTD __m256i _mm256_broadcastd_epi32( __m128i a);
VPBROADCASTD __m256i _mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i _mm256_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_broadcastd_epi32(__m128i a);

VPBROADCASTQ __m512i _mm512_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m512i _mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m256i _mm256_mask_broadcastq_epi64(__m256i s, __mmask4 k, __m128i a);
VPBROADCASTQ __m256i _mm256_maskz_broadcastq_epi64( __mmask4 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64(__m128i s, __mmask4 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64( __mmask4 k, __m128i a);

VPBROADCASTW __m512i _mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i _mm512_mask_broadcastw_epi16(__m512i s, __mmask4 k, __m128i a);
VPBROADCASTW __m512i _mm512_maskz_broadcastw_epi16( __mmask4 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i _mm256_mask_broadcastw_epi16(__m256i s, __mmask2 k, __m128i a);
VPBROADCASTW __m256i _mm256_maskz_broadcastw_epi16( __mmask2 k, __m128i a);
VPBROADCASTW __m128i _mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i _mm_mask_broadcastw_epi16(__m128i s, __mmask2 k, __m128i a);
VPBROADCASTW __m128i _mm_maskz_broadcastw_epi16( __mmask2 k, __m128i a);

VPBROADCASTI32x2 __m512i _mm512_broadcast_i32x2( __m128i a);
INSTRUCTION SET REFERENCE, V-Z

VPBROADCAST—I32x2

VBROADCASTI32x2 __m512i _mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBroadcastI32x2 __m512i _mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBroadcastI32x2 __m256i _mm256_mask广播_i32x2(__m256i s, __mmask8 k, __m128i a);
VBroadcastI32x2 __m256i _mm256_maskz广播_i32x2(__mmask8 k, __m128i a);
VBroadcastI32x2 __m128i _mm广播_i32x2(__m128i a);
VBroadcastI32x2 __m128i _mm_mask广播_i32x2(__m128i s, __mmask8 k, __m128i a);
VBroadcastI32x2 __m128i _mm_maskz广播_i32x2(__mmask8 k, __m128i a);

VBroadcastI32x4 __m512i _mm512广播_i32x4(__m128i a);
VBroadcastI32x4 __m512i _mm512_mask广播_i32x4(__m512i s, __mmask16 k, __m128i a);
VBroadcastI32x4 __m512i _mm512_maskz广播_i32x4(__mmask16 k, __m128i a);
VBroadcastI32x4 __m256i _mm256广播_i32x4(__m256i s, __mmask8 k, __m128i a);
VBroadcastI32x4 __m256i _mm256_mask广播_i32x4(__mmask8 k, __m128i a);
VBroadcastI32x4 __m256i _mm256_maskz广播_i32x4(__mmask8 k, __m128i a);
VBroadcastI32x8 __m128i _mm广播_i32x8(__m128i a);
VBroadcastI32x8 __m128i _mm_mask广播_i32x8(__m128i s, __mmask16 k, __m256i a);
VBroadcastI32x8 __m128i _mm_maskz广播_i32x8(__mmask16 k, __m256i a);
VBroadcastI64x2 __m512i _mm512广播_i64x2(__m128i a);
VBroadcastI64x2 __m512i _mm512_mask广播_i64x2(__m512i s, __mmask8 k, __m128i a);
VBroadcastI64x2 __m512i _mm512_maskz广播_i64x2(__mmask8 k, __m128i a);
VBroadcastI64x2 __m256i _mm256广播_i64x2(__m256i s, __mmask8 k, __m128i a);
VBroadcastI64x2 __m256i _mm256_mask广播_i64x2(__mmask8 k, __m128i a);
VBroadcastI64x2 __m256i _mm256_maskz广播_i64x2(__mmask8 k, __m128i a);
VBroadcastI64x4 __m128i _mm广播_i64x4(__m128i a);
VBroadcastI64x4 __m128i _mm_mask广播_i64x4(__m128i s, __mmask8 k, __m256i a);
VBroadcastI64x4 __m128i _mm_maskz广播_i64x4(__mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
If EVEX.L’L’ = 0 for VPBROADCASTI32X4/VPBROADCASTI64X2.
If EVEX.L’L’ < 10b for VPBROADCASTI32X8/VPBROADCASTI64X4.
VPBROADCASTM—Broadcast Mask to Vector Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low byte value in k1 to two locations in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low byte value in k1 to four locations in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1</td>
<td>RM V/V</td>
<td>AVX512CD</td>
<td>Broadcast low byte value in k1 to eight locations in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low word value in k1 to four locations in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low word value in k1 to eight locations in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1</td>
<td>RM V/V</td>
<td>AVX512CD</td>
<td>Broadcast low word value in k1 to sixteen locations in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128). EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

**VPBROADCASTMB2Q**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j*64
  DEST[i+63:i] ← ZeroExtend(SRC[7:0])
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPBROADCASTMW2D**

(KL, VL) = (4, 128), (8, 256),(16, 512)

FOR j ← 0 TO KL-1
  i ← j*32
  DEST[i+31:i] ← ZeroExtend(SRC[15:0])
ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q __m512i __mm512_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m512i __mm512_broadcastmw_epi32(__mmask16);
VPBROADCASTMB2Q __m256i __mm256_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m256i __mm256_broadcastmw_epi32(__mmask8);
VPBROADCASTMB2Q __m128i __mm128_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m128i __mm128_broadcastmw_epi32(__mmask8);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E6NF.
### VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 /r ib</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 /r ib</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 /r ib</td>
<td>A/V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 /r ib</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 /r ib</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 /r ib</td>
<td>A/V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.
The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-8.

<table>
<thead>
<tr>
<th>Operation</th>
<th>PCMPM Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPCMPB/EVEX encoded versions</td>
<td></td>
</tr>
<tr>
<td>(KL, VL) = (16, 128), (32, 256), (64, 512)</td>
<td></td>
</tr>
<tr>
<td>FOR j ← 0 TO KL-1</td>
<td></td>
</tr>
<tr>
<td>i ← j * 8</td>
<td></td>
</tr>
<tr>
<td>IF k2[j] OR <em>no writemask</em></td>
<td></td>
</tr>
<tr>
<td>THEN</td>
<td></td>
</tr>
<tr>
<td>CMP ← SRC1[i+7:j] OP SRC2[i+7:j];</td>
<td></td>
</tr>
<tr>
<td>IF CMP = TRUE</td>
<td></td>
</tr>
<tr>
<td>THEN DEST[j] ← 1;</td>
<td></td>
</tr>
<tr>
<td>ELSE DEST[j] ← 0; Fi;</td>
<td></td>
</tr>
<tr>
<td>ELSE DEST[j] = 0;</td>
<td></td>
</tr>
<tr>
<td>; zeroing-masking only Fi;</td>
<td></td>
</tr>
<tr>
<td>Fi;</td>
<td></td>
</tr>
<tr>
<td>ENDFOR</td>
<td></td>
</tr>
<tr>
<td>DEST[MAX_KL-1:KL] ← 0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5-8. Pseudo-Op and VPCMP* Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>PCMPM Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPCMPB* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VPCMPB* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VPCMPB* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VPCMPB* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VPCMPB* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VPCMPB* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 6</td>
</tr>
</tbody>
</table>
VPCMPUB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k2[j] OR *no writemask*
    THEN
      CMP ← SRC1[i+7:i] OP SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] = 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR

DEST[MAX_KL-1:KL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPB __mmask64 _mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __mmask64 _mm512_mask_cmp_epi8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask32 _mm256_mask_cmp_epi8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 _mm_cmp_epi8_mask( __m128i a, __m128i b, int cmp);
VPCMPB __mmask16 _mm_mask_cmp_epi8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m512i a, __m512i b);
VPCMPB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m256i a, __m256i b);
VPCMPB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m128i a, __m128i b);
VPCMPB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask16 m, __m128i a, __m128i b);
VPCMPB __mmask64 _mm512_cmp_epi8_mask( __m512i a, __m512i b);
VPCMPB __mmask64 _mm512_mask_cmp_epi8_mask( __mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b);
VPCMPB __mmask32 _mm256_mask_cmp_epi8_mask( __mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 _mm_cmp_epi8_mask( __m128i a, __m128i b);
VPCMPB __mmask16 _mm_mask_cmp_epi8_mask( __mmask16 m, __m128i a, __m128i b);
VPCMPUB __mmask64 _mm512_cmp_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPUB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.
VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 1F /r ib</td>
<td>A/V/V</td>
<td>V/X/VX512VL,AVX512F</td>
<td>Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 1F /r ib</td>
<td>A/V/V</td>
<td>AVX512VL,AVX512F</td>
<td>Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 1F /r ib</td>
<td>A/V/V</td>
<td>AVX512F</td>
<td>Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 1E /r ib</td>
<td>A/V/V</td>
<td>V/X/VX512VL,AVX512F</td>
<td>Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 1E /r ib</td>
<td>A/V/V</td>
<td>AVX512VL,AVX512F</td>
<td>Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 1E /r ib</td>
<td>A/V/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (8-bit immediate) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-8.
**Operation**

**CASE (COMPARISON PREDICATE) OF**

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← FALSE;
- 4: OP ← NEQ;
- 5: OP ← NLT;
- 6: OP ← NLE;
- 7: OP ← TRUE;

**ESAC**:

**VPCMPD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j * 32

IF k2[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX_KL-1:KL] ← 0

**VPCMPUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j * 32

IF k2[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX_KL-1:KL] ← 0
INSTRUCTION SET REFERENCE, V-Z

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPD __mmask16 _mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 _mm256_cmp_epi32_mask(__m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m256i a, __m256i b);
VPCMPD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_cmp_epi32_mask(__m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 _mm_cmp_epi32_mask(__m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m128i a, __m128i b);
VPCMPD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp_epi32_mask(__m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m128i a, __m128i b);
VPCMPUD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.
VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-8.
Operation
CASE (COMPARISON PREDICATE) OF
0: OP ← EQ;
1: OP ← LT;
2: OP ← LE;
3: OP ← FALSE;
4: OP ← NEQ;
5: OP ← NLT;
6: OP ← NLE;
7: OP ← TRUE;
ESAC;

VPCMPQ (EVEX encoded versions)
KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];
        ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPCMPUQ (EVEX encoded versions)
KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];
        ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPQ __mmask8 _mm512_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);

VPCMPQ __mmask8 _mm256_cmp_epi64_mask( __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m256i a, __m256i b);
VPCMPQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_cmp_epi64_mask( __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);

VPCMPQ __mmask8 _mm_cmp_epi64_mask( __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_cmp_epi64_mask( __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.
VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

**VPCMPW** performs a comparison between pairs of signed word values.

**VPCMPUW** performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-8.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 3F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>VPCMPW k1 [k2], xmm2, xmm3/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 3F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>VPCMPW k1 [k2], ymm2, ymm3/m256, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 3F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>VPCMPW k1 [k2], zmm2, zmm3/m512, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 3E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>VPCMPUW k1 [k2], xmm2, xmm3/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 3E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>VPCMPUW k1 [k2], ymm2, ymm3/m256, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 3E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>VPCMPUW k1 [k2], zmm2, zmm3/m512, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP ← EQ;
  1: OP ← LT;
  2: OP ← LE;
  3: OP ← FALSE;
  4: OP ← NEQ;
  5: OP ← NLT;
  6: OP ← NLE;
  7: OP ← TRUE;
ESAC;

VPCMPw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      ICMP ← SRC1[i+15:i] OP SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE
        DEST[j] = 0 ; zeroing-masking only
      FI;
  ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPCMPuW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      CMP ← SRC1[i+15:i] OP SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE
        DEST[j] = 0 ; zeroing-masking only
      FI;
  ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPW __mmask32 __mm512_cmp_epi16_mask( __m512i a, __m512i b, int cmp);
VPCMPW __mmask32 __mm512_mask_cmp_epi16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 __mm256_cmp_epi16_mask( __m256i a, __m256i b, int cmp);
VPCMPW __mmask16 __mm256_mask_cmp_epi16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 __mm_cmp_epi16_mask( __m128i a, __m128i b, int cmp);
VPCMPW __mmask8 __mm_mask_cmp_epi16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m512i a, __m512i b);
VPCMPW __mmask32 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m256i a, __m256i b);
VPCMPW __mmask16 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m128i a, __m128i b);
VPCMPW __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 __mm512_cmp_epi16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp_epi16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp_epi16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp_epi16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp_epi16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp_epi16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.
VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 [k1][z], xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed doubleword integer values from xmm2 to xmm1/m128 using controlmask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 [k1][z], ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed doubleword integer values from ymm2 to ymm1/m256 using controlmask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 [k1][z], zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed doubleword integer values from zmm2 to zmm1/m512 using controlmask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

VPCOMPRESSD (EVEX encoded versions) store form

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

i ← j * 32

IF k1[j] OR *no controlmask*

THEN

DEST[k+SIZE-1:k] ← SRC[i+31:i]

k ← k + SIZE

FI;

ENDFOR;

VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register
VPCOMPRESSD (EVEX encoded versions) reg-reg form

(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no controlmask*
  THEN
    DEST[k+SIZE-1:k] ← SRC[i+31:i]
    k ← k + SIZE
  FI;
ENDFOR
IF *merging-masking*
  THEN *DEST[VL-1:k] remains unchanged*
  ELSE DEST[VL-1:k] ← 0
FI
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32( __mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32( __mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32( __mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.
VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 8B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compress packed quadword integer values from xmm2 to xmm1/m128 using controlmask k1.</td>
</tr>
<tr>
<td>VPCOMPRESSQ xmm1/m128 (k1)[z], xmm2</td>
<td></td>
<td></td>
<td>AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 8B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compress packed quadword integer values from ymm2 to ymm1/m256 using controlmask k1.</td>
</tr>
<tr>
<td>VPCOMPRESSQ ymm1/m256 (k1)[z], ymm2</td>
<td></td>
<td></td>
<td>AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 8B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed quadword integer values from zmm2 to zmm1/m512 using controlmask k1.</td>
</tr>
<tr>
<td>VPCOMPRESSQ zmm1/m512 (k1)[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VPCOMPRESSQ (EVEX encoded versions) store form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

\[ \text{SIZE} \leftarrow 64 \]

\[ k \leftarrow 0 \]

FOR \( j \leftarrow 0 \) TO KL-1

\[ i \leftarrow j \times 64 \]

IF \( k1[j] \) OR *no controlmask* THEN

\[ \text{DEST}[k+\text{SIZE}-1:k] \leftarrow \text{SRC}[i+63:i] \]

\[ k \leftarrow k + \text{SIZE} \]

FI;

ENFOR
VPCOMPRESSQ (EVEX encoded versions) reg-reg form

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[ \text{SIZE} \leftarrow 64 \]

\[ k \leftarrow 0 \]

\[ \text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]

\[ i \leftarrow j \ast 64 \]

\[ \text{IF } k1[j] \text{ OR } \text{no controlmask} \]

\[ \text{THEN} \]

\[ \text{DEST}[k+\text{SIZE}-1:k] \leftarrow \text{SRC}[i+63:i] \]

\[ k \leftarrow k + \text{SIZE} \]

\[ \text{FI}; \]

\[ \text{ENDIF} \]

\[ \text{IF } *\text{merging-masking}* \]

\[ \text{THEN } *\text{DEST}[VL-1:k] \text{ remains unchanged}* \]

\[ \text{ELSE } \text{DEST}[VL-1:k] \leftarrow 0 \]

\[ \text{FI} \]

\[ \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VPCOMPRESSQ} & \quad \text{\_mm512i} \quad \text{\_mm512_mask_compress_epi64}(__\text{m512i} \ s, __\text{mmask8} \ c, __\text{m512i} \ a); \\
\text{VPCOMPRESSQ} & \quad \text{\_mm512i} \quad \text{\_mm512_maskz_compress_epi64}(__\text{mmask8} \ c, __\text{m512i} \ a); \\
\text{VPCOMPRESSQ} & \quad \text{void} \quad \text{\_mm512_mask_compressstoreu_epi64}(__\text{m512i} \ s); \\
\text{VPCOMPRESSQ} & \quad \text{\_mm256i} \quad \text{\_mm256_mask_compress_epi64}(__\text{m256i} \ s, __\text{mmask8} \ c, __\text{m256i} \ a); \\
\text{VPCOMPRESSQ} & \quad \text{\_mm256i} \quad \text{\_mm256_maskz_compress_epi64}(__\text{mmask8} \ c, __\text{m256i} \ a); \\
\text{VPCOMPRESSQ} & \quad \text{void} \quad \text{\_mm256_mask_compressstoreu_epi64}(__\text{m256i} \ s); \\
\text{VPCOMPRESSQ} & \quad \text{\_mm128i} \quad \text{\_mm128_mask_compress_epi64}(__\text{m128i} \ s, __\text{mmask8} \ c, __\text{m128i} \ a); \\
\text{VPCOMPRESSQ} & \quad \text{\_mm128i} \quad \text{\_mm128_maskz_compress_epi64}(__\text{mmask8} \ c, __\text{m128i} \ a); \\
\text{VPCOMPRESSQ} & \quad \text{void} \quad \text{\_mm128_mask_compressstoreu_epi64}(__\text{m128i} \ s); \\
\end{align*}
\]

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.
VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 C4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 C4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 C4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 C4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element’s comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

**VPCONFLICTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j*32
  IF MaskBit(j) OR *no writemask* THEN
    FOR k ← 0 TO j-1
      m ← k*32
      IF ((SRC[i+31:i] = SRC[m+31:m]) THEN
        DEST[i+k] ← 1
      ELSE
        DEST[i+k] ← 0
      FI
    ENDFOR
    DEST[i+31:i+j] ← 0
  ELSE
    IF *merging-masking* THEN
      *DEST[i+31:i] remains unchanged*
    ELSE
      DEST[i+31:i] ← 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPCONFLICTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j*64
  IF MaskBit(j) OR *no writemask* THEN
    FOR k ← 0 TO j-1
      m ← k*64
      IF ((SRC[i+63:i] = SRC[m+63:m]) THEN
        DEST[i+k] ← 1
      ELSE
        DEST[i+k] ← 0
      FI
    ENDFOR
    DEST[i+63:i+j] ← 0
  ELSE
    IF *merging-masking* THEN
      *DEST[i+63:i] remains unchanged*
    ELSE
      DEST[i+63:i] ← 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

- `VPCONFLICTD __m512i _mm512_conflict_epi32(__m512i a);`
- `VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);`
- `VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);`
- `VPCONFLICTQ __m512i _mm512_conflict_epi64(__m512i a);`
- `VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);`
- `VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);`
- `VPCONFLICTD __m256i _mm256_conflict_epi32(__m256i a);`
- `VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);`
- `VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);`
- `VPCONFLICTQ __m256i _mm256_conflict_epi64(__m256i a);`
- `VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);`
- `VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);`
- `VPCONFLICTD __m128i _mm_conflict_epi32(__m128i a);`
- `VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);`
- `VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);`
- `VPCONFLICTQ __m128i _mm_conflict_epi64(__m128i a);`
- `VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);`
- `VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4NF.
VPERM2F128 — Permute Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 06 /r ib VPERM2F128 ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

**Figure 5-21. VPERM2F128 Operation**

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.
Operation

VPERM2F128
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
2: DEST[255:128] ← SRC2[127:0]
ESAC

IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[MAXVL-1:128] ← 0
FI

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128: __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)
VPERM2F128: __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)
VPERM2F128: __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 6; additionally
#UD IF VEX.L = 0
          IF VEX.W = 1.
VPERM2I128 — Permute Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 46 /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute 128-bit integer data in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.</td>
</tr>
<tr>
<td>VPERM2I128 ymm1, ymm2, ymm3/m256, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Permute 128-bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

**Figure 5-22. VPERM2I128 Operation**

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.
Operation

VPERM2I128

CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
2: DEST[255:128] ← SRC2[127:0]
ESAC

IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[255:128] ← 0
FI

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD If VEX.L = 0,
If VEX.W = 1.
VPERMB—Permute Packed Bytes Elements

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 8D /r VPERMB xmm1[k1]{k}z, xmm2, xmm3/m128</td>
<td>A/V</td>
<td>V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 8D /r VPERMB ymm1[k1]{k}z, ymm2, ymm3/m256</td>
<td>A/V</td>
<td>V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 8D /r VPERMB zmm1[k1]{k}z, zmm2, zmm3/m512</td>
<td>A/V</td>
<td>V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM[reg (w)]</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

### Operation

**VPERMB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

n ← 3;
ELSE IF VL = 256:

n ← 4;
ELSE IF VL = 512:

n ← 5;
FI;

FOR j ← 0 TO KL-1:

id ← SRC1[*8 + n :*8]; // location of the source byte

IF k1[j] OR *no writemask* THEN

DEST[*8 + 7:j]*8 ← SRC2[id*8 +7: id*8];
ELSE IF zeroing-masking THEN

DEST[*8 + 7:j]*8 ← 0;
*ELSE

DEST[*8 + 7:j]*8 remains unchanged*

FI

ENDFOR

DEST[MAX_VL-1:VL] ← 0;

### Intel C/C++ Compiler Intrinsic Equivalent

VPERMB __m512i _mm512_permutexvar_epi8(__m512i idx, __m512i a);
VPERMB __m512i __mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);
VPERMB __m512i __mm512_maskz_permutexvar_epi8( __mmask64 k, __m512i idx, __m512i a);
VPERMB __m256i __mm256_permutexvar_epi8( __m256i idx, __m256i a);
VPERMB __m256i __mm256_mask_permutexvar_epi8( __m256i s, __mmask32 k, __m256i idx, __m256i a);
VPERMB __m256i __mm256_maskz_permutexvar_epi8( __mmask32 k, __m256i idx, __m256i a);
VPERMB __m128i __mm128_permutexvar_epi8( __m128i idx, __m128i a);
VPERMB __m128i __mm128_mask_permutexvar_epi8( __m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i __mm128_maskz_permutexvar_epi8( __mmask16 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type E4NF.nb.
### VPERMD/VPERMW—Permuted Packed Doublewords/Words Elements

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 36 /r VPERMD zmm1 [k1]z, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 8D /r VPERMW xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 8D /r VPERMW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 8D /r VPERMW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

**VEX.256 encoded VPERMD:** The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

**EVEX encoded VPERMD:** The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

**VPERMW:** first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

**EVEX.128 encoded versions:** Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.
Operation

**VPERMD (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

IF VL = 256 THEN n ← 2; Fl;
IF VL = 512 THEN n ← 3; Fl;
FOR j ← 0 TO KL-1
  i ← j * 32
  id ← 32*SRC1[i+n]
  IF k1[j] OR *no writemask*
      THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← SRC2[31:0];
            ELSE DEST[i+31:i] ← SRC2[id+31:id];
            FI;
      ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
      FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMD (VEX.256 encoded version)**

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] ← 0

**VPERMW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128 THEN n ← 2; Fl;
IF VL = 256 THEN n ← 3; Fl;
IF VL = 512 THEN n ← 4; Fl;
FOR j ← 0 TO KL-1
  i ← j * 16
  id ← 16*SRC1[i+n]
  IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] ← SRC2[id+15:id]
      ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
      FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPERMD __m512i _mm512_permutexvar_epi32( __m512i idx, __m512i a);
VPERMD __m512i _mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i _mm512_maskz_permutexvar_epi32( __mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i _mm256_permutexvar_epi32( __m256i idx, __m256i a);
VPERMD __m256i _mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i _mm256_maskz_permutexvar_epi32( __mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i _mm512_permutexvar_epi16( __m512i idx, __m512i a);
VPERMW __m512i _mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i _mm512_maskz_permutexvar_epi16( __mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i _mm256_permutexvar_epi16( __m256i idx, __m256i a);
VPERMW __m256i _mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i _mm256_maskz_permutexvar_epi16( __mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i _mm_permutexvar_epi16( __m128i idx, __m128i a);
VPERMW __m128i _mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i _mm_maskz_permutexvar_epi16( __mmask8 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPERMD, see Exceptions Type E4NF.
EVEX-encoded VPERMW, see Exceptions Type E4NF.nb.

#UD

If VEX.L = 0.
If EVEX.L'L = 0 for VPERMD.
**VPERM2I2B—Full Permute of Bytes from Two Tables Overwriting the Index**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 75 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 75 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 75 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Permutesthebytevaluesinthesecondoperand (thefirstsourceoperand)andthethirdoperand (thesecondsource operand) using the byte indices in the first operand (thedestinationoperand) to select byte elements from the second or third source operands. Theselectedbyteelementsareshownwriting to the destination at byte granularityunder the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Notethattheseeinstructionspermitebytevalueinthesourceoperandstobecopiedtomorethanone location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed fore VL=256,128.
**Operation**

**VPERMII2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:
   id ← 3;
ELSE IF VL = 256:
   id ← 4;
ELSE IF VL = 512:
   id ← 5;
FI;

TMP_DEST[VL-1:0] ← DEST[VL-1:0];
FOR j ← 0 TO KL-1
   off ← 8*SRC1[j*8 + id: j*8];
   IF k1[j] OR *no writemask*:
   ELSE IF *zeroing-masking*
      DEST[j*8 + 7: j*8] ← 0;
   ELSE
      DEST[j*8 + 7: j*8] remains unchanged*
   FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMII2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i id, __m512i b);
VPERMII2B __m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i id, __mmask64 k, __m512i b);
VPERMII2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i id, __m512i b);
VPERMII2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i id, __m256i b);
VPERMII2B __m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i id, __mmask32 k, __m256i b);
VPERMII2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i id, __m256i b);
VPERMII2B __m128i _mm128i _mm128i_permutex2var_epi8(__m128i a, __m128i id, __m128i b);
VPERMII2B __m128i _mm128i _mm128i_mask2_permutex2var_epi8(__m128i a, __m128i id, __mmask16 k, __m128i b);
VPERMII2B __m128i _mm128i _mm128i_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i id, __m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 75 / r VPERMI2W xmm1[k1][z], xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 75 / r VPERMI2W ymm1[k1][z], ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 75 / r VPERMI2W zmm1[k1][z], zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 76 / r VPERMI2D xmm1[k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 76 / r VPERMI2D ymm1[k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 76 / r VPERMI2D zmm1[k1][z], zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 76 / r VPERMI2Q xmm1[k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 76 / r VPERMI2Q ymm1[k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 76 / r VPERMI2Q zmm1[k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 77 / r VPERMI2PS xmm1[k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 77 / r VPERMI2PS ymm1[k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 77 / r VPERMI2PS zmm1[k1][z], zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands. If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.
Operation

VPERM2W (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
   id ← 2
FI;
IF VL = 256
   id ← 3
FI;
IF VL = 512
   id ← 4
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
   i ← j * 16
   off ← 16*TMP_DEST[i+id]
   IF k1[j] OR *no writemask*
      THEN
         DEST[i+15:i]=TMP_DEST[i+id+1] ? SRC2[off+15:off]
            : SRC1[off+15:off]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+15:i] ← 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPERM2D/VPERM2PS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
   id ← 1
FI;
IF VL = 256
   id ← 2
FI;
IF VL = 512
   id ← 3
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
   i ← j * 32
   off ← 32*TMP_DEST[i+id]
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
               DEST[i+31:i] ← TMP_DEST[i+id+1] ? SRC2[31:0]
                  : SRC1[off+31:off]
            ELSE
               DEST[i+31:i] ← TMP_DEST[i+id+1] ? SRC2[off+31:off]
                  : SRC1[off+31:off]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
  ELSE ; zeroing-masking
    DEST[i+31:i] ← 0
  FI
FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPERMI2Q/VPERMI2PD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
  id ← 0
FI;
IF VL = 256
  id ← 1
FI;
IF VL = 512
  id ← 2
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
  i ← j * 64
  off ← 64*TMP_DEST[i+id]
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← TMP_DEST[i+id+1] ? SRC2[63:0]
          : SRC1[off+63:off]
        ELSE
          DEST[i+63:i] ← TMP_DEST[i+id+1] ? SRC2[off+63:off]
          : SRC1[off+63:off]
      FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPERM2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERM2D __m512i _mm512_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERM2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERM2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERM1 __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERM2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i idx, __m256i b);
VPERM2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i idx, __m256i b);
VPERM2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERM2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i idx, __m128i b);
VPERM2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i idx, __m128i b);
VPERM2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERM2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i idx, __m128i b);
VPERM2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERM2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512d idx, __m512d b);
VPERM2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512d idx, __m512d b);
VPERM2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512d idx, __mmask8 k, __m512d idx, __m512d b);
VPERM2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512d idx, __m512d b);
VPERM2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256d idx, __m256d b);
VPERM2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256d idx, __m256d b);
VPERM2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256d idx, __mmask8 k, __m256d idx, __m256d b);
VPERM2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256d idx, __m256d b);
VPERM2PD __m128d _mm_permutex2var_pd(__m128d a, __m128d idx, __m128d b);
VPERM2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128d idx, __m128d b);
VPERM2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128d idx, __mmask8 k, __m128d idx, __m128d b);
VPERM2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128d idx, __m128d b);
VPERM2PS __m512 __m512_permutex2var_ps(__m512 a, __m512 idx, __m512 b);
VPERM2PS __m512 __m512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512 idx, __m512 b);
VPERM2PS __m512 __m512_mask2_permutex2var_ps(__m512 a, __m512 idx, __mmask16 k, __m512 idx, __m512 b);
VPERM2PS __m512 __m512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512 idx, __m512 b);
VPERM2PS __m256 __m256_permutex2var_ps(__m256 a, __m256 idx, __m256 b);
VPERM2PS __m256 __m256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256 idx, __m256 b);
VPERM2PS __m256 __m256_mask2_permutex2var_ps(__m256 a, __m256 idx, __mmask8 k, __m256 idx, __m256 b);
VPERM2PS __m256 __m256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256 idx, __m256 b);
VPERM2PS __m128 __m128_permutex2var_ps(__m128 a, __m128 idx, __m128 b);
VPERM2PS __m128 __m128_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128 idx, __m128 b);
VPERM2PS __m128 __m128_mask2_permutex2var_ps(__m128 a, __m128 idx, __mmask8 k, __m128 idx, __m128 b);
VPERM2PS __m128 __m128_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128 idx, __m128 b);
VPERM2Q __m512 __m512_permutex2var_epi64(__m512 a, __m512 idx, __m512 b);
VPERM2Q __m512 __m512_mask_permutex2var_epi64(__m512 a, __mmask16 k, __m512 idx, __m512 b);
VPERM2Q __m512 __m512_mask2_permutex2var_epi64(__m512 a, __m512 idx, __mmask16 k, __m512 idx, __m512 b);
VPERM2Q __m512 __m512_maskz_permutex2var_epi64(__mmask16 k, __m512 a, __m512 idx, __m512 b);
VPERM2Q __m256 __m256_permutex2var_epi64(__m256 a, __m256 idx, __m256 b);
VPERM2Q __m256 __m256_mask_permutex2var_epi64(__m256 a, __mmask8 k, __m256 idx, __m256 b);
VPERM2Q __m256 __m256_mask2_permutex2var_epi64(__m256 a, __m256 idx, __mmask8 k, __m256 idx, __m256 b);
VPERM2Q __m256 __m256_maskz_permutex2var_epi64(__mmask8 k, __m256 a, __m256 idx, __m256 b);
VPERM2Q __m128 __m128_permutex2var_epi64(__m128 a, __m128 idx, __m128 b);
VPERM2Q __m128 __m128_mask_permutex2var_epi64(__m128 a, __mmask8 k, __m128 idx, __m128 b);
VPERM2Q __m128 __m128_mask2_permutex2var_epi64(__m128 a, __m128 idx, __mmask8 k, __m128 idx, __m128 b);
VPERM2Q __m128 __m128_maskz_permutex2var_epi64(__mmask8 k, __m128 a, __m128 idx, __m128 b);
VPERMI2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMI2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMI2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMI2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMI2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VPERMI2D/Q/PS/PD: See Exceptions Type E4NF.
VPERMI2W: See Exceptions Type E4NF.nb.
VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values

### Instruction Set Reference, V-Z

**VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values**

**Opcode/Instruction** | Op / En | 64/32 bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128 | A | V/V | AVX | Permute double-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256 | A | V/V | AVX | Permute double-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.0F38.W1 0D /r VPERMILPD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst | C | V/V | AVX512VL AVX512F | Permute double-precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 0D /r VPERMILPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst | C | V/V | AVX512VL AVX512F | Permute double-precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 0D /r VPERMILPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst | C | V/V | AVX512F | Permute double-precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8 | B | V/V | AVX | Permute double-precision floating-point values in xmm2/m128 using controls from imm8.
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8 | B | V/V | AVX | Permute double-precision floating-point values in ymm2/m256 using controls from imm8.
EVEX.128.66.0F3A.W1 05 /r ib VPERMILPD xmm1 (k1)[z], xmm2/m128/m64bcst, imm8 | D | V/V | AVX512VL AVX512F | Permute double-precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W1 05 /r ib VPERMILPD ymm1 (k1)[z], ymm2/m256/m64bcst, imm8 | D | V/V | AVX512VL AVX512F | Permute double-precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 (k1)[z], zmm2/m512/m64bcst, imm8 | D | V/V | AVX512F | Permute double-precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description

(variable control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

(immediate control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.
Operation

VPERMILPD (EVEX immediate versions)

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j * 64

IF (EVEX.b = 1) AND (SRC1 *is memory*)

THEN TMP_SRC1[i+63:i] ← SRC1[63:0];
ELSE TMP_SRC1[i+63:i] ← SRC1[i+63:i];

FI;

ENDFOR;

IF (imm8[0] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] ← TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] ← TMP_SRC1[63:0]; FI;
IF (imm8[1] = 1) THEN TMP_DEST[127:64] ← TMP_SRC1[127:64]; FI;

IF VL >= 256


FI;

IF VL >= 512


FI;

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

VPERMILPD (256-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]

DEST[MAXVL-1:256] ← 0
VPERMILPD (128-bit immediate version)
IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
DEST[MAXVL-1:128] ← 0

VPERMILPD (EVEX variable versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
  FI;
ENDFOR;

IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; Fl;
IF (TMP_SRC2[1] = 1) THEN TMP_DEST[63:0] ← SRC1[127:64]; Fl;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] ← SRC1[63:0]; Fl;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] ← SRC1[127:64]; Fl;
IF VL >= 256
  FI;
IF VL >= 512
  IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] ← SRC1[319:256]; Fl;
  IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] ← SRC1[319:256]; Fl;
  IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] ← SRC1[447:384]; Fl;
  IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] ← SRC1[447:384]; Fl;
  IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] ← SRC1[511:448]; Fl;
  FI;
ENDFOR
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPERMILPD (256-bit variable version)
IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:256] ← 0

VPERMILPD (128-bit variable version)
IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMILPD __m512d _mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d _mm512_mask_permute_pd( __m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d _mm512_maskz_permute_pd( __mmask8 k, __m512d a, int imm);
VPERMILPD __m256d _mm256_mask_permute_pd( __m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d _mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d _mm_mask_permute_pd( __m128d a, int imm);
VPERMILPD __m128d _mm_maskz_permute_pd( __mmask8 k, __m128d a, int imm);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4; additionally
#UD If VEX.W = 1.
EVEX-encoded instruction, see Exceptions Type E4NF.
#UD If either (E)VEX.vvvv != 1111B and with imm8.
### VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 0C /r VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 04 /r ib VPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
(variable control version)
Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-26). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

(immediate control version)
Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.
Operation

Select4(SRC, control) {
    CASE (control[1:0]) OF
        0: TMP ← SRC[31:0];
        1: TMP ← SRC[63:32];
        2: TMP ← SRC[95:64];
        3: TMP ← SRC[127:96];
    ESAC;
    RETURN TMP
}

VPERMILPS (EVEX immediate versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+31:i] ← SRC1[31:0];
        ELSE TMP_SRC1[i+31:i] ← SRC1[i+31:i];
    FI;
ENDFOR;

TMP_DEST[31:0] ← Select4(TMP_SRC1[127:0], imm8[1:0]);
TMP_DEST[95:64] ← Select4(TMP_SRC1[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC1[127:0], imm8[7:6]); FI;

IF VL >= 256
    TMP_DEST[159:128] ← Select4(TMP_SRC1[255:128], imm8[1:0]); FI;
    FI;

IF VL >= 512
    TMP_DEST[287:256] ← Select4(TMP_SRC1[383:256], imm8[1:0]); FI;
    TMP_DEST[415:384] ← Select4(TMP_SRC1[511:384], imm8[1:0]); FI;
    TMP_DEST[479:448] ← Select4(TMP_SRC1[511:384], imm8[5:4]); FI;
    TMP_DEST[511:480] ← Select4(TMP_SRC1[511:384], imm8[7:6]); FI;
    FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] ← 0 ;zeroing-masking
            FI;
        FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0
VPERMILPS (256-bit immediate version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);

VPERMILPS (128-bit immediate version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128] ← 0

VPERMILPS (EVEX variable versions)
KL, VL = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0];
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i];
  FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] ← Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] ← Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
FI;
IF VL >= 512
  TMP_DEST[287:256] ← Select4(SRC1[383:256], TMP_SRC2[257:256]);
  TMP_DEST[319:288] ← Select4(SRC1[383:256], TMP_SRC2[289:288]);
  TMP_DEST[479:448] ← Select4(SRC1[511:384], TMP_SRC2[449:448]);
  TMP_DEST[511:480] ← Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[i] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*
        THEN *DEST[i+31:i] remains unchanged*
        ELSE DEST[i+31:i] ← 0 ;zeroing-masking
INSTRUCTION SET REFERENCE, V-Z

**VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values**

Fi;
Fi;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**VPERMILPS (256-bit variable version)**
DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[255:224] ← Select4(SRC1[255:128], SRC2[225:224]);
DEST[MAXVL-1:256] ← 0

**VPERMILPS (128-bit variable version)**
DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMILPS __m512 _mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 __m512_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __m512_mask_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 __m256_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __m256_mask_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 __m128_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __m128_mask_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 __m512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 __m512_mask_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 __m512_maskz_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m256 __m256_permutevar_ps( __m256i i, __m256 a);
VPERMILPS __m256 __m256_mask_permutevar_ps( __m256i i, __m256 a);
VPERMILPS __m256 __m256_maskz_permutevar_ps( __m256i i, __m256 a);
VPERMILPS __m128 __m128_permutevar_ps( __m128i i, __m128 a);
VPERMILPS __m128 __m128_maskz_permutevar_ps( __m128i i, __m128 a);
VPERMILPS __m128 __m128_permute_ps( __m128 a, int control);
VPERMILPS __m256 __m256_permute_ps( __m256 a, int control);
VPERMILPS __m256 __m256_permutevar_ps( __m256 a, __m256i control);
VPERMILPS __m256 __m256_permutevar_ps( __m256 a, __m256i control);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;
#UD If VEX.W = 1.
EVEX-encoded instruction, see Exceptions Type E4NF.
#UD If either (E)VEX.vvvv != 1111B and with imm8.
VPERMPD—Permute Double-Precision Floating-Point Elements

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W1 01</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 01</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 01</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Permute double-precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 16</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 16</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Permute double-precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The imm8 version: Copies quadword elements of double-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Operation

**VPERMPD (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN TMP_SRC[i+63:j] ← SRC[i+63:j];
    ELSE TMP_SRC[i+63:j] ← SRC[i+63:j];
  FI;
ENDFOR;

TMP_DEST[63:0] ← (TMP_SRC[256:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] ← (TMP_SRC[256:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC[256:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
  TMP_DEST[319:256] ← (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:j] ← TMP_DEST[i+63:j]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN DEST[i+63:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:j] ← 0 ;zeroing-masking
      FI;
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMPD (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:j] ← SRC2[i+63:j];
    ELSE TMP_SRC2[i+63:j] ← SRC2[i+63:j];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[63:0] ← (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
  TMP_DEST[63:0] ← (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
VPERMPD—Permute Double-Precision Floating-Point Elements

```
TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
```

FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-mask*
            THEN *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**VPERMPD (VEX.256 encoded version)**

```
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VPERMPD _m512d _mm512_permutex_pd(_m512d a, int imm);
VPERMPD _m512d _mm512_mask_permutex_pd(_m512d s, _m512d a);
VPERMPD _m512d _mm512_maskz_permutex_pd(_m512d a, int imm);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD IF VEX.L = 0.
    IF VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E4NF.

#UD IF encoded with EVEX.128.
    IF VEX.vvvv != 1111B and with imm8.
VPERMPS—Permute Single-Precision Floating-Point Elements

Summary

**VPERMPS**—Permute single-precision floating-point elements from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

Operation

**VPERMPS (EVEX forms)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

i ← j * 64

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN TMP_SRC2[i+31:i] ← SRC2[31:0];
ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i];

FI;

ENDFOR;

IF VL = 256

TMP_DEST[31:0] ← (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];

TMP_DEST[63:32] ← (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];

TMP_DEST[95:64] ← (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];

TMP_DEST[127:96] ← (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];

TMP_DEST[159:128] ← (TMP_SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];


TMP_DEST[255:224] ← (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
VPERMPS—Permute Single-Precision Floating-Point Elements

If $VL = 512$

$$
\begin{align*}
\text{TMP\_DEST}[31:0] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[3:0] \times 32))[31:0]; \\
\text{TMP\_DEST}[63:32] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[35:32] \times 32))[31:0]; \\
\text{TMP\_DEST}[95:64] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[67:64] \times 32))[31:0]; \\
\text{TMP\_DEST}[127:96] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[99:96] \times 32))[31:0]; \\
\text{TMP\_DEST}[159:128] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[131:128] \times 32))[31:0]; \\
\text{TMP\_DEST}[191:160] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[163:160] \times 32))[31:0]; \\
\text{TMP\_DEST}[223:192] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[195:192] \times 32))[31:0]; \\
\text{TMP\_DEST}[255:224] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[227:224] \times 32))[31:0]; \\
\text{TMP\_DEST}[287:256] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[259:256] \times 32))[31:0]; \\
\text{TMP\_DEST}[319:288] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[291:288] \times 32))[31:0]; \\
\text{TMP\_DEST}[351:320] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[323:320] \times 32))[31:0]; \\
\text{TMP\_DEST}[383:352] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[355:352] \times 32))[31:0]; \\
\text{TMP\_DEST}[415:384] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[387:384] \times 32))[31:0]; \\
\text{TMP\_DEST}[447:416] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[419:416] \times 32))[31:0]; \\
\text{TMP\_DEST}[479:448] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[451:448] \times 32))[31:0]; \\
\text{TMP\_DEST}[511:480] & \leftarrow (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[483:480] \times 32))[31:0];
\end{align*}
$$

For $j \leftarrow 0$ to $KL-1$

$$
\begin{align*}
i & \leftarrow j \times 32 \\
\text{IF } k1[j] \text{ OR } \text{no writemask} & \text{ THEN } \text{DEST}[i+31:i] \leftarrow \text{TMP\_DEST}[i+31:i] \\
& \text{ELSE} \\
& \text{IF } \text{merging-masking} & \text{ THEN } \text{DEST}[i+31:i] \text{ remains unchanged} \\
& \text{ELSE } \text{zeroing-masking} & \text{DEST}[i+31:i] \leftarrow 0 \\
& \text{FI}; \\
& \text{FI}; \\
& \text{ENDFOR}
\end{align*}
$$

DEST[MAXVL-1:VL] $\leftarrow 0$

VPERMPS (VEX.256 encoded version)

$$
\begin{align*}
\text{DEST}[31:0] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[2:0] \times 32))[31:0]; \\
\text{DEST}[63:32] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[34:32] \times 32))[31:0]; \\
\text{DEST}[95:64] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[66:64] \times 32))[31:0]; \\
\text{DEST}[127:96] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[98:96] \times 32))[31:0]; \\
\text{DEST}[159:128] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[130:128] \times 32))[31:0]; \\
\text{DEST}[191:160] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[162:160] \times 32))[31:0]; \\
\text{DEST}[223:192] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[194:192] \times 32))[31:0]; \\
\text{DEST}[255:224] & \leftarrow (\text{SRC2}[255:0] \gg (\text{SRC1}[226:224] \times 32))[31:0]; \\
\text{DEST}[\text{MAXVL}-1:256] & \leftarrow 0
\end{align*}
$$
Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS __m512 __mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 __mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 __mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 __mm256_permutexvar_ps(__m256i i, __m256 a);
VPERMPS __m256 __mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256i i, __m256 a);
VPERMPS __m256 __mm256_maskz_permutexvar_ps(__mmask8 k, __m256i i, __m256 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally
#UD If VEX.L = 0.
EVEX-encoded instruction, see Exceptions Type E4NF.
VPERMQ—Qwords Element Permutation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W1 00 / r ib VPERMQ ymm1, ymm2/m256, imm8</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 00 / r ib VPERMQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 00 / r ib VPERMQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 36 / r VPERMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 36 / r VPERMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (w)</td>
</tr>
</tbody>
</table>

**Description**

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand select which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.
Operation

VPERM Q (EVEX - imm8 control forms)
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN TMP_SRC[i+63:i] ← SRC[63:0];
    ELSE TMP_SRC[i+63:i] ← SRC[i+63:i];
  FI;
ENDFOR;
TMP_DEST[63:0] ← (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] ← (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
  TMP_DEST[319:256] ← (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
  TMP_DEST[383:320] ← (TMP_SRC[511:256] >> (IMM8[65:64] * 64))[63:0];
  TMP_DEST[511:448] ← (TMP_SRC[511:256] >> (IMM8[193:192] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i];
    ELSE *
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0 ;zeroing-masking
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPERMQ (EVEX - vector control forms)
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[63:0] ← (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
  TMP_DEST[63:0] ← (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0 ;zeroing-masking
      FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPERMQ (VEX.256 encoded version)
DEST[63:0] ← SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] ← SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64(__m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i _mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_maskz_permutex_epi64( __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i _mm256_mask_permutexvar_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4; additionally
#UD If VEX.L = 0.
If VEX.vvvv !≠ 1111B.
EVEX-encoded instruction, see Exceptions Type E4NF.
#UD If encoded with EVEX.128.
If VEX.vvvv !≠ 1111B and with imm8.
VPERM2B—Full Permute of Bytes from Two Tables Overwriting a Table

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.w0 7D/r VPERMT2B xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.w0 7D/r VPERMT2B ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.w0 7D/r VPERMT2B zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.
**Operation**

**VPERMT2B (EVEX encoded versions)**

\( (KL, VL) = (16, 128), (32, 256), (64, 512) \)

**IF** \( VL = 128: \)

\[ \text{id} \leftarrow 3; \]

**ELSE IF** \( VL = 256: \)

\[ \text{id} \leftarrow 4; \]

**ELSE IF** \( VL = 512: \)

\[ \text{id} \leftarrow 5; \]

**FI;**

\[ \text{TMP}_\text{DEST}[VL-1:0] \leftarrow \text{DEST}[VL-1:0]; \]

**FOR** \( j \leftarrow 0 \) **TO** \( KL-1 \)

\[ \text{off} \leftarrow 8*\text{SRC}_1[j*8 + \text{id}]*8; \]

**IF** \( k1[j] \) OR *no writemask*:

\[ \text{DEST}[j*8 + 7:j*8] \leftarrow \text{SRC}_1[j*8+\text{id}+1]*8 \text{ SRC}_2[\text{off}+7:\text{off}]: \text{TMP}_\text{DEST}[\text{off}+7:\text{off}]; \]

**ELSE IF** *zeroing-masking*:

\[ \text{DEST}[j*8 + 7:j*8] \leftarrow 0; \]

*ELSE:

\[ \text{DEST}[j*8 + 7:j*8] \text{ remains unchanged} \]

**FI;**

**ENDFOR**

\[ \text{DEST}[\text{MAX}_\text{VL}-1:VL] \leftarrow 0; \]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[ \text{VPERMT2B } \_\_\_m512i } \_\_\_mm512\_\_permute2var\_\_epi8(\_\_\_m512i } a, \_\_\_m512i } idx, \_\_\_m512i } b); \]

\[ \text{VPERMT2B } \_\_\_m512i } \_\_\_mm512\_\_mask\_\_permute2var\_\_epi8(\_\_\_m512i } a, \_\_\_mmask64 k, \_\_\_m512i } idx, \_\_\_m512i } b); \]

\[ \text{VPERMT2B } \_\_\_m512i } \_\_\_mm512\_\_maskz\_\_permute2var\_\_epi8(\_\_\_mmask64 k, \_\_\_m512i } a, \_\_\_m512i } idx, \_\_\_m512i } b); \]

\[ \text{VPERMT2B } \_\_\_m256i } \_\_\_mm256\_\_permute2var\_\_epi8(\_\_\_m256i } a, \_\_\_m256i } idx, \_\_\_m256i } b); \]

\[ \text{VPERMT2B } \_\_\_m256i } \_\_\_mm256\_\_mask\_\_permute2var\_\_epi8(\_\_\_mmask32 k, \_\_\_m256i } a, \_\_\_m256i } idx, \_\_\_m256i } b); \]

\[ \text{VPERMT2B } \_\_\_m256i } \_\_\_mm256\_\_maskz\_\_permute2var\_\_epi8(\_\_\_mmask32 k, \_\_\_m256i } a, \_\_\_m256i } idx, \_\_\_m256i } b); \]

\[ \text{VPERMT2B } \_\_\_m128i } \_\_\_mm128\_\_permute2var\_\_epi8(\_\_\_m128i } a, \_\_\_m128i } idx, \_\_\_m128i } b); \]

\[ \text{VPERMT2B } \_\_\_m128i } \_\_\_mm128\_\_mask\_\_permute2var\_\_epi8(\_\_\_mmask16 k, \_\_\_m128i } a, \_\_\_m128i } idx, \_\_\_m128i } b); \]

\[ \text{VPERMT2B } \_\_\_m128i } \_\_\_mm128\_\_maskz\_\_permute2var\_\_epi8(\_\_\_mmask16 k, \_\_\_m128i } a, \_\_\_m128i } idx, \_\_\_m128i } b); \]

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.
### VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 7D /r VPERMT2W xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 7D /r VPERMT2W ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 7D /r VPERMT2W zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A V/V</td>
<td>AVX512BW</td>
<td>Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 7E /r VPERMT2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 7E /r VPERMT2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 7E /r VPERMT2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 7E /r VPERMT2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 7F /r VPERMT2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
Description
Permutes 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands. If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0]/bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation
**VPERMT2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
   id ← 2
FI;
IF VL = 256
   id ← 3
FI;
IF VL = 512
   id ← 4
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
   i ← j * 16
   off ← 16*SRC1[i+id]
   IF k1[j] OR *no writemask*
      THEN
         DEST[i+15:i]=SRC1[i+id+1] ? SRC2[off+15:off]
         : TMP_DEST[off+15:off]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
         ELSE ; zeroing-masking

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRMreg (r,w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**
DEST[i+15:i] ← 0
FI
FIN;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPERMT2D/VPERMT2PS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id ← 1
FI;
IF VL = 256
    id ← 2
FI;
IF VL = 512
    id ← 3
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
    i ← j * 32
    off ← 32*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
            DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[31:0]
            : TMP_DEST[off+31:off]
        ELSE
            DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[off+31:off]
            : TMP_DEST[off+31:off]
        FI
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI
FIN;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPERMT2Q/VPERMT2PD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    id ← 0
FI;
IF VL = 256
    id ← 1
FI;
IF VL = 512
    id ← 2
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
i ← j * 64
off ← 64 * SRC1[i+id:i]
IF k1[i] OR "no writemask"
THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[63:0]
: TMP_DEST[off+63:off]
ELSE
DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[off+63:off]
: TMP_DEST[off+63:off]
FI
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] ← 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMT2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMT2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMT2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512b);
VPERMT2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
INSTRUCTION SET REFERENCE, V-Z

VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

VPERMT2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMT2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 _mm128_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm128_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm128_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMT2PS __m128 _mm128_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMT2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMT2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMT2Q __m128i _mm128i_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm128i_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm128i_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2Q __m128i _mm128i_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMT2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMT2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMT2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMT2W __m128i _mm128i_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMT2W __m128i _mm128i_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2W __m128i _mm128i_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2W __m128i _mm128i_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
VPERMT2D/Q/PS/PD: See Exceptions Type E4NF.
VPERMT2W: See Exceptions Type E4NF.nb.
**VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register**

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 89 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 89 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 89 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VPEXPANDD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

\[ k \leftarrow 0 \]

FOR \[ j \leftarrow 0 \) TO KL-1

\[ i \leftarrow j \times 32 \]

IF k1[j] OR *no writemask*

THEN

\[ \text{DEST}[i+31:i] \leftarrow \text{SRC}[k+31:k]; \]

\[ k \leftarrow k + 32 \]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+31:i] remains unchanged*

ELSE ; zeroing-masking

\[ \text{DEST}[i+31:i] \leftarrow 0 \]

FI

FI;

ENDFOR

\[ \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \]
Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32( __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32 ( __mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32( __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32 ( __mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32( __mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz_expand_epi32 ( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 (k1)[z], xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 (k1)[z], ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 (k1)[z], zmm2/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

VPEXPANDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

k ← 0

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      DEST[i+63:i] ← SRC[k+63:k];
      k ← k + 64
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          THEN DEST[i+63:i] ← 0
      FI
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDQ __m512i _mm512_maskexpandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskzexpandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskexpand_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskzexpand_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_maskexpandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskzexpandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskexpand_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskzexpand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_maskexpandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskzexpandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskexpand_epi64(__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskzexpand_epi64(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD xmm1, vm32x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD ymm1, vm32y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather dword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64y, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero’ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.
VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDD (VEX.128 version)
FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 3
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)
VPGATHERQD (VEX.128 version)
FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERDD (VEX.256 version)
FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 7
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)
VPGATHERQD (VEX.256 version)
FOR j ← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j ← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent
VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDD: __m256i _mm256_i32gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);
VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQD: __m256i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERQD: __m256i _mm256_mask_i64gather_epi32 (__m256i src, int const * base, __m256i index, __m128i mask, const int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 12.
VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 90 /vsib VPGATHERDD xmm1 [k1], vm32x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 90 /vsib VPGATHERDD ymm1 [k1], vm32y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 90 /vsib VPGATHERDD zmm1 [k1], vm32z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 90 /vsib VPGATHERDQ xmm1 [k1], vm32x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 90 /vsib VPGATHERDQ ymm1 [k1], vm32y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 90 /vsib VPGATHERDQ zmm1 [k1], vm32y</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address `BASE_ADDR` and index vector `VINDEX` with scale `SCALE` are gathered. The result is written into vector `zmm1`. The elements are specified via the `VSIB` (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register (`zmm1`) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

**Note that:**

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.
Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1). The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VPGATHERDD (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    THEN DEST[i+31:i] ← MEM[BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], 1)
    k1[j] ← 0
  ELSE *DEST[i+31:i] ← remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR

k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0

**VPGATHERDQ (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    THEN DEST[i+63:i] ← MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP])
    k1[j] ← 0
  ELSE *DEST[i+63:i] ← remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR

k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD __m512i_mm512_i32gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERDD __m512i_mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i_mm256_mmask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i_mm128i_mask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i_mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i_mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERDQ __m256i_mm256_mmask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i_mm128i_mask_i32logather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.
VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W1 90 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VPGATHERDQ xmm1, vm32x, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 91 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VPGATHERQQ xmm1, vm64x, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 90 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VPGATHERDQ ymm1, vm32x, ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 91 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VPGATHERQQ ymm1, vm64y, ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.
VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB{7:6};
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDQ (VEX.128 version)

FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] ← FFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] ← 0;
  FI;
ENDFOR

FOR j ← 0 to 1
  k ← j * 32;
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63:i] ← 0;
ENDFOR

MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

**VPGATHERQQ (VEX.128 version)**

FOR $j \leftarrow 0$ to $1$
  $i \leftarrow j \times 64$;
  IF MASK[63+i] THEN
    $\text{MASK}[i +63:i] \leftarrow \text{FFFFFFFF_FFFFFFFFH}$; // extend from most significant bit
  ELSE
    $\text{MASK}[i +63:i] \leftarrow 0$;
  FI;
ENDFOR

FOR $j \leftarrow 0$ to $1$
  $i \leftarrow j \times 64$;
  DATA_ADDR \leftarrow \text{BASE_ADDR} + \text{SignExtend(VINDEX1[i+63:i])*SCALE} + \text{DISP};
  IF MASK[63+i] THEN
    $\text{DEST}[i +63:i] \leftarrow \text{FETCH_64BITS}(\text{DATA_ADDR})$; // a fault exits the instruction
    $\text{FI}$;
    $\text{MASK}[i +63:i] \leftarrow 0$;
  ENDIF;
ENDFOR

$\text{MASK}[:128] \leftarrow 0$;
$\text{DEST}[:128] \leftarrow 0$;

(non-masked elements of the mask register have the content of respective element cleared)

**VPGATHERQQ (VEX.256 version)**

FOR $j \leftarrow 0$ to $3$
  $i \leftarrow j \times 64$;
  IF MASK[63+i] THEN
    $\text{MASK}[i +63:i] \leftarrow \text{FFFFFFFF_FFFFFFFFH}$; // extend from most significant bit
  ELSE
    $\text{MASK}[i +63:i] \leftarrow 0$;
  FI;
ENDFOR

FOR $j \leftarrow 0$ to $3$
  $i \leftarrow j \times 64$;
  DATA_ADDR \leftarrow \text{BASE_ADDR} + \text{SignExtend(VINDEX1[i+63:i])*SCALE} + \text{DISP};
  IF MASK[63+i] THEN
    $\text{DEST}[i +63:i] \leftarrow \text{FETCH_64BITS}(\text{DATA_ADDR})$; // a fault exits the instruction
    $\text{FI}$;
    $\text{MASK}[i +63:i] \leftarrow 0$;
  ENDIF;
ENDFOR

$\text{MASK}[:128] \leftarrow 0$;
$\text{DEST}[:128] \leftarrow 0$;

(non-masked elements of the mask register have the content of respective element cleared)

**VPGATHERDQ (VEX.256 version)**

FOR $j \leftarrow 0$ to $3$
  $i \leftarrow j \times 64$;
  IF MASK[63+i] THEN
    $\text{MASK}[i +63:i] \leftarrow \text{FFFFFFFF_FFFFFFFFH}$; // extend from most significant bit
  ELSE
    $\text{MASK}[i +63:i] \leftarrow 0$;
  FI;
ENDFOR

FOR $j \leftarrow 0$ to $3$
  $k \leftarrow j \times 32$;
  $i \leftarrow j \times 64$;
DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k]) * SCALE + DISP;
IF MASK[63+i] THEN
DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent
VPGATHERDQ: __m128i _mm_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDQ: __m256i _mm256_i32gather_epi64 (__int64 const * base, __m256i index, const int scale);
VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, __int64 const * base, __m128i index, __m256i mask, const int scale);
VPGATHERQQ: __m128i _mm_i64gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQQ: __m256i _mm256_i64gather_epi64 (__int64 const * base, __m256i index, const int scale);
VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, __int64 const * base, __m256i index, __m256i mask, const int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 12.
**VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices**

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>ModRM:reg (w) BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 91 /vsib VPGATHERQQ xmm1 {k1}, vm64x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 {k1}, vm64y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 {k1}, vm64z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

**Note that:**
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.
• These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VPGATHERQD (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j]
    THEN DEST[i+31:i] ← MEM[BASE_ADDR + (VINDEX[k+63:k] * SCALE + DISP)], 1)
    k1[j] ← 0
  ELSE *DEST[i+31:i] remains unchanged*  ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL/2] ← 0

**VPGATHERQQ (EVEX encoded version)**

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    THEN DEST[i+63:i] ← MEM[BASE_ADDR + (VINDEX[i+63:i] * SCALE + DISP])
    k1[j] ← 0
  ELSE *DEST[i+63:i] remains unchanged*  ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VPGATHERQD __m256i __mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i __mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i __mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i __mm512_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i __mm512_i64gather_epi64(__m512i vdx, void * base, int scale);
VPGATHERQQ __m512i __mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i __mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i __mm128_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.
VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 44 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 44 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 44 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 44 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 44 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 44 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VPLZCNTD**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j \times 32\]

IF MaskBit(j) OR *no writemask*

THEN

\[temp \leftarrow 32\]

\[DEST[i+31:i] \leftarrow 0\]

WHILE (temp > 0) AND (SRC[i+temp-1] = 0)

DO

\[temp \leftarrow temp - 1\]

\[DEST[i+31:i] \leftarrow DEST[i+31:i] + 1\]

OD

ELSE

IF *merging-masking*

THEN *DEST[i+31:i] remains unchanged*

ELSE \[DEST[i+31:i] \leftarrow 0\]

FI

ENDFOR

\[DEST[MAXVL-1:VL] \leftarrow 0\]

**VPLZCNTQ**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j \times 64\]

IF MaskBit(j) OR *no writemask*

THEN

\[temp \leftarrow 64\]

\[DEST[i+63:i] \leftarrow 0\]

WHILE (temp > 0) AND (SRC[i+temp-1] = 0)

DO

\[temp \leftarrow temp - 1\]

\[DEST[i+63:i] \leftarrow DEST[i+63:i] + 1\]

OD

ELSE

IF *merging-masking*

THEN *DEST[i+63:i] remains unchanged*

ELSE \[DEST[i+63:i] \leftarrow 0\]

FI

ENDFOR

\[DEST[MAXVL-1:VL] \leftarrow 0\]
Intel C/C++ Compiler Intrinsic Equivalent

VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.
VPMADD52HUQ—Packed Multiply of Unsigned 52-bit Unsigned Integers and Add High 52-bit Products to 64-bit Accumulators

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>32/64 bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 B5 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPMADD52HUQ xmm1 {k1}[z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 B5 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPMADD52HUQ ymm1 {k1}[z], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 B5 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA</td>
<td>Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPMADD52HUQ zmm1 {k1}[z], zmm2, zmm3/m512/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m(r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.
Operation

VPMADD52HUQ (EVEX encoded)

\( (KL, VL) = (2, 128), (4, 256), (8, 512) \)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\( i \leftarrow j * 64; \)

IF \( k1[j] \) OR *no writemask* THEN

IF src2 is Memory AND EVEX.b=1 THEN

\( tsrc2[63:0] \leftarrow \text{ZeroExtend64}(src2[51:0]); \)

ELSE

\( tsrc2[63:0] \leftarrow \text{ZeroExtend64}(src2[i+51:i]); \)

FI;

\( \text{Temp128}[127:0] \leftarrow \text{ZeroExtend64}(src1[i+51:i]) \times tsrc2[63:0]; \)

\( \text{Temp2}[63:0] \leftarrow \text{DEST}[i+63:i] + \text{ZeroExtend64}(\text{temp128}[103:52]); \)

\( \text{DEST}[i+63:i] \leftarrow \text{Temp2}[63:0]; \)

ELSE

IF *zeroing-masking* THEN

\( \text{DEST}[i+63:i] \leftarrow 0; \)

ELSE *merge-masking*

\( \text{DEST}[i+63:i] \) is unchanged;

FI;

FI;

ENDFOR

\( \text{DEST}[\text{MAX}_VL-1:VL] \leftarrow 0 \)

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52HUQ __m512i _mm512_madd52hi_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_mask_madd52hi_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_maskz_madd52hi_epu64( __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_mask_madd52hi_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_maskz_madd52hi_epu64( __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m128i _mm_madd52hi_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_mask_madd52hi_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_maskz_madd52hi_epu64( __mmask8 k, __m128i a, __m128i b, __m128i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.
**VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword Accumulators**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>32/64 bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.wT B4 / r VPMADD52LUQ xmm1 (k1){z}, xmm2,xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.wT B4 / r VPMADD52LUQ ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.wT B4 / r VPMADD52LUQ zmm1 (k1){z}, zmm2,zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA</td>
<td>Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m(r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Multiples packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.
**Operation**

**VPMADD52LUQ (EVEX encoded)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\(i \leftarrow j \times 64;\)

IF \(k1[j] \text{ OR no writemask} \) THEN

IF src2 is Memory AND EVEX.b=1 THEN

\(\text{tsrc2}[63:0] \leftarrow \text{ZeroExtend64}([src2][51:0]);\)

ELSE

\(\text{tsrc2}[63:0] \leftarrow \text{ZeroExtend64}([src2][i+51:i]);\)

FI;

\(\text{Temp128}[127:0] \leftarrow \text{ZeroExtend64}([src1][i+51:i]) \times \text{tsrc2}[63:0];\)

\(\text{Temp2}[63:0] \leftarrow \text{DEST}[i+63:i] + \text{ZeroExtend64}([\text{temp128}[51:0]]);\)

\(\text{DEST}[i+63:i] \leftarrow \text{Temp2}[63:0];\)

ELSE

IF *zeroing-masking* THEN

\(\text{DEST}[i+63:i] \leftarrow 0;\)

ELSE *merge-masking*

\(\text{DEST}[i+63:i] \text{ is unchanged};\)

FI;

FI;

ENDFOR

\(\text{DEST}[\text{MAX}_{\text{VL}}-1:VL] \leftarrow 0;\)

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{VPMADD52LUQ}\_\_m512i\_\_nm512\_madd52lo\_epu64(\_m512i a, \_m512i b, \_m512i c);
\]

\[
\text{VPMADD52LUQ}\_\_m512i\_\_nm512\_\_mask\_madd52lo\_epu64(\_m512i s, \_mmask8 k, \_m512i a, \_m512i b, \_m512i c);
\]

\[
\text{VPMADD52LUQ}\_\_m512i\_\_nm512\_\_maskz\_madd52lo\_epu64(\_mmask8 k, \_m512i a, \_m512i b, \_m512i c);
\]

\[
\text{VPMADD52LUQ}\_\_m256i\_\_nm256\_madd52lo\_epu64(\_m256i a, \_m256i b, \_m256i c);
\]

\[
\text{VPMADD52LUQ}\_\_m256i\_\_nm256\_\_mask\_madd52lo\_epu64(\_m256i s, \_mmask8 k, \_m256i a, \_m256i b, \_m256i c);
\]

\[
\text{VPMADD52LUQ}\_\_m256i\_\_nm256\_\_maskz\_madd52lo\_epu64(\_mmask8 k, \_m256i a, \_m256i b, \_m256i c);
\]

\[
\text{VPMADD52LUQ}\_\_m128i\_\_nm128i\_\_madd52lo\_epu64(\_m128i a, \_m128i b, \_m128i c);
\]

\[
\text{VPMADD52LUQ}\_\_m128i\_\_nm128i\_\_mask\_madd52lo\_epu64(\_m128i s, \_mmask8 k, \_m128i a, \_m128i b, \_m128i c);
\]

\[
\text{VPMADD52LUQ}\_\_m128i\_\_nm128i\_\_maskz\_madd52lo\_epu64(\_mmask8 k, \_m128i a, \_m128i b, \_m128i c);
\]

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.
VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRMr/m (w)</td>
<td>VEX.vvvv</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVD and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.
VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

**VPMASKMOVD - 256-bit load**

\[
\begin{align*}
\text{DEST}[31:0] &\leftarrow \text{IF} (\text{SRC1}[31]) \text{Load}_32(\text{mem}) \text{ELSE} 0 \\
\text{DEST}[63:32] &\leftarrow \text{IF} (\text{SRC1}[63]) \text{Load}_32(\text{mem} + 4) \text{ELSE} 0 \\
\text{DEST}[95:64] &\leftarrow \text{IF} (\text{SRC1}[95]) \text{Load}_32(\text{mem} + 8) \text{ELSE} 0 \\
\text{DEST}[127:96] &\leftarrow \text{IF} (\text{SRC1}[127]) \text{Load}_32(\text{mem} + 12) \text{ELSE} 0 \\
\text{DEST}[159:128] &\leftarrow \text{IF} (\text{SRC1}[159]) \text{Load}_32(\text{mem} + 16) \text{ELSE} 0 \\
\text{DEST}[191:160] &\leftarrow \text{IF} (\text{SRC1}[191]) \text{Load}_32(\text{mem} + 20) \text{ELSE} 0 \\
\text{DEST}[223:192] &\leftarrow \text{IF} (\text{SRC1}[223]) \text{Load}_32(\text{mem} + 24) \text{ELSE} 0 \\
\text{DEST}[255:224] &\leftarrow \text{IF} (\text{SRC1}[255]) \text{Load}_32(\text{mem} + 28) \text{ELSE} 0
\end{align*}
\]

**VPMASKMOVD - 128-bit load**

\[
\begin{align*}
\text{DEST}[31:0] &\leftarrow \text{IF} (\text{SRC1}[31]) \text{Load}_32(\text{mem}) \text{ELSE} 0 \\
\text{DEST}[63:32] &\leftarrow \text{IF} (\text{SRC1}[63]) \text{Load}_32(\text{mem} + 4) \text{ELSE} 0 \\
\text{DEST}[95:64] &\leftarrow \text{IF} (\text{SRC1}[95]) \text{Load}_32(\text{mem} + 8) \text{ELSE} 0 \\
\text{DEST}[127:97] &\leftarrow \text{IF} (\text{SRC1}[127]) \text{Load}_32(\text{mem} + 12) \text{ELSE} 0 \\
\text{DEST}[\text{MAXVL}-1:128] &\leftarrow 0
\end{align*}
\]

**VPMASKMOVQ - 256-bit load**

\[
\begin{align*}
\text{DEST}[63:0] &\leftarrow \text{IF} (\text{SRC1}[63]) \text{Load}_64(\text{mem}) \text{ELSE} 0 \\
\text{DEST}[127:64] &\leftarrow \text{IF} (\text{SRC1}[127]) \text{Load}_64(\text{mem} + 8) \text{ELSE} 0 \\
\text{DEST}[195:128] &\leftarrow \text{IF} (\text{SRC1}[191]) \text{Load}_64(\text{mem} + 16) \text{ELSE} 0 \\
\text{DEST}[255:196] &\leftarrow \text{IF} (\text{SRC1}[255]) \text{Load}_64(\text{mem} + 24) \text{ELSE} 0
\end{align*}
\]

**VPMASKMOVQ - 128-bit load**

\[
\begin{align*}
\text{DEST}[63:0] &\leftarrow \text{IF} (\text{SRC1}[63]) \text{Load}_64(\text{mem}) \text{ELSE} 0 \\
\text{DEST}[127:64] &\leftarrow \text{IF} (\text{SRC1}[127]) \text{Load}_64(\text{mem} + 16) \text{ELSE} 0 \\
\text{DEST}[\text{MAXVL}-1:128] &\leftarrow 0
\end{align*}
\]

**VPMASKMOVD - 256-bit store**

\[
\begin{align*}
\text{IF} (\text{SRC1}[31]) \text{DEST}[31:0] &\leftarrow \text{SRC2}[31:0] \\
\text{IF} (\text{SRC1}[63]) \text{DEST}[63:32] &\leftarrow \text{SRC2}[63:32] \\
\text{IF} (\text{SRC1}[95]) \text{DEST}[95:64] &\leftarrow \text{SRC2}[95:64] \\
\text{IF} (\text{SRC1}[127]) \text{DEST}[127:96] &\leftarrow \text{SRC2}[127:96] \\
\text{IF} (\text{SRC1}[159]) \text{DEST}[159:128] &\leftarrow \text{SRC2}[159:128] \\
\text{IF} (\text{SRC1}[191]) \text{DEST}[191:160] &\leftarrow \text{SRC2}[191:160] \\
\text{IF} (\text{SRC1}[223]) \text{DEST}[223:192] &\leftarrow \text{SRC2}[223:192] \\
\text{IF} (\text{SRC1}[255]) \text{DEST}[255:224] &\leftarrow \text{SRC2}[255:224]
\end{align*}
\]
VPMASKMOVD - 128-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VPMASKMOVQ - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

VPMASKMOVQ - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent
VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)
VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)
VPMASKMOVQ: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);
VPMASKMOVQ: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m128i b);
VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)
VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128i b)
VPMASKMOVQ: __m128i _mm_maskload_epi64(__int64 const *a, __m128i mask);
VPMASKMOVQ: void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 6 (No AC# reported for any mask bit combinations).
### VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1</td>
<td>RM V/V</td>
<td>AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1</td>
<td>RM V/V</td>
<td>AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1</td>
<td>RM V/V</td>
<td>AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1</td>
<td>RM V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1</td>
<td>RM V/V</td>
<td>AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.</td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of the most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVB2M (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  IF SRC[i+7]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPMOVW2M (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF SRC[i+15]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPMOVD2M (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF SRC[i+31]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPMOVQ2M (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF SRC[i+63]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalents
VPMPOVB2M __mmask64 __mm512_movepi8_mask( __m512i );
VPMPOVD2M __mmask16 __mm512_movepi32_mask( __m512i );
VPMPOVQ2M __mmask8 __mm512_movepi64_mask( __m512i );
VPMPOVw2M __mmask32 __mm512_movepi16_mask( __m512i );
VPMPOVB2M __mmask32 __mm256_movepi8_mask( __m256i );
VPMPOVD2M __mmask8 __mm256_movepi32_mask( __m256i );
VPMPOVQ2M __mmask8 __mm256_movepi64_mask( __m256i );
VPMPOVw2M __mmask16 __mm256_movepi16_mask( __m256i );
VPMPOVB2M __mmask16 __mm_movepi8_mask( __m128i );
VPMPOVD2M __mmask8 __mm_movepi32_mask( __m128i );
VPMPOVQ2M __mmask8 __mm_movepi64_mask( __m128i );
VPMPOVw2M __mmask8 __mm_movepi16_mask( __m128i );

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E7NM
#UD If EVEX.vvvv != 1111B.
### VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

**Instruction Encoding**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 31 /r VPMOVDB xmm1/m32 [k1]{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed double-word integers from xmm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 21 /r VPMOVDSB xmm1/m32 [k1]{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed double-word integers from xmm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m32 [k1]{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned byte integers in xmm1/m32 using unsign saturated under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 31 /r VPMOVDB xmm1/m64 [k1]{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed double-word integers from ymm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 21 /r VPMOVDSB xmm1/m64 [k1]{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed double-word integers from ymm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m64 [k1]{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned byte integers in xmm1/m64 using unsign saturated under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 31 /r VPMOVDB xmm1/m128 [k1]{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 21 /r VPMOVDSB xmm1/m128 [k1]{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m128 [k1]{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsign saturated under writemask k1.</td>
</tr>
</tbody>
</table>

**Description**

VPMOVDB down converts 32-bit integer elements in the source operand (the second operand) into packed byte using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB convert unsigned double-word values into unsigned byte values using unsign saturated.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VPMOVDB instruction (EVEX encoded versions) when dest is a register**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \) TO \(KL - 1\)

\(i \leftarrow j \times 8\)

\(m \leftarrow j \times 32\)

IF \(k1[j] \) OR *no writemask*  
THEN \(\text{DEST}[i+7:i] \leftarrow \text{TruncateDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE  
IF *merging-masking*  
THEN *DEST*[i+7:i] remains unchanged*  
ELSE *zeroing-masking*  
\(\text{DEST}[i+7:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL/4] \leftarrow 0;\)

**VPMOVDB instruction (EVEX encoded versions) when dest is memory**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \) TO \(KL - 1\)

\(i \leftarrow j \times 8\)

\(m \leftarrow j \times 32\)

IF \(k1[j] \) OR *no writemask*  
THEN \(\text{DEST}[i+7:i] \leftarrow \text{TruncateDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE *DEST*[i+7:i] remains unchanged*  
FI

ENDFOR

**VPMOVSDB instruction (EVEX encoded versions) when dest is a register**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \) TO \(KL - 1\)

\(i \leftarrow j \times 8\)

\(m \leftarrow j \times 32\)

IF \(k1[j] \) OR *no writemask*  
THEN \(\text{DEST}[i+7:i] \leftarrow \text{SaturateSignedDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE *merging-masking*  
THEN *DEST*[i+7:i] remains unchanged*  
ELSE *zeroing-masking*  
\(\text{DEST}[i+7:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL/4] \leftarrow 0;\)
VPMOVDB instruction (EVEX encoded versions) when dest is memory

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
i \leftarrow j \times 8
\]
\[
m \leftarrow j \times 32
\]
IF \(k1[j] \text{ OR } \text{no writemask}\)
\[
\text{THEN } \text{DEST}[i+7:i] \leftarrow \text{SaturateSignedDoubleWordToByte}(\text{SRC}[m+31:m])
\]
ELSE \(\text{DEST}[i+7:i] \text{ remains unchanged}^*\) ; merging-masking
\[
\text{FI;}
\]
ENDFOR

VPMOVUSDB instruction (EVEX encoded versions) when dest is a register

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
i \leftarrow j \times 8
\]
\[
m \leftarrow j \times 32
\]
IF \(k1[j] \text{ OR } \text{no writemask}\)
\[
\text{THEN } \text{DEST}[i+7:i] \leftarrow \text{SaturateUnsignedDoubleWordToByte}(\text{SRC}[m+31:m])
\]
ELSE
\[
\text{IF } \text{merging-masking}^* \; ; \text{merging-masking}
\]
\[
\text{THEN } \text{DEST}[i+7:i] \text{ remains unchanged}^*\)
\]
ELSE \(\text{zeroing-masking}^*\) \; ; zeroing-masking
\[
\text{DEST}[i+7:i] \leftarrow 0
\]
\[
\text{FI}
\]
ENDFOR
DEST[\text{MAXVL-1:VL/4}] \leftarrow 0;

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
i \leftarrow j \times 8
\]
\[
m \leftarrow j \times 32
\]
IF \(k1[j] \text{ OR } \text{no writemask}\)
\[
\text{THEN } \text{DEST}[i+7:i] \leftarrow \text{SaturateUnsignedDoubleWordToByte}(\text{SRC}[m+31:m])
\]
ELSE \(\text{DEST}[i+7:i] \text{ remains unchanged}^*\) ; merging-masking
\[
\text{FI;}
\]
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB void _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVDw/VPMOVSDw/VPMOVUSDw—Down Convert DWord to Word

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 33 /r VPMOVDw xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed double-word integers from ( xmm2 ) into 4 packed word integers in ( xmm1/m64 ) with truncation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 23 /r VPMOVSDw xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed double-word integers from ( xmm2 ) into 4 packed signed word integers in ( xmm1/m64 ) using signed saturation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 13 /r VPMOVUSDw xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned double-word integers from ( xmm2 ) into 4 packed unsigned word integers in ( xmm1/m64 ) using unsigned saturation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 33 /r VPMOVDw xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed double-word integers from ( ymm2 ) into 8 packed word integers in ( xmm1/m128 ) with truncation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 23 /r VPMOVSDw xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed double-word integers from ( ymm2 ) into 8 packed signed word integers in ( xmm1/m128 ) using signed saturation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 13 /r VPMOVUSDw xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed unsigned double-word integers from ( ymm2 ) into 8 packed unsigned word integers in ( xmm1/m128 ) using unsigned saturation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 33 /r VPMOVDw ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed double-word integers from ( zmm2 ) into 16 packed word integers in ( ymm1/m256 ) with truncation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 23 /r VPMOVSDw ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed signed double-word integers from ( zmm2 ) into 16 packed signed word integers in ( ymm1/m256 ) using signed saturation under writemask ( k1 ).</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 13 /r VPMOVUSDw ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed unsigned double-word integers from ( zmm2 ) into 16 packed unsigned word integers in ( ymm1/m256 ) using unsigned saturation under writemask ( k1 ).</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half Mem</td>
<td>ModRM:rs/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VPMOVDw down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDw converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDw converts unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVDw instruction (EVEX encoded versions) when dest is a register

$(KL, VL) = (4, 128), (8, 256), (16, 512)$
FOR $j \leftarrow 0$ TO KL-1
  $i \leftarrow j \times 16$
  $m \leftarrow j \times 32$
  IF $k1[j]$ OR *no writemask*
    THEN $DEST[i+15:i] \leftarrow \text{TruncateDoubleWordToWord} (\text{SRC}[m+31:m])$
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST*[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        $DEST[i+15:i] \leftarrow 0$
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] \leftarrow 0;

VPMOVDw instruction (EVEX encoded versions) when dest is memory

$(KL, VL) = (4, 128), (8, 256), (16, 512)$
FOR $j \leftarrow 0$ TO KL-1
  $i \leftarrow j \times 16$
  $m \leftarrow j \times 32$
  IF $k1[j]$ OR *no writemask*
    THEN $DEST[i+15:i] \leftarrow \text{TruncateDoubleWordToWord} (\text{SRC}[m+31:m])$
    ELSE
    *DEST*[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVSDw instruction (EVEX encoded versions) when dest is a register

$(KL, VL) = (4, 128), (8, 256), (16, 512)$
FOR $j \leftarrow 0$ TO KL-1
  $i \leftarrow j \times 16$
  $m \leftarrow j \times 32$
  IF $k1[j]$ OR *no writemask*
    THEN $DEST[i+15:i] \leftarrow \text{SaturateSignedDoubleWordToWord} (\text{SRC}[m+31:m])$
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST*[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        $DEST[i+15:i] \leftarrow 0$
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] \leftarrow 0;
VPMOVSDW instruction (EVEX encoded versions) when dest is memory
\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
\begin{align*}
i &\leftarrow j \times 16 \\
m &\leftarrow j \times 32 \\
\text{IF } k1[j] \text{ OR } *\text{no writemask}* \\
&\quad \text{THEN } \text{DEST}[i+15:i] \leftarrow \text{SaturateSignedDoubleWordToWord}(\text{SRC}[m+31:m]) \\
&\quad \text{ELSE} \\
&\quad \quad \text{*DEST}[i+15:i] \text{ remains unchanged*} \quad ; \text{merging-masking} \\
\text{FI}; \\
\end{align*}
\]
ENDFOR

VPMOVUSDW instruction (EVEX encoded versions) when dest is a register
\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
\begin{align*}
i &\leftarrow j \times 16 \\
m &\leftarrow j \times 32 \\
\text{IF } k1[j] \text{ OR } *\text{no writemask}* \\
&\quad \text{THEN } \text{DEST}[i+15:i] \leftarrow \text{SaturateUnsignedDoubleWordToWord}(\text{SRC}[m+31:m]) \\
&\quad \text{ELSE} \\
&\quad \quad \text{IF } *\text{merging-masking*} \quad ; \text{merging-masking} \\
&\quad \quad \quad \text{THEN } *\text{DEST}[i+15:i] \text{ remains unchanged*} \\
&\quad \quad \quad \text{ELSE } *\text{zeroing-masking*} \quad ; \text{zeroing-masking} \\
&\quad \quad \quad \text{DEST}[i+15:i] \leftarrow 0 \\
\text{FI} \\
\text{FI} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL}-1:VL/2] \leftarrow 0;
\]

VPMOVUSDW instruction (EVEX encoded versions) when dest is memory
\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
\begin{align*}
i &\leftarrow j \times 16 \\
m &\leftarrow j \times 32 \\
\text{IF } k1[j] \text{ OR } *\text{no writemask}* \\
&\quad \text{THEN } \text{DEST}[i+15:i] \leftarrow \text{SaturateUnsignedDoubleWordToWord}(\text{SRC}[m+31:m]) \\
&\quad \text{ELSE} \\
&\quad \quad \text{*DEST}[i+15:i] \text{ remains unchanged*} \quad ; \text{merging-masking} \\
\text{FI}; \\
\text{ENDFOR} \\
\]
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDW __m256i _mm512_cvtepi32_epi16( __m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16( _m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtepi32_epi16( __mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_cvtsepi32_epi16 __m512i a);
VPMOVDW __m256i _mm512_cvtusepi32_epi16 __m512i a);
VPMOVDW __m256i _mm512_cvtepi32_epi16( __m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_cvtsepi32_epi16(__m256i a);
VPMOVDW __m256i _mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_cvtusepi32_epi16(__m256i a);
VPMOVDW __m256i _mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVDW __m128i _mm256_cvtsepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtsepi32_epi16(__m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtsepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_cvtusepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtusepi32_epi16(__m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtusepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm_cvtepi32_epi16(__m128i a);
VPMOVDW __m128i _mm_mask_cvtepi32_epi16(__m128i s, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void _mm_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm256_cvtsepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtsepi32_epi16(__m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtsepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_cvtusepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtusepi32_epi16(__m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtusepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm_cvtsepi32_epi16(__m128i a);
VPMOVDW __m128i _mm_mask_cvtsepi32_epi16(__m128i s, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_maskz_cvtsepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void _mm_mask_cvtsepi32_storeu_epi16(void * d, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_cvtusepi32_epi16(__m128i a);
VPMOVDW __m128i _mm_mask_cvtusepi32_epi16(__m128i s, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_maskz_cvtusepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void _mm_mask_cvtusepi32_storeu_epi16(void * d, __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts a mask register to a vector register. Each element in the destination register is set to all 1’s or all 0’s depending on the value of the corresponding bit in the source mask register. The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register. EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVM2B (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF SRC[i]
        THEN DEST[i+7:i] ← -1
        ELSE DEST[i+7:i] ← 0
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

VPMOVM2W (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF SRC[i]
        THEN DEST[i+15:i] ← -1
        ELSE DEST[i+15:i] ← 0
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

VPMOVM2D (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF SRC[i]
        THEN DEST[i+31:i] ← -1
        ELSE DEST[i+31:i] ← 0
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0

VPMOVM2Q (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF SRC[i]
        THEN DEST[i+63:i] ← -1
        ELSE DEST[i+63:i] ← 0
    FI;
ENDFOR

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i _mm512_movm_epi8(__mmask64 );
VPMOVM2D __m512i _mm512_movm_epi32(__mmask8 );
VPMOVM2Q __m512i _mm512_movm_epi64(__mmask16 );
VPMOVM2W __m512i _mm512_movm_epi16(__mmask32 );
VPMOVM2B __m256i _mm256_movm_epi8(__mmask32 );
VPMOVM2D __m256i _mm256_movm_epi32(__mmask8 );
VPMOVM2Q __m256i _mm256_movm_epi64(__mmask8 );
VPMOVM2W __m256i _mm256_movm_epi16(__mmask16 );
VPMOVM2B __m128i _mm_movm_epi8(__mmask16 );
VPMOVM2D __m128i _mm_movm_epi32(__mmask8 );
VPMOVM2Q __m128i _mm_movm_epi64(__mmask8 );
VPMOVM2W __m128i _mm_movm_epi16(__mmask8 );

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E7NM

#UD If EVEX.vvvv != 1111B.
VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

### Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB converts unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EIXEV.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Instruction Set Encoding

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 32 /r</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 22 /r</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 12 /r</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 32 /r</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 22 /r</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 12 /r</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 32 /r</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 22 /r</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 12 /r</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed byte integers in xmm1/m16 with truncation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed byte integers in xmm1/m16 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned byte integers in xmm1/m16 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>

| Instruction Operand Encoding |

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Eighth Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EIXEV.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVQB instruction (EVEX encoded versions) when dest is a register
KL, VL = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;

VPMOVQB instruction (EVEX encoded versions) when dest is memory
KL, VL = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVSQB instruction (EVEX encoded versions) when dest is a register
KL, VL = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;

VPMOVSQB instruction (EVEX encoded versions) when dest is memory
KL, VL = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
VPMOVQSQB instruction (EVEX encoded versions) when dest is memory

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 8\)
  \(m \leftarrow j \times 64\)
  IF \(k[1][j]\) OR *no writemask*
    THEN \(DEST[i+7:i] \leftarrow \text{SaturateSignedQuadWordToByte}(SRC[m+63:m])\)
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 8\)
  \(m \leftarrow j \times 64\)
  IF \(k[1][j]\) OR *no writemask*
    THEN \(DEST[i+7:i] \leftarrow \text{SaturateUnsignedQuadWordToByte}(SRC[m+63:m])\)
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        \(DEST[i+7:i] \leftarrow 0\)
      FI
    FI
ENDFOR

\(DEST[MAXVL-1:VL/8] \leftarrow 0;\)

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 8\)
  \(m \leftarrow j \times 64\)
  IF \(k[1][j]\) OR *no writemask*
    THEN \(DEST[i+7:i] \leftarrow \text{SaturateUnsignedQuadWordToByte}(SRC[m+63:m])\)
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 25 /r VPMOVSQD xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed double-word integers in xmm1/m64 using signed saturation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned double-word integers in xmm1/m64 using unsigned saturation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 25 /r VPMOVSQD xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed double-word integers in xmm1/m128 using signed saturation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned double-word integers in xmm1/m128 using unsigned saturation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 35 /r VPMOVQD ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed double-word integers in ymm1/m256 with truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 25 /r VPMOVSQD ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed double-word integers in ymm1/m256 using signed saturation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned double-word integers in ymm1/m256 using unsigned saturation subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half Mem</td>
<td>ModRM:rs/m(w)</td>
<td>ModRM:reg(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVQD instruction (EVEX encoded version) reg-reg form

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
  FI
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

VPMOVQD instruction (EVEX encoded version) memory form

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVSQD instruction (EVEX encoded version) reg-reg form

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

INSTRUCTION SET REFERENCE, V-Z

Vol. 2C 5-425

VPMOVQD instruction (EVEX encoded version) memory form

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\[ i \leftarrow j \times 32 \]
\[ m \leftarrow j \times 64 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+31:i] \leftarrow \text{SaturateSignedQuadWordToDWord} (\text{SRC}[m+63:m]) \)
ELSE *DEST[31:i] remains unchanged* ; merging-masking
FI;
ENDFOR

VPMOVUSQD instruction (EVEX encoded version) reg-reg form

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\[ i \leftarrow j \times 32 \]
\[ m \leftarrow j \times 64 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+31:i] \leftarrow \text{SaturateUnsignedQuadWordToDWord} (\text{SRC}[m+63:m]) \)
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking

\[ \text{DEST}[i+31:i] \leftarrow 0 \]
FI

FI;
ENDFOR

DEST[\( MAXVL-1:VL/2 \)] \( \leftarrow 0; \)

VPMOVUSQD instruction (EVEX encoded version) memory form

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\[ i \leftarrow j \times 32 \]
\[ m \leftarrow j \times 64 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+31:i] \leftarrow \text{SaturateUnsignedQuadWordToDWord} (\text{SRC}[m+63:m]) \)
ELSE *DEST[31:i] remains unchanged* ; merging-masking
FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQD __m256i _mm512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD void __m512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_cvtepi64_epi32(__m512i a);

VPMOVSD __m256i _mm512_mask_cvtsi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVSD __m256i _mm512_maskz_cvtsi64_epi32(__mmask8 k, __m512i a);
VPMOVSD void __m512_mask_cvtsi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSD __m256i _mm512_cvtsi64_epi32(__m512i a);
VPMOVSD void __mm512_mask_cvtsi64_storeu_epi32(void * d, __mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 34</td>
<td>r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed word integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVQW xmm1/m32 (k1)[z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 24</td>
<td>r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m32 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSQW xmm1/m32 (k1)[z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 14</td>
<td>r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned word integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSQW xmm1/m32 (k1)[z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 34</td>
<td>r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVQW xmm1/m64 (k1)[z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 24</td>
<td>r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSQW xmm1/m64 (k1)[z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 14</td>
<td>r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSQW xmm1/m64 (k1)[z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 34</td>
<td>r</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVQW xmm1/m128 (k1)[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 24</td>
<td>r</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSQW xmm1/m128 (k1)[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 14</td>
<td>r</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSQW xmm1/m128 (k1)[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

- **Op/En**
  - A: Quarter Mem

- **Tuple Type**
  - ModRM:r/m (w)
  - ModRM:reg (r)

- **Operand 1**
  - ModRM:r/m (w)

- **Operand 2**
  - ModRM:reg (r)

- **Operand 3**
  - NA

- **Operand 4**
  - NA

**Description**

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW converts unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

**VPMOVQW instruction (EVEX encoded versions) when dest is a register**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 16\)

\(m \leftarrow j \times 64\)

IF \(k1[j] \text{ OR } \text{*no writemask*}\)

THEN \(\text{DEST}[i+15:i] \leftarrow \text{TruncateQuadWordToWord} (\text{SRC}[m+63:m])\)

ELSE

IF \(*\text{merging-masking*}\) ; merging-masking

THEN \(*\text{DEST}[i+15:i] \text{ remains unchanged*}\)

ELSE \(*\text{zeroing-masking*}\) ; zeroing-masking

\(\text{DEST}[i+15:i] \leftarrow 0\)

\(\text{FI}\)

\(\text{FI}\);

ENDFOR

\(\text{DEST}[^{\text{MAXVL-1:VL/4}}] \leftarrow 0;\)

**VPMOVQW instruction (EVEX encoded versions) when dest is memory**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 16\)

\(m \leftarrow j \times 64\)

IF \(k1[j] \text{ OR } \text{*no writemask*}\)

THEN \(\text{DEST}[i+15:i] \leftarrow \text{TruncateQuadWordToWord} (\text{SRC}[m+63:m])\)

ELSE

\(*\text{DEST}[i+15:i] \text{ remains unchanged*}\) ; merging-masking

\(\text{FI}\)

ENDFOR

**VPMOVUSQW instruction (EVEX encoded versions) when dest is a register**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 16\)

\(m \leftarrow j \times 64\)

IF \(k1[j] \text{ OR } \text{*no writemask*}\)

THEN \(\text{DEST}[i+15:i] \leftarrow \text{SaturateSignedQuadWordToWord} (\text{SRC}[m+63:m])\)

ELSE

\(*\text{DEST}[i+15:i] \text{ remains unchanged*}\) ; merging-masking

\(\text{FI}\)

ENDFOR

\(\text{DEST}[^{\text{MAXVL-1:VL/4}}] \leftarrow 0;\)
VPMOVUSQW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVUSQW instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] ← 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;

VPMOVUSQW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW __m128i _mm512_cvtepi64_epi16( __m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_cvtepi64_epi16( __m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_cvtepi64_epi16( __m512i a);

VPMOVSQW __m128i _mm512_cvtsepi64_epi16( __m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16( __mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_cvtsepi64_epi16( __m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16( __mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVSQD __m128i _mm256_cvtusepi64_epi32( __m256i a);
VPMOVSQD __m128i _mm256_mask_cvtusepi64_epi32( __m256i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtusepi64_storeu_epi32( void * d, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtusepi64_epi32( __m128i a);
VPMOVSQD __m128i _mm_mask_cvtusepi64_epi32( __m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtusepi64_epi32( __mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtusepi64_storeu_epi32( void * d, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32( __m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32( __m256i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32( void * d, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_cvtusepi64_epi32( __m256i a);
VPMOVSQD __m128i _mm256_mask_cvtusepi64_epi32( __m256i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtusepi64_storeu_epi32( void * d, __mmask8 k, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 30 /r VPMOVWB xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 8 packed word integers from xmm2 into 8 packed bytes in xmm1/m64 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 20 /r VPMOVSWB xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 8 packed signed word integers from xmm2 into 8 packed signed bytes in xmm1/m64 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m64 [k1][z], xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in 8mm1/m64 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 30 /r VPMOVWB xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 16 packed word integers from ymm2 into 16 packed bytes in xmm1/m128 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 20 /r VPMOVSWB xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 16 packed signed word integers from ymm2 into 16 packed signed bytes in xmm1/m128 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB xmm1/m128 [k1][z], ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 16 packed unsigned word integers from ymm2 into 16 packed unsigned bytes in xmm1/m128 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 30 /r VPMOVWB ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512BW</td>
<td>Converts 32 packed word integers from zmm2 into 32 packed bytes in ymm1/m256 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 20 /r VPMOVSWB ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512BW</td>
<td>Converts 32 packed signed word integers from zmm2 into 32 packed signed bytes in ymm1/m256 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB ymm1/m256 [k1][z], zmm2</td>
<td>A V/V</td>
<td>AVX512BW</td>
<td>Converts 32 packed unsigned word integers from zmm2 into 32 packed unsigned bytes in ymm1/m256 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half Mem</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

**VPMOVWB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
   i ← j * 8
   m ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+7:i] = 0
      ENDIF
   FI;
ENDFOR

DEST[MAXVL-1:VL/2] ← 0;

**VPMOVWB instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
   i ← j * 8
   m ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
   ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
   FI;
ENDFOR

**VPMOVSWB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
   i ← j * 8
   m ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+7:i] = 0
      ENDIF
   FI;
ENDFOR

DEST[MAXVL-1:VL/2] ← 0;
VPMOVSwB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVUSwB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] = 0
      FI
    FI;
ENDFOR

DEST[MAXVL-1:VL/2] ← 0;

VPMOVUSwB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVUSWB __m256i _mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i _mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB void _mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m516i __mm512_cvtsepi16_epi8(__m512i a);
VPMOVUSWB __m516i __mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void _mm512_maskz_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m128i _mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSWB __m128i _mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtusepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm256_cvtsepi16_epi8(__m256i a);
VPMOVUSWB __m128i _mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtsepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm256_cvtepi16_epi8(__m256i a);
VPMOVUSWB __m128i _mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMULTISHIFTQB – Select Packed Unaligned Bytes from Quadword Sources

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512_VBMI AVX512VL</td>
<td>Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 83 /r VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512_VBMI AVX512VL</td>
<td>Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 83 /r VPMULTISHIFTQB zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512_VBMI</td>
<td>Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is extracted from the input qword source at the location specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.
**Operation**

**VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(i \leftarrow 0 \) TO \(KL-1\)

\[\text{IF EVEX.b=1 AND src2 is memory THEN}\]

\[t_{\text{cur}} \leftarrow \text{src2.qword}[0]; \quad //\text{broadcasting}\]

\[\text{ELSE}\]

\[t_{\text{cur}} \leftarrow \text{src2.qword}[i];\]

\[\text{FI};\]

FOR \(j \leftarrow 0 \) TO \(7\)

\[\text{ctrl} \leftarrow \text{src1.qword}[i].\text{byte}[j] \& 63;\]

FOR \(k \leftarrow 0 \) TO \(7\)

\[\text{res.bit}[k] \leftarrow t_{\text{cur}}.\text{bit}[ (\text{ctrl}+k) \mod 64 ];\]

\[\text{ENDFOR}\]

\[\text{IF k1[\ast i+j]} \text{ or no writemask THEN}\]

\[\text{DEST.qword}[i].\text{byte}[j] \leftarrow \text{res};\]

\[\text{ELSE IF zeroing-masking THEN}\]

\[\text{DEST.qword}[i].\text{byte}[j] \leftarrow 0;\]

\[\text{ENDIF}\]

\[\text{ENDFOR}\]

\[\text{DEST.qword}[\text{MAX}_V L-1: V L] \leftarrow 0;\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[\text{VPMULTISHIFTQB } \_\text{m}512i \_\text{mm}512 \_\text{mm}512 \_\text{multishift} \_\text{epi}64 \_\text{epi}8( \_\text{m}512i a, \_\text{m}512i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}512i \_\text{mm}512 \_\text{mask} \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{m}512i s, \_\text{mm}512i k, \_\text{m}512i a, \_\text{m}512i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}512i \_\text{mm}512 \_\text{maskz} \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{mm}512i k, \_\text{m}512i a, \_\text{m}512i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}256i \_\text{mm}256 \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{m}256i a, \_\text{m}256i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}256i \_\text{mm}256 \_\text{mask} \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{m}256i s, \_\text{mm}256i k, \_\text{m}256i a, \_\text{m}256i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}256i \_\text{mm}256 \_\text{maskz} \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{mm}256i k, \_\text{m}256i a, \_\text{m}256i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}128i \_\text{mm}128 \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{m}128i a, \_\text{m}128i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}128i \_\text{mm}128 \_\text{mask} \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{m}128i s, \_\text{mm}128i k, \_\text{m}128i a, \_\text{m}128i b);\]

\[\text{VPMULTISHIFTQB } \_\text{m}128i \_\text{mm}128 \_\text{maskz} \_\text{multishift} \_\text{epi}64 \_\text{epi}8(\_\text{mm}128i k, \_\text{m}128i a, \_\text{m}128i b);\]

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.
## VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.D.128.66.0F38.W0 72 /1 ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.D.128.66.0F38.W1 72 /1 ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.D.256.66.0F38.W0 72 /1 ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.D.256.66.0F38.W1 72 /1 ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.D.512.66.0F38.W0 72 /1 ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate left of doublewords in zmm2/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.D.512.66.0F38.W1 72 /1 ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>_operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:reg/m (R)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

**Operation**

LEFT Rotate DWords (SRC, COUNT_SRC)

COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT Rotate QWords (SRC, COUNT_SRC)

COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC << COUNT) | (SRC >> (64 - COUNT));

VPROLD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask* THEN
   IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← LEFT Rotate DWords(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] ← LEFT Rotate DWords(SRC1[i+31:i], imm8)
   FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
         ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] ← 0
         FI
      FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
**VPROLVD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← LEFT Rotate DWords(SRC1[i+31:i], SRC2[31:0])
            ELSE DEST[i+31:i] ← LEFT Rotate DWords(SRC1[i+31:i], SRC2[i+31:i])
        FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPROLOQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+63:i] ← LEFT Rotate QWords(SRC1[63:0], imm8)
            ELSE DEST[i+63:i] ← LEFT Rotate QWords(SRC1[i+63:i], imm8)
        FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR

DEST[MAXVL-1:VL] ← 0
VPROLVQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← LEFT Rotate QWORDS(SRC1[i+63:i], SRC2[63:0])
            ELSE DEST[i+63:i] ← LEFT Rotate QWORDS(SRC1[i+63:i], SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    END FOR

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPROLD __m512i _mm512_rol_epi32(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi32(__mmask16 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi32(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi32(__mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rol_epi32(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi32(__mmask8 k, __m128i a, int imm);
VPROLD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLD __m512i _mm512_mask_rolv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPROLD __m512i _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPROLD __m256i _mm256_rolv_epi32(__m256i a, __m256i cnt);
VPROLD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLD __m128i _mm_rolv_epi32(__m128i a, __m128i cnt);
VPROLD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPROLD __m512i _mm512_rolv_epi64(__m512i a, __m512i cnt);
VPROLD __m512i _mm512_mask_rolv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPROLD __m512i _mm512_maskz_rolv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPROLD __m256i _mm256_rolv_epi64(__m256i a, __m256i cnt);
VPROLD __m256i _mm256_mask_rolv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPROLD __m128i _mm_rolv_epi64(__m128i a, __m128i cnt);
VPROLD __m128i _mm_mask_rolv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLD __m128i _mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPROLD __m512i _mm512_rolv_epi64(__m512i a, __m512i cnt);
VPROLD __m512i _mm512_mask_rolv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPROLD __m512i _mm512_maskz_rolv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPROLVQ__m128i__mm_mask_grvl_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVQ__m128i__mm_maskz_grvl_epi64(__mmask8 k, __m128i a, __m128i cnt);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.
VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 14 /r VPRORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.ND.128.66.0F0.W0 72 /0 ib VPRORD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 14 /r VPRORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.ND.128.66.0F0.W1 72 /0 ib VPRORQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 14 /r VPRORVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.ND.256.66.0F0.W0 72 /0 ib VPRORVD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 14 /r VPRORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.ND.256.66.0F0.W1 72 /0 ib VPRORQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.ND.512.66.0F0.W0 72 /0 ib VPRORVD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 14 /r VPRORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.ND.512.66.0F0.W1 72 /0 ib VPRORQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>VEX.vvvv (w)</td>
<td>ModRM/r/m (R)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM/reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**
Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

**EVEX.128 encoded version:** The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

**EVEX.256 encoded version:** The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

**EVEX.512 encoded version:** The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

**Operation**
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC >> COUNT) | (SRC << (64 - COUNT));

**VPRORD (EVEX encoded versions)**
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask* THEN
      IF (EVEX.b = 1) AND (SRC1 *is memory*)
         THEN DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[31:0], imm8)
         ELSE DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
      FI;
      ELSE
         IF *merging-masking*;
            merging-masking
         THEN *DEST[i+31:i] remains unchanged*
         ELSE *zeroing-masking*;
            zeroing-masking
               DEST[i+31:i] ← 0
         FI
      FI
   ENDFOR
DEST[MAXVL-1:VL] ← 0
VPRORVD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
            ELSE DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPRORQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[63:0], imm8)
            ELSE DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VPRORVQ (EVEX encoded versions)

KL, VL = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
i ← j * 64
IF k1[] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
ELSE DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
DEST[i+63:i] ← 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPROR m512i _mm512_ror_epi32(__m512i a, int imm);
VPROR m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROR m512i _mm512_maskz_ror_epi32(__mmask16 k, __m512i a, int imm);
VPROR m256i _mm256_ror_epi32(__m256i a, int imm);
VPROR m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROR m256i _mm256_maskz_ror_epi32(__mmask8 k, __m256i a, int imm);
VPROR m128i _mm_ror_epi32(__m128i a, int imm);
VPROR m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROR m128i _mm_maskz_ror_epi32(__mmask8 k, __m128i a, int imm);
VPRORVD m512i _mm512_rorv_epi32(__m512i a, __m512i cnt);
VPRORVD m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORVD m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORVD m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORVD m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVD m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORVD m128i _mm_rorv_epi32(__m128i a, __m128i cnt);
VPRORVD m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVD m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPRORVQ__m128i__mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVQ__m128i__mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E4.
VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:
- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.

Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

This instruction does not perform AC checks, and so will never deliver an AC fault.

Not valid with 16-bit effective addresses. Will deliver a #UD fault.

If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

**VPSCATTERDD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask*
    THEN MEM[B]ASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] ← SRC[i+31:i]
k1[j] ← 0
FI;
ENDFOR

k1[MAX_KL-1:KL] ← 0

**VPSCATTERDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
i ← j * 64
k ← j * 32
IF k1[j] OR *no writemask*
    THEN MEM[B]ASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ← SRC[i+63:i]
k1[j] ← 0
FI;
ENDFOR

k1[MAX_KL-1:KL] ← 0
VPSCATTERQD (EVEX encoded versions)

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ← SRC[i+31:i]
    k1[j] ← 0

ENDFOR
k1[MAX_KL-1:KL] ← 0

VPSCATTERQQ (EVEX encoded versions)

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] ← SRC[i+63:i]
    Fi;
ENDFOR
k1[MAX_KL-1:KL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

VPSCATTERDD void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

SIMD Floating-Point Exceptions

None
Other Exceptions
See Exceptions Type E12.
### VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 47 / r VPSLLVD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 47 / r VPSLLVQ xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 47 / r VPSLLVD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 47 / r VPSLLVQ ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 12 / r VPSLLVW xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 12 / r VPSLLVW ymm1 (k1)[z], ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 12 / r VPSLLVW zmm1 (k1)[z], zmm2, zmm3/m512</td>
<td>B V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 47 / r VPSLLVD xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 47 / r VPSLLVD ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 47 / r VPSLLVD zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 47 / r VPSLLVQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 47 / r VPSLLVQ ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 47 / r VPSLLVQ zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation
VPSLLVW (EVEX encoded version)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j $\leftarrow$ 0 TO KL-1
  i $\leftarrow$ j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] $\leftarrow$ ZeroExtend(SRC1[i+15:i] $\leftarrow$ SRC2[i+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+15:i] $\leftarrow$ 0
      FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] $\leftarrow$ 0;
VPSLLVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[MAXVL-1:128] ← 0;

VPSLLVD (VEX.256 version)
COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] ← 0;
DEST[MAXVL-1:256] ← 0;

VPSLLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
      ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
**VPSLLVQ (VEX.128 version)**

COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] ← 0;
DEST[MAXVL-1:128] ← 0;

**VPSLLVQ (VEX.256 version)**

COUNT_0 ← SRC2[63 : 0];
/* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 */
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
/* Repeat shift operation for 2nd through 4th dwords */
IF COUNT_3 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] ← 0;
DEST[MAXVL-1:256] ← 0;

**VPSLLVQ (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
      ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_sllv_epi16(__m256i m, __m256i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_sllv_epi32(__m256i m, __m256i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVQ __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVQ __m256i _mm256_maskz_sllv_epi64(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVQ __m128i _mm_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded VPSLLVD/VPSLLVQ, see Exceptions Type E4.
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb.
## VPSRAVw/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 46 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 46 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 11 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 11 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 11 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 46 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 46 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 46 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 46 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 46 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 46 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element are filled with the corresponding sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 16 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRAVw (EVEX encoded version)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[\text{FOR } j \leftarrow 0 \text{ TO KL-1} \]
\[\quad i \leftarrow j \times 16 \]
\[\quad \text{IF } k1[j] \text{ OR } \text{no writemask}^* \]
\[\quad \text{THEN} \]
\[\quad \quad \text{COUNT} \leftarrow \text{SRC2}[i+3:i] \]
\[\quad \quad \text{IF COUNT < 16} \]
\[\quad \quad \quad \text{THEN} \quad \text{DEST}[i+15:i] \leftarrow \text{SignExtend(SRC1}[i+15:i] \gg \text{COUNT}) \]
\[\quad \quad \quad \text{ELSE} \]
\[\quad \quad \quad \quad \text{FOR } k \leftarrow 0 \text{ TO 15} \]
\[\quad \quad \quad \quad \quad \text{DEST}[i+k] \leftarrow \text{SRC1}[i+15] \]
\[\quad \quad \quad \quad \end{FOR} \]
\[\quad \quad \Fi \]
\[\quad \quad \text{ELSE} \]
\[\quad \quad \quad \text{IF } \text{*merging-masking}^* \quad \text{; merging-masking} \]
\[\quad \quad \quad \quad \text{THEN } *\text{DEST}[i+15:i] \text{ remains unchanged}^* \]
\[\quad \quad \quad \quad \text{ELSE} \quad \text{; zeroing-masking} \]
\[\quad \quad \quad \quad \text{DEST}[i+15:i] \leftarrow 0 \]
\[\quad \quad \Fi \]
\[\quad \text{ENDFOR}; \]
\[\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0; \]

VPSRAVw/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic
VPSRAVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0]
   (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 ← SRC2[100 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
   (* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] ← 0;

VPSRAVD (VEX.256 version)
COUNT_0 ← SRC2[31 : 0];
   (* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2 *)
COUNT_7 ← SRC2[228 : 224];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
   (* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAXVL-1:256] ← 0;

VPSRAVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask* THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
         THEN
            COUNT ← SRC2[4:0]
            IF COUNT < 32
               THEN
                  DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
               ELSE
                  FOR k← 0 TO 31
                     DEST[i+k] ← SRC1[i+31]
                  ENDFOR;
               FI
            ELSE
               COUNT ← SRC2[i+4:i]
               IF COUNT < 32
                  THEN
                     DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
                  ELSE
                     FOR k← 0 TO 31
                        DEST[i+k] ← SRC1[i+31]
                     ENDFOR;
               FI
         ELSE
            FI
      FI
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[31:0] remains unchanged* 
      ELSE ; zeroing-masking
         DEST[31:0] ← 0
      FI
   FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
VPSRAVQ (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                COUNT ← SRC2[5:0]
                IF COUNT < 64
                    THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] >> COUNT)
                    ELSE
                        FOR k ← 0 TO 63
                            DEST[i+k] ← SRC1[i+63]
                        ENDFOR;
                FI
            ELSE
                COUNT ← SRC2[i+5:i]
                IF COUNT < 64
                    THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] >> COUNT)
                    ELSE
                        FOR k ← 0 TO 63
                            DEST[i+k] ← SRC1[i+63]
                        ENDFOR;
                FI
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[63:0] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[63:0] ← 0
                    FI
                FI
        FI
    ELSE
    ENDFOR;
DEST[MAXVL-1:VL] ← 0;
**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRAVD

```
__m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
__m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
__m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
__m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
__m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
__m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
__m128i _mm_srav_epi32(__m128i a, __m128i cnt);
__m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
__m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
```

VPSRAVQ

```
__m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
__m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
__m512i _mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
__m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
__m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
__m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
__m128i _mm_srav_epi64(__m128i a, __m128i cnt);
__m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
__m128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
```

VPSRAVW

```
__m512i _mm512_srav_epi16(__m512i a, __m512i cnt);
__m512i _mm512_mask_srav_epi16(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
__m512i _mm512_maskz_srav_epi16(__mmask16 m, __m512i a, __m512i cnt);
__m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
__m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
__m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
__m128i _mm_srav_epi16(__m128i a, __m128i cnt);
__m128i _mm_mask_srav_epi16(__m128i s, __mmask16 m, __m128i a, __m128i cnt);
__m128i _mm_maskz_srav_epi16(__mmask16 m, __m128i a, __m128i cnt);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.
### VPSRLVw/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.w0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V (r)</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.w1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V (r)</td>
<td>AVX2</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.w0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V (r)</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.w1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V (r)</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.w1 10 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V (r)</td>
<td>AVX512VL, AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.w1 10 /r VPSRLW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V (r)</td>
<td>AVX512VL, AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.w1 10 /r VPSRLW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V (r)</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.w0 45 /r VPSRLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V (r)</td>
<td>AVX512VL, AVX512F</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.w0 45 /r VPSRLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V (r)</td>
<td>AVX512VL, AVX512F</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.w0 45 /r VPSRLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V (r)</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.w1 45 /r VPSRLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V (r)</td>
<td>AVX512VL, AVX512F</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.w1 45 /r VPSRLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V (r)</td>
<td>AVX512VL, AVX512F</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.w1 45 /r VPSRLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V (r)</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation**

**VPSRLVW (EVEX encoded version)**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)

\[i \leftarrow j \times 16\]

IF \(k1[j] \text{ OR } \text{no writemask}\)

\[\text{THEN } \text{DEST}[i+15:i] \leftarrow \text{ZeroExtend}(\text{SRC1}[i+15:i] \gg \text{SRC2}[i+15:i])\]

ELSE

\[\text{IF } \text{merging-masking}; \text{merging-masking}\]

\[\text{THEN } \text{DEST}[i+15:i] \text{ remains unchanged}\]

ELSE

\[\text{zeroing-masking}\]

\[\text{DEST}[i+15:i] \leftarrow 0\]

FI

ENDFOR;

\[\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0;\]

**VPSRLVD (VEX.128 version)**

\[\text{COUNT}_0 \leftarrow \text{SRC2}[31 : 0]\]

\[\text{(* Repeat Each COUNT}_i \text{ for the 2nd through 4th dwords of SRC2*)}\]

\[\text{COUNT}_3 \leftarrow \text{SRC2}[127 : 96];\]

IF \(\text{COUNT}_0 < 32 \text{ THEN}\)

\[\text{DEST}[31:0] \leftarrow \text{ZeroExtend}((\text{SRC1}[31:0] \gg \text{COUNT}_0);\]

ELSE

\[\text{DEST}[31:0] \leftarrow 0;\]

\[\text{(* Repeat shift operation for 2nd through 4th dwords *)}\]

IF \(\text{COUNT}_3 < 32 \text{ THEN}\)

\[\text{DEST}[127:96] \leftarrow \text{ZeroExtend}((\text{SRC1}[127:96] \gg \text{COUNT}_3);\]

ELSE

\[\text{DEST}[127:96] \leftarrow 0;\]

\[\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0;\]
VPSRLVD (VEX.256 version)
COUNT_0 ← SRC2[31 : 0];
 (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
  DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
  DEST[31:0] ← 0;
  (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
  DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
  DEST[255:224] ← 0;
DEST[MAXVL-1:256] ← 0;

VPSRLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
      ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;

VPSRLVQ (VEX.128 version)
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
  DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
  DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
  DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
  DEST[127:64] ← 0;
DEST[MAXVL-1:128] ← 0;
VPSRLVQ (VEX.256 version)
COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
  DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
  DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
  DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
  DEST[255:192] ← 0;
DEST[MAXVL-1:256] ← 0;
VPSRLVQ (EVEX encoded version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
      ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_srlv_epi16(__m256i s, __m256i cnt);
VPSRLVW __m256i _mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i _mm256_maskz_srlv_epi16(__m256i a, __m256i cnt);
VPSRLVW __m128i _mm128i_srlv_epi16(__m128i s, __m128i cnt);
VPSRLVW __m128i _mm128i_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m128i _mm128i_maskz_srlv_epi16(__mmask8 k, __m128i a, __m128i cnt);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded VPSRLVD/Q, see Exceptions Type E4.
EVEX-encoded VPSRLVWl, see Exceptions Type E4.nb.
### VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 [k1][z], xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 [k1][z], ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 [k1][z], zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMzr/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.
Table 5-9 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

<table>
<thead>
<tr>
<th>VPTERNLOGD reg1, reg2, src3, 0xE2</th>
<th>Bit Result with Imm8=0xE2</th>
<th>VPTERNLOGD reg1, reg2, src3, 0xE4</th>
<th>Bit Result with Imm8=0xE4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit(reg1)</td>
<td>Bit(reg2)</td>
<td>Bit(src3)</td>
<td>Bit(reg1)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

**Operation**

**VPTERNLOGD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            FOR k ← 0 TO 31
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
                    ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
                FI;
                ; table lookup of immediate bellow;
        ELSE
            IF *merging-masking*
                THEN *DEST[31+i][i] remains unchanged*
            ELSE
                ; zeroing-masking
                DEST[31+i][i] ← 0
            FI;
        FI;
    END FOR;
DEST[MAXVL-1:VL] ← 0
VPTERNLOGQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      FOR k ← 0 TO 63
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
          THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
          ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[63+i:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[63+i:i] ← 0
        FI;
      FI;
    ENDFOR;
  DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents
VPTERNLOGD __m512i _mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_maskz_ternarylogic_epi32(__mmask s, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_maskz_ternarylogic_epi32(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_maskz_ternarylogic_epi32(__mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i _mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_maskz_ternarylogic_epi64(__mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i _mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_mask_ternarylogic_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_maskz_ternarylogic_epi64(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i _mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_mask_ternarylogic_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_maskz_ternarylogic_epi64(__mmask8 m, __m128i a, __m128i b, int imm);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E4.
### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512</td>
<td>A/V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512</td>
<td>A/V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

**VPTESTMD/VPTESTMQ:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

**VPTESTMB/VPTESTMW:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

**Operation**

**VPTESTMB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
   i ← j * 8
   IF k1[j] OR *no writemask*
      THEN DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
      ELSE DEST[j] = 0 ; zeroing-masking only
   FI;
ENDFOR

DEST[MAX_KL-1:KL] ← 0

**VPTESTMW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
      ELSE DEST[j] = 0 ; zeroing-masking only
   FI;
ENDFOR

DEST[MAX_KL-1:KL] ← 0

**VPTESTMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
            ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
         FI;
      ELSE DEST[j] ← 0 ; zeroing-masking only
   FI;
ENDFOR

DEST[MAX_KL-1:KL] ← 0
VPTESTMQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
      ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
    FI;
  ELSE DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents
VPTESTMB __mmask64 _mm512_test_epi8_mask( __m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask( __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask( __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask(__m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask(__mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
VPTESTMD/Q: See Exceptions Type E4.
VPTESTMB/W: See Exceptions Type E4.nb.
## VPTESTNMB/W/D/Q—Logical NAND and Set

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.F3.0F38.W0 26 /r VPTESTNMB k2 [k1], xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W0 26 /r VPTESTNMB k2 [k1], ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W0 26 /r VPTESTNMB k2 [k1], zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512BW</td>
<td>Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.F3.0F38.W1 26 /r VPTESTNMW k2 [k1], xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W1 26 /r VPTESTNMW k2 [k1], ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W1 26 /r VPTESTNMW k2 [k1], zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512BW</td>
<td>Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.F3.0F38.W0 27 /r VPTESTNMD k2 [k1], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W0 27 /r VPTESTNMD k2 [k1], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W0 27 /r VPTESTNMD k2 [k1], zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.F3.0F38.W1 27 /r VPTESTNMQ k2 [k1], xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W1 27 /r VPTESTNMQ k2 [k1], ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W1 27 /r VPTESTNMQ k2 [k1], zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F AVX512F</td>
<td>Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
</tbody>
</table>
### Description
Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

### Operation
#### VPTESTNMB
\[
(\text{KL}, \text{VL}) = (16, 128), (32, 256), (64, 512)
\]
```
FOR j ← 0 TO KL-1
    i ← j*8
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

#### VPTESTNMDw
\[
(\text{KL}, \text{VL}) = (8, 128), (16, 256), (32, 512)
\]
```
FOR j ← 0 TO KL-1
    i ← j*16
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```
VPTESTNMD

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0
                ELSE DEST[i+j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:j] == 0)? 1 : 0
            FI
        ELSE DEST[i+j] ← 0; zeroing masking only
        FI
    ENDFOR

DEST[MAX_KL-1:KL] ← 0

VPTESTNMQ

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0
                ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0
            FI
        ELSE DEST[j] ← 0; zeroing masking only
        FI
    ENDFOR

DEST[MAX_KL-1:KL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPTESTNMB __mmask64 __mm512_testn_epi8_mask(__m512i a, __m512i b);
VPTESTNMB __mmask64 __mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 __mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 __mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 __mm128_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 __mm128_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMB __mmask8 __mm64_testn_epi8_mask(__m64i a, __m64i b);
VPTESTNMB __mmask8 __mm64_mask_testn_epi8_mask(__mmask8, __m64i a, __m64i b);
VPTESTNMQ __mmask16 __mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask16 __mm512_mask_testn_epi16_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMQ __mmask8 __mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm256_mask_testn_epi16_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm128_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMQ __mmask8 __mm128_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 __mm64_testn_epi16_mask(__m64i a, __m64i b);
VPTESTNMQ __mmask8 __mm64_mask_testn_epi16_mask(__mmask8, __m64i a, __m64i b);
VPTESTNMQ __mmask8 __mm32_testn_epi16_mask(__m32i a, __m32i b);
VPTESTNMQ __mmask8 __mm32_mask_testn_epi16_mask(__mmask8, __m32i a, __m32i b);
VPTESTNMQ __mmask8 __mm16_testn_epi16_mask(__m16i a, __m16i b);
VPTESTNMQ __mmask8 __mm16_mask_testn_epi16_mask(__mmask8, __m16i a, __m16i b);
VPTESTNMQ __mmask8 __mm8_testn_epi16_mask(__m8i a, __m8i b);
VPTESTNMQ __mmask8 __mm8_mask_testn_epi16_mask(__mmask8, __m8i a, __m8i b);
VPTESTNMQ __mmask8 __mm4_testn_epi16_mask(__m4i a, __m4i b);
VPTESTNMQ __mmask8 __mm4_mask_testn_epi16_mask(__mmask8, __m4i a, __m4i b);
VPTESTNMQ __mmask8 __mm2_testn_epi16_mask(__m2i a, __m2i b);
VPTESTNMQ __mmask8 __mm2_mask_testn_epi16_mask(__mmask8, __m2i a, __m2i b);
VPTESTNMQ __mmask8 __mm1_testn_epi16_mask(__m1i a, __m1i b);
VPTESTNMQ __mmask8 __mm1_mask_testn_epi16_mask(__mmask8, __m1i a, __m1i b);
VPTESTNMQ __mmask8 __mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VPTESTNMD/VPTESTNMQ: See Exceptions Type E4.
VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb.
VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 50 / r lb VRANGEPD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate two RANGE operation output value from 2 pairs of double-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 50 / r lb VRANGEPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate four RANGE operation output value from 4 pairs of double-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 50 / r lb VRANGEPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate eight RANGE operation output value from 8 pairs of double-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMrsr/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

![Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS](image-url)
When one or more of the input value is a NaN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2] has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
<th>IE Signaling Due to Comparison</th>
<th>Imm8[3:2] Effect to Range Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sNaN1</td>
<td>sNaN2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>sNaN1</td>
<td>qNaN2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>sNaN1</td>
<td>Norm2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>qNaN1</td>
<td>sNaN2</td>
<td>Quiet(sNaN2)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>qNaN1</td>
<td>qNaN2</td>
<td>qNaN1</td>
<td>No</td>
<td>Applicable</td>
</tr>
<tr>
<td>qNaN1</td>
<td>Norm2</td>
<td>Norm2</td>
<td>No</td>
<td>Applicable</td>
</tr>
<tr>
<td>Norm1</td>
<td>sNaN2</td>
<td>Quiet(sNaN2)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>Norm1</td>
<td>qNaN2</td>
<td>Norm1</td>
<td>No</td>
<td>Applicable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td>+0</td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td>+0</td>
</tr>
</tbody>
</table>

| MIN_ABS (|a| = |b|, a>0, b<0) | MAX_ABS (|a| = |b|, a>0, b<0) |
|----------|--------------|
| Sr1 | Sr2 | Result | Sr1 | Sr2 | Result |
| a | b | b | a | b | a |
| b | a | b | b | a | a |
Operation

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-10
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction != 0)) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-11
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-12
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[63] << 63) OR (TMP[62:0])// Preserve Src1 sign bit
    01: dest ← TMP[63:0];// Preserve sign of compare result
    10: dest ← (0 << 63) OR (TMP[62:0])// Zero out sign bit
    11: dest ← (1 << 63) OR (TMP[62:0])// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGEPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b == 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
            ELSE DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*;
        ELSE ; zeroing-masking
            DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:
    zmm_dst is the destination operand.
    zmm_src is the input operand to compare against ±1023 (this is SRC1).
    zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).
    IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e. SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_src > +1023, the result of VRANGE will be the maximal value of +1023.
Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPD __m512d_mm512_range_pd (__m512d a, __m512d b, int imm);
VRANGEPD __m512d_mm512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d_mm512_mask_range_pd (__m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d_mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d_mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d_mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d_mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d_mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d_mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d_mm128_range_pd (__m128d a, __m128d b, int imm);
VRANGEPD __m128d_mm128_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d_mm128_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.
VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst[5ae], imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.
Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-10
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN // Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN // Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;
    IF (SRC2 = QNAN) THEN {TMP[31:0] ← SRC1[31:0]}
    ELSE IF (SRC1 = QNAN) THEN {TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-11
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-12
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]); // Preserve Src1 sign bit
    01: dest ← TMP[31:0]; // Preserve sign of compare result
    10: dest ← (0 <= 31) OR (TMP[30:0]); // Zero out sign bit
    11: dest ← (1 <= 31) OR (TMP[30:0]); // Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGEPS

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

\[ \text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]
\[ i \leftarrow j \times 32 \]
\[ \text{IF } k1[j] \text{ OR } \text{no writemask} \text{ THEN} \]
\[ \text{IF } (EVEX.b == 1) \text{ AND } (SRC2 \text{ is memory}) \]
\[ \text{THEN } \text{DEST}[i+31:i] \leftarrow \text{RangeSP}(\text{SRC1}[i+31:i], \text{SRC2}[31:0], \text{CmpOpCtl}[1:0], \text{SignSelCtl}[1:0]); \]
\[ \text{ELSE } \text{DEST}[i+31:i] \leftarrow \text{RangeSP}(\text{SRC1}[i+31:i], \text{SRC2}[i+31:i], \text{CmpOpCtl}[1:0], \text{SignSelCtl}[1:0]); \]
\[ \text{FI;} \]
\[ \text{ELSE} \]
\[ \text{IF } \text{merging-masking}; \text{merging-masking} \]
\[ \text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged} \]
\[ \text{ELSE}; \text{zeroing-masking} \]
\[ \text{DEST}[i+31:i] = 0 \]
\[ \text{FI;} \]
\[ \text{FI;} \]
\[ \text{ENDFOR} ; \]
\[ \text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0 \]

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

```
VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

- \text{zmm\textunderscore dst} \text{is} the destination operand.
- \text{zmm\textunderscore src} \text{is} the input operand to compare against ±150.
- \text{zmm\textunderscore 150} \text{is} the reference operand, contains the value of 150.
- \text{IMM=02(imm8[1:0]=’10)} \text{selects the Min Absolute value operation with selection of src1.sign.}

\text{In case} \ |\text{zmm\textunderscore src}| < 150, \text{then its value will be written into zmm\textunderscore dst. Otherwise, the value stored in zmm\textunderscore dst will get the value of 150 (received on zmm\textunderscore 150).}
\text{However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from zmm\textunderscore src. So, even in the case of} \ |\text{zmm\textunderscore src}| \geq 150, \text{the selected sign of SRC1 is kept.}
\text{Thus, if zmm\textunderscore src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm\textunderscore src > +150, the result of VRANGE will be the maximal value of +150.}
Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPS __m512 __m512_range_ps (__m512 a, __m512 b, int imm);
VRANGEPS __m512 __m512_range_round_ps (__m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 __m512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 __m512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 __m512_maskz_range_ps (__mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 __m512_maskz_range_round_ps (__mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 __m256_range_ps (__m256 a, __m256 b, int imm);
VRANGEPS __m256 __m256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 __m256_maskz_range_ps (__mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 __m128_range_ps (__m128 a, __m128 b, int imm);
VRANGEPS __m128 __m128_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 __m128_maskz_range_ps (__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.
**VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.66.0F3A.W1 51 /r</td>
<td>A</td>
<td>V</td>
<td>AVX512DQ</td>
<td>Calculate a RANGE operation output value from 2 double-precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

This instruction calculates a range operation output from two input double-precision FP values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NaN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NaN is listed in Table 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEDP/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.
Operation

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-10
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN // Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction != 0)) THEN // Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN {TMP[63:0] ← SRC1[63:0]}
    ELSE IF (SRC1 = QNAN) THEN {TMP[63:0] ← SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-11
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-12
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[63] << 63) OR (TMP[62:0]); // Preserve Src1 sign bit
    01: dest ← TMP[63:0]; // Preserve sign of compare result
    10: dest ← (0 << 63) OR (TMP[62:0]); // Zero out sign bit
    11: dest ← (1 << 63) OR (TMP[62:0]); // Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGESD

If k1[0] OR *no writemask*
   THEN DEST[63:0] ← RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
   ELSE
      IF *merging-masking* ; merging-mask
         THEN *DEST[63:0] remains unchanged*
      ELSE ; zeroing-mask
         DEST[63:0] = 0
      FI;
   FI;

DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;

Where:
 xmm_dst is the destination operand.
 xmm_src is the input operand to compare against ±1023.
 xmm_1023 is the reference operand, contains the value of 1023.
 IMM=02(imm8[1:0]=’10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 1023, then its value will be written into xmm_dst. Otherwise, the value stored in xmm_dst will get the value of 1023 (received on xmm_1023).

However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if xmm_src < -1023, the result of VRANGE will be the minimal value of -1023 while if xmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESD __m128d _mm_range_sd ( __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 51 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction calculates a range operation output from two input single-precision FP values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.
Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-10
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF (Src1.exp = 0 ) and (Src1.fraction != 0 ) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF (Src2.exp = 0 ) and (Src2.fraction != 0 ) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-11
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-12
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGESS

IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
        DEST[31:0] = 0
FI;
FI;
DEST[MAXVL-1:128] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGESS zmm_dst, zmm_src, zmm_150, 02h;

Where:
 xmm_dst is the destination operand.
 xmm_src is the input operand to compare against ±150.
 xmm_150 is the reference operand, contains the value of 150.
 IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in xmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if xmm_src < -150, the result of VRANGESS will be the minimal value of -150 while if xmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESS __m128 _mm_range_ss ( __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.
VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 4C /r VRCP14PD xmm1 [k1]{z}, xmm2/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 4C /r VRCP14PD ymm1 [k1]{z}, ymm2/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 4C /r VRCP14PD zmm1 [k1]{z}, zmm2/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocals of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Full</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand. The maximum relative error for this approximation is less than 2^-14.

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ±∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-13. VRCP14PD/VRCP14SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ≤ X ≤ 2^-1024</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>-2^-1024 ≤ X ≤ -0</td>
<td>-INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>X &gt; 2^1022</td>
<td>Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>X &lt; -2^1022</td>
<td>-Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>X = 2^n</td>
<td>2^n</td>
<td></td>
</tr>
<tr>
<td>X = -2^n</td>
<td>-2^n</td>
<td></td>
</tr>
</tbody>
</table>

* in this case the mantissa is shifted right by one or two bits

Operation

**VRCP14PD ((EVEX encoded versions))**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \]
\[i \leftarrow j \times 64 \]
\[\text{IF } k1[j] \text{ OR } \text{no writemask} \text{ THEN} \]
\[\text{IF (EVEX.b = 1) AND (SRC is memory*)} \]
\[\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[63:0]); \]
\[\text{ELSE } \text{DEST}[i+63:i] \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[i+63:i]); \]
\[\text{FI;} \]
\[\text{ELSE} \]
\[\text{IF } \text{merging-masking*} \quad ; \text{merging-masking} \]
\[\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged*} \]
\[\text{ELSE} \quad ; \text{zeroing-masking} \]
\[\text{DEST}[i+63:i] \leftarrow 0 \]
\[\text{FI;} \]
\[\text{FI;} \]
\[\text{ENDFOR;} \]
\[\text{DEST}[\text{MAXVL-1:VL}] \leftarrow 0 \]

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PD __m512d _mm512_rcp14_pd( __m512d a);
VRCP14PD __m512d _mm512_mask_rcp14_pd(__m512d s, __mmask8 k, __m512d a);
VRCP14PD __m512d _mm512_maskz_rcp14_pd( __mmask8 k, __m512d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 4D / r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-13 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


**Operation**

**VRCP14SD (EVEX version)**

IF k1[0] OR *no writemask*

THEN DEST[63:0] ← APPROXIMATE(1.0/SRC2[63:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged* 

ELSE ; zeroing-masking

DEST[63:0] ← 0 

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VRCP14SD __m128d _mm_rcp14_sd( __m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd( __mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E5.
**VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 4C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 4C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 4C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than $2^{-14}$.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-14. VRCP14PS/VRCP14SS Special Cases**

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq X \leq 2^{-128}$</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$-2^{-128} \leq X \leq -0$</td>
<td>-INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$X &gt; 2^{126}$</td>
<td>Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>$X &lt; -2^{126}$</td>
<td>-Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>$X = 2^n$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X = -2^n$</td>
<td>$-2^n$</td>
<td></td>
</tr>
</tbody>
</table>

* in this case the mantissa is shifted right by one or two bits

Operation

**VRCP14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

\[ i \leftarrow j \times 32 \]

IF k1[j] OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN DEST[i+31:i] ← APPROXIMATE(1.0/SRC[31:0]);

ELSE DEST[i+31:i] ← APPROXIMATE(1.0/SRC[i+31:i]);

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PS __m512 _mm512_rcp14_ps( __m512 a);
VRCP14PS __m512 __mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 __mm512_maskz_rcp14_ps( __mmask16 k, __m512 a);
VRCP14PS __m256 __mm256_rcp14_ps( __m256 a);
VRCP14PS __m256 __mm256_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 __mm256_maskz_rcp14_ps( __mmask8 k, __m256 a);
VRCP14PS __m128 __mm128_rcp14_ps( __m128 a);
VRCP14PS __m128 __mm128_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 __mm128_maskz_rcp14_ps( __mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.
VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LiG.66.0F38.W0 4D /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:32]) of xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-14 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


Operation

**VRCP14SS (EVEX version)**

IF k1[0] OR *no writemask*

THEN DEST[31:0] ← APPROXIMATE(1.0/SRC2[31:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;


DEST[MAXVL-1:128] ← 0

VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value
Intel C/C++ Compiler Intrinsic Equivalent
VRCP14SS __m128_mm_rcp14_ss(__m128 a, __m128 b);
VRCP14SS __m128_mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128_mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E5.
VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Op / Instruction | Op / En | 64/32 bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
EVEX.128.66.0F3A.W1 56 / r ib | A | V/V | AVX512VL AVX512DQ | Perform reduction transformation on packed double-precision floating point values in xmm2/m128/m64bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.

EVEX.256.66.0F3A.W1 56 / r ib | A | V/V | AVX512VL AVX512DQ | Perform reduction transformation on packed double-precision floating point values in ymm2/m256/m64bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.

EVEX.512.66.0F3A.W1 56 / r ib | A | V/V | AVX512DQ | Perform reduction transformation on double-precision floating point values in zmm2/m512/m64bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Perform reduction transformation of the packed binary encoded double-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is an unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src - (ROUND(2^M*src))*2^-M;

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2, where 'man2' is the normalized significand and 'p' is the unbiased exponent.

Then if RC = RNE: 0<=|Reduced Result|<=2^p-M-1

Then if RC ≠ RNE: 0<=|Reduced Result|<2^p-M

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

![Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS](image-url)
Handling of special case of input values are listed in Table 5-15.

| $|\text{Src1}| < 2^{-M-1}$ | Round Mode | Returned value |
|------------------|-------------|--------------|
| $|\text{Src1}| > 2^{-M}$ | RPI, $\text{Src1} > 0$ | Round ($\text{Src1} - 2^{-M}$) * |
| $|\text{Src1}| = 0$, or $\text{Dest} = \pm \text{INF}$ | any | +0.0 |
| $|\text{Src1}| = \pm \text{NAN}$ | n/a | QNaN($\text{Src1}$) |

* Round control = (imm8[7:4])? MXCSR.RC: imm8.RC

**Operation**

ReduceArgumentDP($\text{SRC}[63:0], \text{imm8}[7:0]$)

```plaintext
// Check for NaN
IF (SRC[63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;

M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
RC ← imm8[1:0]; // Round Control for ROUND() operation
RC source ← imm[2];
SPE ← 0; // Suppress Precision Exception
TMP[63:0] ← $2^{M} \times \text{ROUND}(2^{M} \times \text{SRC}[63:0], \text{SPE}, \text{RC_source}, \text{RC})$; // ROUND() treats SRC and $2^{M}$ as standard binary FP values
TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
```

**VREDUCEPD**

$\text{KL, VL)} = (2, 128), (4, 256), (8, 512)$

FOR $j \leftarrow 0$ TO $\text{KL} - 1$

    $i \leftarrow j \times 64$

    IF $k1[j]$ OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+63:j] ← ReduceArgumentDP($\text{SRC}[i+63:j], \text{imm8}[7:0]$);
            ELSE DEST[i+63:j] ← ReduceArgumentDP($\text{SRC}[i+63:j], \text{imm8}[7:0]$);
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:j] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:j] = 0
                FI;
            FI;
    ELSE
        FI;
    ENDFOR;

DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VREducePD __m512d __mm512_mask_reduce_pd(__m512d a, int imm, int sae)
VREducePD __m512d __mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREducePD __m512d __mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREducePD __m256d __mm256_mask_reduce_pd(__m256d a, int imm)
VREducePD __m256d __mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREducePD __m256d __mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREducePD __m128d __mm_mask_reduce_pd(__m128d a, int imm)
VREducePD __m128d __mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREducePD __m128d __mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Exceptions Type E2, additionally
#UD If EVEX.vvvv != 1111B.
VREUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG,66.0F3A.W1 57</td>
<td>A</td>
<td>V/V</td>
<td>AVX512 DQ</td>
<td>Perform a reduction transformation on a scalar double-precision floating point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Perform a reduction transformation of the binary encoded double-precision FP value in the low qword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[ \text{dest} = \text{src} - (ROUND(2^M*}\text{src})*2^{-M}; \]

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering \( \text{src} = 2^p*\text{man2} \), where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent.

Then if RC = RNE: \( 0 \leq |\text{Reduced Result}| \leq 2^{p-M-1} \)

Then if RC ≠ RNE: \( 0 \leq |\text{Reduced Result}| < 2^{p-M} \)

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-15.

**Operation**

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```c
{  // Check for NaN
  IF (SRC[63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[63:0] ← 2^M * [ROUND(2^M*SRC[63:0], SPE, RC_source, RC)]; // ROUND() treats SRC and 2^M as standard binary FP values
  TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RCSPE controls
  RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```
VREDUCESD

IF k1[0] or *no writemask*
    THEN DEST[63:0] ← ReduceArgumentDP(SRC2[63:0], imm8[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[63:0] = 0
    FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCESD __m128d _mm_mask_reduce_sd(__m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

**SIMD Floating-Point Exceptions**

Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**
See Exceptions Type E3.
**VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 56 /r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 56 /r ib VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 56 /r ib VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Perform reduction transformation of the packed binary encoded single-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M \times \text{src})) \times 2^{-M},
\]

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering \( \text{src} = 2^p \times \text{man2} \), where \'man2\' is the normalized significand and \'p\' is the unbiased exponent.

Then if \( \text{RC} = \text{RNE} \): \( 0 <= |\text{Reduced Result}| <= 2^{p-M-1} \)

Then if \( \text{RC} \neq \text{RNE} \): \( 0 <= |\text{Reduced Result}| < 2^{p-M} \)

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-15.
Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
  // Check for NaN
  IF (SRC[31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[31:0] ← 2^M * ROUND(2^M * SRC[31:0], SPE, RC_source, RC); // ROUND() treats SRC and 2^M as standard binary FP values
  TMP[31:0] ← SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}

VREDUCEPS
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] ← ReduceArgumentSP(SRC[31:0], imm8[7:0]);
      ELSE DEST[i+31:i] ← ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] = 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPS __m512 _mm512_mask_reduce_ps( __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 _mm256_mask_reduce_ps( __m256 a, int imm)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 _mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps( __m128 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 _mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2, additionally
#UD IF EVEX.vvvv ! = 1111B.
VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 57</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform a reduction transformation on a scalar single-precision floating point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Perform a reduction transformation of the binary encoded single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is an unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M*\text{src}))*2^{-M};$$

where “Round()” treats “src”, “2^M”, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2, where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent.

Then if RC = RNE: 0<=|Reduced Result|<=2^p*M-1

Then if RC ≠ RNE: 0<|Reduced Result|<2^p*M

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-15.

**Operation**

```c
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC[31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[31:0] ← 2^-M * ROUND(2^M*SRC[31:0], SPE, RC_source, RC); // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```
VREDECSS
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← ReduceArgumentSP(SRC2[31:0], imm8[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] = 0
        FI;
    FI;
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VREDECSS __m128_mm_mask_reduce_ss(__m128 a, __m128 b, int imm, int sae)
VREDECSS __m128_mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDECSS __m128_mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

SIMD Floating-Point Exceptions
Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Exceptions Type E3.
VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 09 / r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed double-precision floating-point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 09 / r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed double-precision floating-point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 09 / r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds packed double-precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Round the double-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}\_\text{to}\_\text{INT}(x \times 2^M, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).
VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

```
<table>
<thead>
<tr>
<th>imm8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SPE</td>
<td>RS</td>
<td>Round Control Override</td>
</tr>
</tbody>
</table>
```

- `Imm8[7:4]` : Number of fixed points to preserve
- `Imm8[3]` : Suppress Precision Exception: `Imm8[3]`
  - `Imm8[3] = 0b` : Use MXCSR exception mask
  - `Imm8[3] = 1b` : Suppress
  - `Imm8[2] = 0b` : Use `Imm8[1:0]`
  - `Imm8[2] = 1b` : Use MXCSR
- `Imm8[1:0]` : Round nearest even
  - `Imm8[1:0] = 00b` : Round nearest even
  - `Imm8[1:0] = 01b` : Round down
  - `Imm8[1:0] = 10b` : Round up
  - `Imm8[1:0] = 11b` : Truncate

Handling of special case of input values are listed in Table 5-16.

<table>
<thead>
<tr>
<th>Src1=±inf</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1=±NAN</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1 converted to QNAN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1=±0</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>
Operation
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    M ← imm8[7:4] ; get the scaling factor
    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC
    Dest[63:0] ← 2^M* TMP[63:0] ; scale down back to 2^-M
    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
return(Dest[63:0])
}

VRNDSCALEPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF *src is a memory operand*
    THEN TMP_SRC ← BROADCAST64(SRC, VL, k1)
ELSE TMP_SRC ← SRC
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPD __m512d _mm512_roundscale_pd( __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd( __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd( __m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd( __m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd( __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd( __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd( __m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd( __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd( __m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd( __m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd( __mmask8 k, __m128d a, int imm);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2.
### Description
Rounds a double-precision floating-point value in the low quadword (see Figure 5-29) element the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the third operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALES is

\[
\text{ROUND}(x) = 2^{-M}\text{Round}_t\text{o}_\text{INT}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALES is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

\[
\text{ROUND}(x) = \text{Round}_t\text{o}_\text{INT}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-16.
**Operation**

\[
\text{RoundToIntegerDP}(\text{SRC}[63:0], \text{imm8}[7:0]) \{
\text{if (imm8}[2] = 1)
\quad \text{rounding_direction} \leftarrow \text{MXCSR:RC} ; \text{get round control from MXCSR}
\text{else}
\quad \text{rounding_direction} \leftarrow \text{imm8}[1:0] ; \text{get round control from imm8}[1:0]
\}
\]

\[
\text{M} \leftarrow \text{imm8}[7:4] ; \text{get the scaling factor}
\]

\[
\text{case (rounding_direction)}
\quad 00: \text{TMP}[63:0] \leftarrow \text{round_to_nearest_even_integer}(2^M \times \text{SRC}[63:0])
\quad 01: \text{TMP}[63:0] \leftarrow \text{round_to_equal_or_smaller_integer}(2^M \times \text{SRC}[63:0])
\quad 10: \text{TMP}[63:0] \leftarrow \text{round_to_equal_or_larger_integer}(2^M \times \text{SRC}[63:0])
\quad 11: \text{TMP}[63:0] \leftarrow \text{round_to_nearest_smallest_magnitude_integer}(2^M \times \text{SRC}[63:0])
\]

\[
\text{ESAC}
\]

\[
\text{Dest}[63:0] \leftarrow 2^{-M} \times \text{TMP}[63:0] ; \text{scale down back to } 2^{-M}
\]

\[
\text{if (imm8}[3] = 0) \text{ Then } ; \text{check SPE}
\quad \text{if (SRC}[63:0] \neq \text{Dest}[63:0]) \text{ Then } ; \text{check precision lost}
\quad \text{set_precision}() ; \text{set #PE}
\]

\[
\text{FI;}
\]

\[
\text{return(Dest}[63:0])
\}
\]

**VRNDSCALESD (EVEX encoded version)**

\[
\text{IF k1}[0] \text{ or } \text{"no writemask"}
\quad \text{THEN } \text{DEST}[63:0] \leftarrow \text{RoundToIntegerDP}(\text{SRC}[63:0], \text{Zero_upper_imm}[7:0])
\text{ELSE}
\quad \text{IF } \text{"merging-masking" ; merging-masking}
\quad \text{THEN } \text{DEST}[63:0] \text{ remains unchanged}*
\quad \text{ELSE} ; \text{zeroing-masking}
\quad \text{THEN } \text{DEST}[63:0] \leftarrow 0
\]

\[
\text{FI;}
\]

\[
\text{DEST}[127:64] \leftarrow \text{SRC}[127:64]
\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{VRNDSCALESD}_{\text{__m128d}} \text{__mm_roundscale_sd} (\text{__m128d} \text{a}, \text{__m128d} \text{b}, \text{int imm});
\text{VRNDSCALESD}_{\text{__m128d}} \text{__mm_roundscale_round_sd} (\text{__m128d} \text{a}, \text{__m128d} \text{b}, \text{int imm, int sae});
\text{VRNDSCALESD}_{\text{__m128d}} \text{__mm_mask_roundscale_sd} (\text{__m128d} \text{s}, \text{__mmask8} \text{k}, \text{__m128d} \text{a}, \text{__m128d} \text{b}, \text{int imm});
\text{VRNDSCALESD}_{\text{__m128d}} \text{__mm_mask_roundscale_round_sd} (\text{__m128d} \text{s}, \text{__mmask8} \text{k}, \text{__m128d} \text{a}, \text{__m128d} \text{b}, \text{int imm, int sae});
\text{VRNDSCALESD}_{\text{__m128d}} \text{__mm_maskz_roundscale_sd} (\text{__mmask8} \text{k}, \text{__m128d} \text{a}, \text{__m128d} \text{b}, \text{int imm});
\text{VRNDSCALESD}_{\text{__m128d}} \text{__mm_maskz_roundscale_round_sd} (\text{__mmask8} \text{k}, \text{__m128d} \text{a}, \text{__m128d} \text{b}, \text{int imm, int sae});
\]

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.
VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

### Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1’ then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}_\text{INT}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round}_\text{INT}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]
Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-16.

**Operation**

RoundToIntegerSP(SRC[31:0], imm8[7:0])

if (imm8[2] = 1)
    rounding_direction ← MXCSR:RC ; get round control from MXCSR
else
    rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
Fl
M ← imm8[7:4] ; get the scaling factor

case (rounding_direction)
00: TMP[31:0] ← round_to_nearest_even_integer(2^M*SRC[31:0])
01: TMP[31:0] ← round_to_equal_or_smaller_integer(2^M*SRC[31:0])
10: TMP[31:0] ← round_to_equal_or_larger_integer(2^M*SRC[31:0])
11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
ESAC;

Dest[31:0] ← 2^-M* TMP[31:0] ; scale down back to 2^-M
if (imm8[3] = 0) Then ; check SPE
    if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
        set_precision() ; set #PE
    Fl;
Fl;
return(Dest[31:0])

VRNDSCALEPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF *src is a memory operand*
    THEN TMP_SRC ← BROADCAST32(SRC, VL, k1)
    ELSE TMP_SRC ← SRC
Fl:

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← RoundToIntegerSP(TMP_SRC[i+31:i]), imm8[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
    Fl;
Fl;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VRNDSCALEPS __m512 __mm512_roundscale_ps( __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_roundscale_round_ps( __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_ps( __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_round_ps( __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 __mm256_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m256 __mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_maskz_roundscale_ps( __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 __mm128_roundscale_ps(__m128 a, int imm);
VRNDSCALEPS __m128 __mm128_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 __mm128_maskz_roundscale_ps( __mmask8 k, __m128 a, int imm);

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2.
VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 / r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>TUPLE1 SCALAR</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMrm/r (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

\[
ROUND(x) = 2^{-M}*\text{Round	extunderscore to	extunderscore INT}(x*2^M, \text{round	extunderscore ctrl}),
\]

\[
\text{round	extunderscore ctrl} = \text{imm}[3:0];
\]

\[
M=\text{imm}[7:4];
\]

The operation of \(x*2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

\[
ROUND(x) = \text{Round	extunderscore to	extunderscore INT}(x, \text{round	extunderscore ctrl}),
\]

\[
\text{round	extunderscore ctrl} = \text{imm}[3:0];
\]

**EVEX encoded version:** The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-16.
**Operation**

RoundToIntegerSP([SRC][31:0], imm8[7:0]) {
  if (imm8[2] = 1)
    rounding_direction := MXCSR:RC ; get round control from MXCSR
  else
    rounding_direction := imm8[1:0] ; get round control from imm8[1:0]
  FI
  M := imm8[7:4] ; get the scaling factor

  case (rounding_direction)
  00: TMP[31:0] := round_to_nearest_even_integer(2^M*SRC[31:0])
  01: TMP[31:0] := round_to_equal_or_smaller_integer(2^M*SRC[31:0])
  10: TMP[31:0] := round_to_equal_or_larger_integer(2^M*SRC[31:0])
  11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
  ESAC;

  Dest[31:0] := 2^-M * TMP[31:0] ; scale down back to 2^-M
  if (imm8[3] = 0) Then ; check SPE
    if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
      set_precision() ; set #PE
    FI;
  FI;
  return(Dest[31:0])
}

**VRNDSCALESS (EVEX encoded version)**

IF k1[0] or *no writemask*
  THEN DEST[31:0] := RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[31:0] remains unchanged*
  ELSE ; zeroing-masking
    THEN DEST[31:0] := 0
  FI;
  DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRNDSCALESS __m128_mm_roundscale_ss (__m128 a, __m128 b, int imm);
VRNDSCALESS __m128_mm_roundscale_round_ss (__m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128_mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128_mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128_mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.
VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $\infty$ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VRSQRT14PD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[63:0]));
    ELSE DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

Table 5-17. VRSQRT14PD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>X = 2⁻²ⁿ</td>
<td>2ⁿ</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>X = +0</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PD __m512d __m512_rsqrt14_pd(__m512d a);
VRSQRT14PD __m512d __m512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d __m512_maskz_rsqrt14_pd( __mmask8 k, __m512d a);
VRSQRT14PD __m256d __m256_rsqrt14_pd( __m256d a);
VRSQRT14PD __m256d __m256_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d __m256_maskz_rsqrt14_pd( __mmask8 k, __m256d a);
VRSQRT14PD __m128d __m128_rsqrt14_pd( __m128d a);
VRSQRT14PD __m128d __m128_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d __m128_maskz_rsqrt14_pd( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.
VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 4F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square root of the scalar double-precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Computes the approximate reciprocal of the square roots of the scalar double-precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an +$\infty$ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


Operation

VRSQRT14SD (EVEX version)

IF k1[0] or *no writemask*
    THEN DEST[63:0] $\leftarrow$ APPROXIMATE(1.0/ SQRT(SRC2[63:0]))
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged* 
    ELSE ; zeroing-masking
        THEN DEST[63:0] $\leftarrow$ 0
    FI;
FI;
DEST[127:64] $\leftarrow$ SRC1[127:64]
DEST[MAXVL-1:128] $\leftarrow$ 0
Table 5-18. VRSQRT14SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>$X = 2^{-2^n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14SD __m128d __mm_rsqrt14_sd(__m128d a, __m128d b);
VRSQRT14SD __m128d __mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d __mm_maskz_rsqrt14_sd(__mmask8d m, __m128d a, __m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E5.
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 4E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 4E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 4E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2⁻¹⁴.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ±∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VRSQRT14PS (EVEX encoded versions)**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 32\)

IF \(k1[j]\) OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN DEST\([i+31:i]\) \(\leftarrow\) APPROXIMATE\((1.0/\text{SQRT}(\text{SRC}[31:0]));\)

ELSE DEST\([i+31:i]\) \(\leftarrow\) APPROXIMATE\((1.0/\text{SQRT}(\text{SRC}[i+31:i]));\)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST\([i+31:i]\) remains unchanged*;

ELSE ; zeroing-masking

DEST\([i+31:i]\) \(\leftarrow 0\)

FI;

FI;

ENDFOR;

DEST\([\text{MAXVL}-1:VL]\) \(\leftarrow\) 0

### Table 5-19. VRSQRT14PS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>(X = 2^{-2n})</td>
<td>(2^n)</td>
<td></td>
</tr>
<tr>
<td>(X &lt; 0)</td>
<td>QNaN_Indefinite</td>
<td>Including (-\text{INF})</td>
</tr>
<tr>
<td>(X = -0)</td>
<td>(-\text{INF})</td>
<td></td>
</tr>
<tr>
<td>(X = +0)</td>
<td>(+\text{INF})</td>
<td></td>
</tr>
<tr>
<td>(X = +\text{INF})</td>
<td>(+0)</td>
<td></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14PS __m512 _mm512_rsqrt14_ps( __m512 a);

VRSQRT14PS __m512 _mm512_mask_rsqrt14_ps(__m512 s, __mmask16 k, __m512 a);

VRSQRT14PS __m512 _mm512_maskz_rsqrt14_ps( __mmask16 k, __m512 a);

VRSQRT14PS __m256 _mm256_rsqrt14_ps( __m256 a);

VRSQRT14PS __m256 _mm256_mask_rsqrt14_ps(__m256 s, __mmask8 k, __m256 a);

VRSQRT14PS __m256 _mm256_maskz_rsqrt14_ps( __mmask8 k, __m256 a);

VRSQRT14PS __m128 _mm_rsqrt14_ps( __m128 a);

VRSQRT14PS __m128 _mm_mask_rsqrt14_ps(__m128 s, __mmask8 k, __m128 a);

VRSQRT14PS __m128 _mm_maskz_rsqrt14_ps( __mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.
VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W0 4F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $\infty$, zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


#### Operation

**VRSQRT14SS (EVEX version)**

IF $k1[0]$ or *no writemask*
THEN DEST[31:0] ← APPROXIMATE(1.0/ SQRT(SRC2[31:0]))
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[31:0] remains unchanged*
  ELSE ; zeroing-masking
    THEN DEST[31:0] ← 0
  FI;
FI;
DEST[MAXVL-1:1:128] ← 0
Table 5-20. VRSQRT14SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SS __m128 _mm_rsqrt14_ss( __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_maskz_rsqrt14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.
VS\text{SCALEFPD}—Scale Packed Float64 Values With Float64 Values

**Opcode/Instruction**  | **Op / En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
---|---|---|---|---
EVEX.NDS.128:66.0F38.W1 2C /r  | A | V/V | AVX512VL AVX512F | Scale the packed double-precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.
EVEX.NDS.256:66.0F38.W1 2C /r  | A | V/V | AVX512VL AVX512F | Scale the packed double-precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.
EVEX.NDS.512:66.0F38.W1 2C /r  | A | V/V | AVX512F | Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

\[ zmm1 := zmm2 \times 2^{\text{floor}(zmm3)} \]

Floor(zmm3) means maximum integer value \( \leq zmm3 \).

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-21 and Table 5-22.

### Table 5-21. VS\text{SCALEFPD}/SD/PS/SS Special Cases

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Set IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>±NaN</td>
<td>±NaN(Src2)</td>
<td></td>
</tr>
<tr>
<td>±NaN</td>
<td>QNaN(Src1)</td>
<td>+INF</td>
</tr>
<tr>
<td>±NaN</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
</tr>
<tr>
<td>±Inf</td>
<td>QNaN(Src2)</td>
<td>Src1</td>
</tr>
<tr>
<td>±0</td>
<td>QNaN(Src2)</td>
<td>QNaN_Indefinite</td>
</tr>
<tr>
<td>Denorm/Norm</td>
<td>QNaN(Src2)</td>
<td>±INF (Src1 sign)</td>
</tr>
</tbody>
</table>
VSCALEFPD—Scale Packed Float64 Values With Float64 Values

INSTRUCTION SET REFERENCE, V-Z

Table 5-22. Additional VSCALEFPD/SD Special Cases

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>±0 or ±Min-Denormal (Src1 sign)</td>
<td>Underflow</td>
</tr>
<tr>
<td></td>
<td>±INF (Src1 sign) or ±Max-normal (Src1 sign)</td>
<td>Overflow</td>
</tr>
</tbody>
</table>

Operation

SCALE(SRC1, SRC2)
{
  TMP_SRC2 ← SRC2
  TMP_SRC1 ← SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 64 bits floating-point value */
  DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}

VSCALEFPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
      ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VSACLEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSACLEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSACLEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSACLEFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VSACLEFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSACLEFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VSACLEFPD __m256d _mm256_scalef_pd(__m256d a, __m256d b);
VSACLEFPD __m256d _mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VSACLEFPD __m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VSACLEFPD __m128d _mm_scalef_pd(__m128d a, __m128d b);
VSACLEFPD __m128d _mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VSACLEFPD __m128d _mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Exceptions Type E2.
VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

**Instruction Set Reference, V-Z**

VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}

**Description**
Performs a floating-point scale of the packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

\[
xmm1 := xmm2 \times 2^{\text{floor}(xmm3)}.
\]

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

**EVEX encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

**Handling of special-case input values are listed in Table 5-21 and Table 5-22.**

**Operation**

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2 = 0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1 = 0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```
VS CaLEFSD (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] OR *no writemask*
   THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[63:0] remains unchanged*
           ELSE ; zeroing-masking
               DEST[63:0] ← 0
       FI
   FI
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VS CaLEFSD __m128d __mm_scalef_round_sd(__m128d a, __m128d b, int);
VS CaLEFSD __m128d __mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VS CaLEFSD __m128d __mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Exceptions Type E3.
VS Ca lPacked Float32 Values With Float32 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 2C/r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 2C/r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 2C/r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>A/V/V</td>
<td>AVX512F</td>
<td>Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

\[ z_{mm} := z_{mm2} \times 2^{\text{floor}(z_{mm3})} \]

Floor(z_{mm3}) means maximum integer value \( \leq z_{mm3} \).

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

**Table 5-23. Additional VSCALEFPS/SS Special Cases**

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>result</td>
<td>&lt; 2^{-149} )</td>
</tr>
<tr>
<td>(</td>
<td>result</td>
<td>\geq 2^{128} )</td>
</tr>
</tbody>
</table>
Operation
SCALE(SRC1, SRC2)
{ ; Check for denormal operands
  TMP_SRC2 ← SRC2
  TMP_SRC1 ← SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 32 bits floating-point value */
  DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

VSCALEFPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);
      ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;

Intel C/C++ Compiler Intrinsic Equivalent
VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m512 _mm512_scalef_ps(__m512 a, __m512 b);
VSCALEFPS __m512 _mm512_mask_scalef_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m512 _mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
VSCALEFPS _m256 _mm256_scalef_ps(__m256 a, __m256 b);
VSCALEFPS _m256 _mm256_mask_scalef_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VSCALEFPS _m256 _mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b);
VSCALEFPS _m128 _mm_scalef_ps(__m128 a, __m128 b);
VSCALEFPS _m128 _mm_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSCALEFPS _m128 _mm_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.
Other Exceptions
See Exceptions Type E2.
VSACLEFSS—Scale Scalar Float32 Value With Float32 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W0 2D r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

\[ \text{xmm1} := \text{xmm2} \times 2^{\text{floor}(\text{xmm3})}. \]

Floor(xmm3) means maximum integer value \( \leq \text{xmm3} \).

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-21 and Table 5-23.
Operation
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * Pow(2, Floor(TMP_SRC2[31:0]))
}

VSCALEFSS (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] ← 0
        FI
    FI;
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Exceptions Type E3.
VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:
- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
• If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.
• If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

**Operation**
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
(30,125),(280,137)

**VSCATTERDPS (EVEX encoded versions)**
( KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] ← SRC[i+31:i]
    k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

**VSCATTERDPD (EVEX encoded versions)**
( KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ← SRC[i+63:i]
    k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
**VSCATTERQPS (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j * 32
k ← j * 64

IF k1[j] OR *no writemask*

THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ←

SRC[i+31:i]

k1[j] ← 0

Fi;

ENDFOR

k1[MAX_KL-1:KL] ← 0

**VSCATTERQPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] ←

SRC[i+63:i]

k1[j] ← 0

Fi;

ENDFOR

k1[MAX_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERPD void _mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERDPD void _mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void _mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERDPS void _mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void _mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void _mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void _mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERDPD void _mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void _mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void _mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void _mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void _mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void _mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
SIMD Floating-Point Exceptions
Invalid, Overflow, Underflow, Precision, Denormal

Other Exceptions
See Exceptions Type E12.
### VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-bit Granularity

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction_operand Encoding

<table>
<thead>
<tr>
<th>Op/En Tuples Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
</tr>
</tbody>
</table>

### Description

256-bit Version: Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.
**Operation**

Select2(SRC, control) {
CASE (control[0]) OF
0: TMP ← SRC[127:0];
1: TMP ← SRC[255:128];
ESAC;
RETURN TMP
}

Select4(SRC, control) {
CASE (control[1:0]) OF
0: TMP ← SRC[127:0];
1: TMP ← SRC[255:128];
2: TMP ← SRC[383:256];
3: TMP ← SRC[511:384];
ESAC;
RETURN TMP
}

**VSHUFF32x4 (EVEX versions)**

(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
i ← j * 32
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
FI;
ENDFOR;
IF VL = 256
TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
TMP_DEST[383:256] ← Select4(TMPSRC2[511:0], imm8[5:4]);
TMP_DEST[511:384] ← Select4(TMPSRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask*
THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
THEN DEST[i+31:i] ← 0
FI;
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
VSHUFF64x2 (EVEX 512-bit version)
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VSHUFI32x4 (EVEX 512-bit version)
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
ELSE
    IF *merging-masking*  ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*  ; zeroing-masking
            THEN DEST[i+31:i] ← 0
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0

VSHUF64x2 (EVEX 512-bit version)
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*  ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*  ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VSHUFI32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512i _mm512_shuffle_f32x4(__m512d a, __m512d b, int imm);
VSHUFF32x4 __m512i _mm512_mask_shuffle_f32x4(__m512d s, __mmask16 k, __m512d a, __m512d b, int imm);
VSHUFF32x4 __m512i _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_mask_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E4NF.

#UD If EVEX.L'L = 0 for VSHUFF32x4/VSHUFF64x2.
VTESTPD/VTESTPS—Packed Bit Test

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 0E /r VTESTPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0E /r VTESTPS ymm1, ymm2/m256</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 0F /r VTESTPD xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0F /r VTESTPD ymm1, ymm2/m256</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M reg field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Operation

**VTESTPS (128-bit version)**
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
   THEN ZF ← 1;
   ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
   THEN CF ← 1;
   ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;

**VTESTPS (VEX.256 encoded version)**
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
   THEN ZF ← 1;
   ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
   THEN CF ← 1;
   ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;

**VTESTPD (128-bit version)**
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF (TEMP[63] = TEMP[127] = 0)
   THEN ZF ← 1;
   ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[63] = TEMP[127] = 0)
   THEN CF ← 1;
   ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;

**VTESTPD (VEX.256 encoded version)**
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
   THEN ZF ← 1;
   ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
   THEN CF ← 1;
   ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
Intel C/C++ Compiler Intrinsic Equivalent

**VTESTPS**

```c
int _mm256_testz_ps (__m256 s1, __m256 s2);
int _mm256_testc_ps (__m256 s1, __m256 s2);
int _mm256_testnzc_ps (__m256 s1, __m128 s2);
int _mm_testz_ps (__m128 s1, __m128 s2);
int _mm_testc_ps (__m128 s1, __m128 s2);
int _mm_testnzc_ps (__m128 s1, __m128 s2);
```

**VTESTPD**

```c
int _mm256_testz_pd (__m256d s1, __m256d s2);
int _mm256_testc_pd (__m256d s1, __m256d s2);
int _mm256_testnzc_pd (__m256d s1, __m256d s2);
int _mm_testz_pd (__m128d s1, __m128d s2);
int _mm_testc_pd (__m128d s1, __m128d s2);
int _mm_testnzc_pd (__m128d s1, __m128d s2);
```

**Flags Affected**
The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Exceptions Type 4; additionally

- **#UD**
  - If VEX.vvvv ≠ 1111B.
  - If VEX.W = 1 for VTESTPS or VTESTPD.
VZEROALL—Zero All YMM Registers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.0F.WIG 77</td>
<td>ZO</td>
<td>V/V</td>
<td>AVX</td>
<td>Zero all YMM registers.</td>
</tr>
<tr>
<td>VZEROALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation**

`simd_reg_file[][]` is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512. On a machine supporting Intel AVX but not Intel AVX-512, the width is “MAXVL”.

**VZEROALL (VEX.256 encoded version)**

IF (64-bit mode)

`limit ← 15`

ELSE

`limit ← 7`

FOR i in 0 .. limit:

`simd_reg_file[i][MAXVL-1:0] ← 0`

**Intel C/C++ Compiler Intrinsic Equivalent**

`VZEROALL: _mm256_zeroall()`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 8.
VZEROUPPER—Zero Upper Bits of YMM Registers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.0F.WIG 77</td>
<td>ZO</td>
<td>V/V</td>
<td>AVX</td>
<td>Zero upper 128 bits of all YMM registers.</td>
</tr>
<tr>
<td>VZEROUPPER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation**

simd_reg_file[] is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512. On a machine supporting Intel AVX but not Intel AVX-512, the width is “MAXVL”.

VZEROUPPER

IF (64-bit mode)
    limit ← 15
ELSE
    limit ← 7
FOR i in 0 .. limit:
    simd_reg_file[i][MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VZEROUPPER: _mm256_zeroupper()

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 8.
**WAIT/FWAIT—Wait**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B</td>
<td>WAIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Check pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>9B</td>
<td>FWAIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Check pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction’s results. See the section titled “Floating-Point Exception Synchronization” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on using the WAIT/FWAIT instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

`CheckForPendingUnmaskedFloatingPointExceptions;`

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

- **#NM** If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
**Description**

Writes back all modified cache lines in the processor’s internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

**Operation**

WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)

**Intel C/C++ Compiler Intrinsic Equivalent**

WBINVD void _wbinvd(void);

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If the current privilege level is not 0.
#UD If the LOCK prefix is used.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 09</td>
<td>WBINVD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Write back and flush internal caches; initiate writing-back and flushing of external caches.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
WRFSBASE/WRGSBASE—Write FS/GS Segment Base

**Description**
Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field. The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

**Operation**
FS/GS segment base address ← SRC;

**Flags Affected**
None

**C/C++ Compiler Intrinsic Equivalent**
WRFSBASE:  void _writefsbase_u32( unsigned int );
WRFSBASE:  _writefsbase_u64( unsigned __int64 );
WRGSBASE:  void _writegsbase_u32( unsigned int );
WRGSBASE:  _writegsbase_u64( unsigned __int64 );

**Protected Mode Exceptions**
#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

**Real-Address Mode Exceptions**
#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**
#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**
#UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.
64-Bit Mode Exceptions

#UD  If the LOCK prefix is used.
     If CR4.FSGSBASE[bit 16] = 0.
     If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0

#GP(0)  If the source register contains a non-canonical address.
### WRMSR—Write to Model Specific Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 30</td>
<td>WRMSR</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Write the value in EDX:EAX to MSR specified by ECX.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 2, “Model-Specific Registers (MSRs)” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

### Operation

\[
\text{MSR}[\text{ECX}] \leftarrow \text{EDX:EAX};
\]

### Flags Affected

None.
Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the value in ECX specifies a reserved or unimplemented MSR address.
If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
If the source register contains a non-canonical address and ECX specifies one of the following
MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32 GS_BASE, IA32 KERNEL GS_BASE,
IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If the value in ECX specifies a reserved or unimplemented MSR address.
If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
If the source register contains a non-canonical address and ECX specifies one of the following
MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32 GS_BASE, IA32 KERNEL GS_BASE,
IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
WRPKRU—Write Data to User Page Key Register

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 EF</td>
<td>WRPKRU</td>
<td>ZO</td>
<td>V/V</td>
<td>OSPKE</td>
<td>Writes EAX into PKRU.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX and RAX are ignored.

Operation

IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
ELSE #GP(0);
FI;

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRPKRU: void _wrpkru(uint32_t);

Protected Mode Exceptions

#GP(0) If ECX ≠ 0.
    If EDX ≠ 0.

#UD If the LOCK prefix is used.
    If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
## XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 XACQUIRE</td>
<td>V/V</td>
<td>HLE</td>
<td>A hint used with an “XACQUIRE-enabled” instruction to start lock elision on the instruction memory operand address.</td>
</tr>
<tr>
<td>F3 XRELEASE</td>
<td>V/V</td>
<td>HLE</td>
<td>A hint used with an “XRELEASE-enabled” instruction to end lock elision on the instruction memory operand address.</td>
</tr>
</tbody>
</table>

### Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The “MOV mem, reg” (Opcode 88H/89H) and “MOV mem, imm” (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode C6.

### NOTES:
1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.
**Intel 64 and IA-32 Compatibility**

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to:

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to:

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

**Operation**

**XACQUIRE**

IF XACQUIRE-enabled instruction

THEN

IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN

HLE_NEST_COUNT++

IF (HLE_NEST_COUNT = 1) THEN

HLE_ACTIVE ← 1

IF 64-bit mode

THEN

restartRIP ← instruction pointer of the XACQUIRE-enabled instruction

ELSE

restartEIP ← instruction pointer of the XACQUIRE-enabled instruction

FI;

Enter HLE Execution (* record register state, start tracking memory state *)

FI; (* HLE_NEST_COUNT = 1 *)

IF ElisionBufferAvailable

THEN

Allocate elision buffer

Record address and data for forwarding and commit checking

Perform elision

ELSE

Perform lock acquire operation transactionally but without elision

FI;

ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)

GOTO HLE_ABORT_PROCESSING

FI;

ELSE

Treat instruction as non-XACQUIRE F2H prefixed legacy instruction

FI;
XRELEASE

IF XRELEASE-enabled instruction
  THEN
      IF (HLE_NEST_COUNT > 0)
          THEN
              HLE_NEST_COUNT--
              IF lock address matches in elision buffer THEN
                  IF lock satisfies address and value requirements THEN
                      Deallocate elision buffer
                      ELSE
                          GOTO HLE_ABORT_PROCESSING
                      FI;
                  ELSE
                      GOTO HLE_ABORT_PROCESSING
                  FI;
          FI;
      IF (HLE_NEST_COUNT = 0)
          THEN
              IF NoAllocatedElisionBuffer
                  THEN
                      Try to commit transactional execution
                      IF fail to commit transactional execution
                          THEN
                              GOTO HLE_ABORT_PROCESSING;
                          ELSE (* commit success *)
                              HLE_ACTIVE ← 0
                          FI;
                      ELSE
                          GOTO HLE_ABORT_PROCESSING
                      FI;
              FI; (* HLE_NEST_COUNT > 0 *)
      ELSE
          Treat instruction as non-XRELEASE F3H prefixed legacy instruction
      FI;

(* For any HLE abort condition encountered during HLE execution *)

HLE_ABORT_PROCESSING:
  HLE_ACTIVE ← 0
  HLE_NEST_COUNT ← 0
  Restore architectural register state
  Discard memory updates performed in transaction
  Free any allocated lock elision buffers
  IF 64-bit mode
      THEN
          RIP ← restartRIP
      ELSE
          EIP ← restartEIP
  FI;
  Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
SIMD Floating-Point Exceptions
None

Other Exceptions
#GP(0) If the use of prefix causes instruction length to exceed 15 bytes.
**XABORT — Transactional Abort**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6 F8 ib XABORT imm8</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Causes an RTM abort if in RTM execution</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

**Operation**

**XABORT**

IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;

(* For any RTM abort condition encountered during RTM execution *)

RTM_ABORT_PROCESSING:

    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
            ELSE
            EIP ← fallbackEIP;
        FI;
    END

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

XABORT:     void _xabort( unsigned int);

**SIMD Floating-Point Exceptions**

None
Other Exceptions

#UD

CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
If LOCK prefix is used.
XADD—Exchange and Add

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C0 /r</td>
<td>XADD r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r8 and r/m8; load sum into r/m8.</td>
</tr>
<tr>
<td>REX + 0F C0 /r</td>
<td>XADD r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange r8 and r/m8; load sum into r/m8.</td>
</tr>
<tr>
<td>0F C1 /r</td>
<td>XADD r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 and r/m16; load sum into r/m16.</td>
</tr>
<tr>
<td>0F C1 /r</td>
<td>XADD r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r32 and r/m32; load sum into r/m32.</td>
</tr>
<tr>
<td>REX.W + 0F C1 /r</td>
<td>XADD r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange r64 and r/m64; load sum into r/m64.</td>
</tr>
</tbody>
</table>

NOTES: * In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>_operand 1</th>
<th>_operand 2</th>
<th>_operand 3</th>
<th>_operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRMr/m (r, w)</td>
<td>ModRM:reg (r, w)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

Protected Mode Exceptions

#GP(0)  If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD  If the LOCK prefix is used but the destination is not a memory operand.
Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
**XBEGIN — Transactional Begin**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C7 F8 XBEGIN rel16</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.</td>
</tr>
<tr>
<td>C7 F8 XBEGIN rel32</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Offset</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

**Operation**

XBEGIN

IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
  THEN
    RTM_NEST_COUNT++
    IF RTM_NEST_COUNT = 1 THEN
      IF 64-bit Mode
        THEN
          fallbackRIP ← RIP + SignExtend64(IMM)
          (* RIP is instruction following XBEGIN instruction *)
        ELSE
          fallbackEIP ← EIP + SignExtend32(IMM)
          (* EIP is instruction following XBEGIN instruction *)
        FI;
      IF (64-bit mode)
        THEN IF (fallbackRIP is not canonical)
          THEN #GP(0)
          FI;
        ELSE IF (fallbackEIP outside code segment limit)
          THEN #GP(0)
          FI;
        FI;
    FI;
    RTM_ACTIVE ← 1
    Enter RTM Execution (* record register state, start tracking memory state*)
  FI; (* RTM_NEST_COUNT = 1 * )
ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
    GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
        FI;
    END

Flags Affected
None

Intel C/C++ Compiler Intrinsic Equivalent
XBEGIN: unsigned int _xbegin( void );

SIMD Floating-Point Exceptions
None

Protected Mode Exceptions
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
    If LOCK prefix is used.
#GP(0) If the fallback address is outside the CS segment.

Real-Address Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
    If LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
    If LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-bit Mode Exceptions

#UD    CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
       If LOCK prefix is used.
#GP(0) If the fallback address is non-canonical.
XCHG—Exchange Register/Memory with Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>90+rw</td>
<td>XCHG AX, r16</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 with AX.</td>
</tr>
<tr>
<td>90+rw</td>
<td>XCHG AX, AX</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange AX with r16.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG EAX, r32</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r32 with EAX.</td>
</tr>
<tr>
<td>REX.W + 90+rd</td>
<td>XCHG RAX, r64</td>
<td>0</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange r64 with RAX.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG r32, EAX</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange EAX with r32.</td>
</tr>
<tr>
<td>REX.W + 90+rd</td>
<td>XCHG r64, RAX</td>
<td>0</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange RAX with r64.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>86 /r</td>
<td>XCHG r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r8 (byte register) with byte from r/m8.</td>
</tr>
<tr>
<td>REX + 86 /r</td>
<td>XCHG r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange r8 (byte register) with byte from r/m8.</td>
</tr>
<tr>
<td>86 /r</td>
<td>XCHG r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange byte from r/m8 with r8 (byte register).</td>
</tr>
<tr>
<td>REX + 86 /r</td>
<td>XCHG r8*, r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange byte from r/m8 with r8 (byte register).</td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 with word from r/m16.</td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange word from r/m16 with r16.</td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r32 with doubleword from r/m32.</td>
</tr>
<tr>
<td>REX.W + 87 /r</td>
<td>XCHG r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange r64 with quadword from r/m64.</td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange doubleword from r/m32 with r32.</td>
</tr>
<tr>
<td>REX.W + 87 /r</td>
<td>XCHG r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange quadword from r/m64 with r64.</td>
</tr>
</tbody>
</table>

**NOTES:**
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AX/EAX/RAX (r, w)</td>
<td>opcode + rd (r, w)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (r, w)</td>
<td>AX/EAX/RAX (r, w)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor’s locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See “Bus Locking” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.
NOTE

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation

TEMP ← DEST;
DEST ← SRC;
SRC ← TEMP;

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If either operand is in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
**XEND — Transactional End**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D5 XEND</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the end of an RTM code region.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

### Operation

**XEND**

IF (RTM_ACTIVE = 0) THEN
   SIGNAL #GP
ELSE
   RTM_NEST_COUNT--
   IF (RTM_NEST_COUNT = 0) THEN
      Try to commit transaction
      IF fail to commit transactional execution
         THEN
            GOTO RTM_ABORT_PROCESSING;
         ELSE (* commit success *)
            RTM_ACTIVE ← 0
      FI;
   FI;
FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)

RTM_ABORT_PROCESSING:
   Restore architectural register state
   Discard memory updates performed in transaction
   Update EAX with status
   RTM_NEST_COUNT ← 0
   RTM_ACTIVE ← 0
   IF 64-bit Mode
      THEN
         RIP ← fallbackRIP
      ELSE
         EIP ← fallbackEIP
   FI;
END
**Flags Affected**
None

**Intel C/C++ Compiler Intrinsic Equivalent**

XEND:   void _xend( void );

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**

#UD    CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
If LOCK or 66H or F2H or F3H prefix is used.

#GP(0)  If RTM_ACTIVE = 0.
XGETBV—Get Value of Extended Control Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D0</td>
<td>XGETBV</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Reads an XCR specified by ECX into EDX:EAX.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, ”Managing State Using the XSAVE Feature Set,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Use of any other value for ECX results in a general-protection (#GP) exception.

**Operation**

EDX:EAX ← XCR[ECX];

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XGETBV: unsigned __int64 _xgetbv( unsigned int);

**Protected Mode Exceptions**

- #GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**
Same exceptions as in protected mode.
**XLAT/XLATB—Table Look-up Translation**

**Description**
Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a "short form" of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**
IF AddressSize  =  16
  THEN
    AL  ← (DS:BX + ZeroExtend(AL));
    ELSE IF (AddressSize  =  32)
      AL  ← (DS:EBX + ZeroExtend(AL)); Fl;
      ELSE (AddressSize  =  64)
        AL  ← (RBX + ZeroExtend(AL));
    FL;

**Flags Affected**
None.

**Protected Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.
XOR—Logical Exclusive OR

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34 ib</td>
<td>XOR AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AL XOR imm8.</td>
</tr>
<tr>
<td>35 iw</td>
<td>XOR AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AX XOR imm16.</td>
</tr>
<tr>
<td>35 id</td>
<td>XOR EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>EAX XOR imm32.</td>
</tr>
<tr>
<td>REX.W + 35 id</td>
<td>XOR RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>RAX XOR imm32 (sign-extended).</td>
</tr>
<tr>
<td>80 /6 ib</td>
<td>XOR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 XOR imm8.</td>
</tr>
<tr>
<td>REX + 80 /6 ib</td>
<td>XOR r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 XOR imm8.</td>
</tr>
<tr>
<td>81 /6 iw</td>
<td>XOR r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR imm16.</td>
</tr>
<tr>
<td>81 /6 id</td>
<td>XOR r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR imm32.</td>
</tr>
<tr>
<td>REX.W + 81 /6 id</td>
<td>XOR r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR imm32 (sign-extended).</td>
</tr>
<tr>
<td>83 /6 ib</td>
<td>XOR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>83 /6 id</td>
<td>XOR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>REX.W + 83 /6 ib</td>
<td>XOR r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>30 /r</td>
<td>XOR r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 XOR r8.</td>
</tr>
<tr>
<td>REX + 30 /r</td>
<td>XOR r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 XOR r8.</td>
</tr>
<tr>
<td>31 /r</td>
<td>XOR r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR r16.</td>
</tr>
<tr>
<td>31 /r</td>
<td>XOR r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR r32.</td>
</tr>
<tr>
<td>REX.W + 31 /r</td>
<td>XOR r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR r64.</td>
</tr>
<tr>
<td>32 /r</td>
<td>XOR r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r8 XOR r/m8.</td>
</tr>
<tr>
<td>REX + 32 /r</td>
<td>XOR r8*, r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r8 XOR r/m8.</td>
</tr>
<tr>
<td>33 /r</td>
<td>XOR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 XOR r/m16.</td>
</tr>
<tr>
<td>33 /r</td>
<td>XOR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r32 XOR r/m32.</td>
</tr>
<tr>
<td>REX.W + 33 /r</td>
<td>XOR r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r64 XOR r/m64.</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m(r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m(r, w)</td>
<td>ModRMreg(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg(r, w)</td>
<td>ModRMr/m(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Perform a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.
In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[ \text{DEST} \leftarrow \text{DEST} \text{ XOR SRC}; \]

**Flags Affected**

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

**Protected Mode Exceptions**

- #GP(0) If the destination operand points to a non-writable segment.
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
### XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 57/r XORPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F.WT 57 /r VXORPD xmm1 [k1]{[z],} xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F.WT 57 /r VXORPD ymm1 [k1]{[z],} ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F.WT 57 /r VXORPD zmm1 [k1]{[z],} zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Performs a bitwise logical XOR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

**EVEX.512 encoded version:** The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

**VEX.256 and EVEX.256 encoded versions:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 and EVEX.128 encoded versions:** The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

**VXORPD (EVEX encoded versions)**

\(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 64\)

IF \(k1[j]\) OR *no writemask* THEN

IF \((EVEX.b == 1)\) AND (SRC2 *is memory*)

THEN \(\text{DEST}[i+63:i] \leftarrow \text{SRC1}[i+63:i] \text{ BITWISE XOR SRC2}[63:0]\);

ELSE \(\text{DEST}[i+63:i] \leftarrow \text{SRC1}[i+63:i] \text{ BITWISE XOR SRC2}[i+63:i]\);

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *\(\text{DEST}[i+63:i]\) remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\(\text{DEST}[i+63:i] = 0\)

FI

ENDIF

\(\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0\)

**VXORPD (VEX.256 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] \text{ BITWISE XOR SRC2}[63:0]\)

\(\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] \text{ BITWISE XOR SRC2}[127:64]\)

\(\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] \text{ BITWISE XOR SRC2}[191:128]\)

\(\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] \text{ BITWISE XOR SRC2}[255:192]\)

\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0\)

**VXORPD (VEX.128 encoded version)**

\(\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] \text{ BITWISE XOR SRC2}[63:0]\)

\(\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] \text{ BITWISE XOR SRC2}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0\)

**XORPD (128-bit Legacy SSE version)**

\(\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] \text{ BITWISE XOR SRC}[63:0]\)

\(\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] \text{ BITWISE XOR SRC}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] \leftarrow \text{DEST}[\text{MAXVL}-1:128]\) (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

\(\text{VXORPD \_m512d \_mm512\_xor\_pd (\_m512d a, \_m512d b);}\)

\(\text{VXORPD \_m512d \_mm512\_mask\_xor\_pd (\_m512d a, \_mm\_mask8 m, \_m512d b);}\)

\(\text{VXORPD \_m512d \_mm512\_maskz\_xor\_pd (\_mm\_mask8 m, \_m512d a);}\)

\(\text{VXORPD \_m256d \_mm256\_xor\_pd (\_m256d a, \_m256d b);}\)

\(\text{VXORPD \_m256d \_mm256\_mask\_xor\_pd (\_m256d a, \_mm\_mask8 m, \_m256d b);}\)

\(\text{VXORPD \_m256d \_mm256\_maskz\_xor\_pd (\_mm\_mask8 m, \_m256d a);}\)

\(\text{XORPD \_m128d \_mm\_xor\_pd (\_m128d a, \_m128d b);}\)

\(\text{VXORPD \_m128d \_mm\_mask\_xor\_pd (\_m128d a, \_mm\_mask8 m, \_m128d b);}\)

\(\text{VXORPD \_m128d \_mm\_maskz\_xor\_pd (\_mm\_mask8 m, \_m128d a);}\)

**SIMD Floating-Point Exceptions**

None
Other Exceptions
Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4.
XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 57 /r XORPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 57 /r VXORPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 57 /r VXORPS xmm1[k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 57 /r VXORPS ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 57 /r VXORPS zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

VXORPS (EVEX encoded versions)

KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0];
      ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] = 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] ← 0

VXORPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:256] ← 0

VXORPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:128] ← 0

XORPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VXORPS __m512 __mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 __mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 __mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 __mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 __mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 __mm256_maskz_xor_ps (__mmask8 m, __m256 a);
VXORPS __m128 __mm128_xor_ps (__m128 a, __m128 b);
VXORPS __m128 __mm128_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
VXORPS __m128 __mm_maskz_xor_ps (__mmask8 m, __m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4.
XRSTOR—Restore Processor Extended States

### Opcode / Instruction

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /5 XRSTOR mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVE</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /5 XRSTOR64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVE</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.8, “Operation of XRSTOR,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[i] = 0, XRSTOR does not update state component i.\(^1\)
- If RFBM[i] = 1 and bit i is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component i.
- If RFBM[i] = 1 and XSTATE_BV[i] = 1, XRSTOR loads state component i from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component i for which RFBM[i] = 1 and XSTATE_BV[i] = 1; it tracks as modified any state component i for which RFBM[i] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

---

1. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.
Operation

RFBM ← XCR0 AND EDX:EAX; /* bitwise logical AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED ← RFBM AND RSTORMASK;
        TO_BE_INITIALIZED ← RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                load x87 state from legacy region of XSAVE area;
                XINUSE[0] ← 1;
            ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                initialize x87 state;
                XINUSE[0] ← 0;
            FI;

            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
                XINUSE[1] ← 1;
            ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0; // this step does not initialize MXCSR
                XINUSE[1] ← 0;
            FI;

        FOR i ← 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset n from base of XSAVE area;
                    // n enumerated by CPUID(EAX=0DH,ECX=i):EBX)
                    XINUSE[i] ← 1;
                ELSIF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
                FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1]= 0
            THEN /* compacted form not supported */
                #GP(0);
        FI;

XRSTOR—Restore Processor Extended States
FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1 THEN
load x87 state from legacy region of XSAVE area;
XINUSE[0] ← 1;
ELSIF TO_BE_INITIALIZED[0] = 1 THEN
initialize x87 state;
XINUSE[0] ← 0;
FI;

IF TO_BE_RESTORED[1] = 1 THEN
load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
XINUSE[1] ← 1;
ELSIF TO_BE_INITIALIZED[1] = 1 THEN
set all XMM registers to 0;
MXCSR ← 1F80H;
XINUSE[1] ← 0;
FI;

NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
IF FORMAT[i] = 1 THEN
IF TO_BE_RESTORED[i] = 1 THEN
load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
XINUSE[i] ← 1;
FI;
NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=ODH,ECX=i);EAX);
FI;
IF TO_BE_INITIALIZED[i] = 1 THEN
initialize XSAVE state component i;
XINUSE[i] ← 0;
ENDFOR;
FI;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation THEN VMXNR ← 1;
ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← (CPL, VMXNR, LAXA, COMPMMASK);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTOR: void _xrstor( void *, unsigned __int64);
XRSTOR: void _xrstor64( void *, unsigned __int64);

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and
CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XSTATE_BV field of the XSAVE header is 1.
If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XCOMP_BV field of the XSAVE header is 1.
If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0
and the corresponding bit in the XSTATE_BV field is 1.
If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
operand is not aligned on a 64-byte boundary, as described above. If the alignment check
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and
CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XSTATE_BV field of the XSAVE header is 1.
If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XCOMP_BV field of the XSAVE header is 1.
If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0) If a memory address is in a non-canonical form.

If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.

If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.

If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.

If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XRSTORS—Restore Processor Extended States Supervisor

**Opcode / Instruction**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C7 /3</td>
<td>XRSTORS mem</td>
<td>M</td>
<td>V/V</td>
<td>XSS</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /3</td>
<td>XRSTORS64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSS</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.12, “Operation of XRSTORS,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, “Extended Region of an XSAVE Area”).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[i] = 0, XRSTORS does not update state component i.
- If RFBM[i] = 1 and bit i is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component i.
- If RFBM[i] = 1 and XSTATE_BV[i] = 1, XRSTORS loads state component i from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component i for which RFBM[i] = 1 and XSTATE_BV[i] = 1; it tracks as modified any state component i for which RFBM[i] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.
**Operation**

\[
\begin{align*}
\text{RFBM} & \leftarrow (\text{XCR0 OR IA32_XSS}) \text{ AND EDX:EAX}; \\
& \quad /\ast \text{ bitwise logical OR and AND } \ast/ \\
\text{COMPMASK} & \leftarrow \text{XCOMP_BV field from XSAVE header}; \\
\text{RSTORMASK} & \leftarrow \text{XSTATE_BV field from XSAVE header}; \\
\text{FORMAT} = \text{COMPMASK AND } 7FFFFFFF_FFFFFFFFH; \\
\text{RESTORE_FEATURES} = \text{FORMAT AND RFBM}; \\
\text{TO_BE_RESTORED} & \leftarrow \text{RESTORE_FEATURES AND RSTORMASK}; \\
\text{FORCE_INIT} & \leftarrow \text{RFBM AND NOT FORMAT}; \\
\text{TO_BE_INITIALIZED} & = \left(\text{RFBM AND NOT RSTORMASK}\right) \text{ OR } \text{FORCE_INIT}; \\
\end{align*}
\]

\[
\begin{align*}
\text{IF } \text{TO_BE_RESTORED}[0] = 1 & \quad \text{THEN} \\
& \quad \text{load x87 state from legacy region of XSAVE area;} \\
& \quad \text{XINUSE}[0] \leftarrow 1; \\
\text{ELSIF } \text{TO_BE_INITIALIZED}[0] = 1 & \quad \text{THEN} \\
& \quad \text{initialize x87 state;} \\
& \quad \text{XINUSE}[0] \leftarrow 0; \\
& \quad \text{FI}; \\
\end{align*}
\]

\[
\begin{align*}
\text{IF } \text{TO_BE_RESTORED}[1] = 1 & \quad \text{THEN} \\
& \quad \text{load SSE state from legacy region of XSAVE area; } // \text{ this step loads the XMM registers and MXCSR} \\
& \quad \text{XINUSE}[1] \leftarrow 1; \\
\text{ELSIF } \text{TO_BE_INITIALIZED}[1] = 1 & \quad \text{THEN} \\
& \quad \text{set all XMM registers to 0;} \\
& \quad \text{MXCSR} \leftarrow 1F80H; \\
& \quad \text{XINUSE}[1] \leftarrow 0; \\
& \quad \text{FI}; \\
\end{align*}
\]

\[
\begin{align*}
\text{NEXT_FEATURE_OFFSET} = 576; & \quad // \text{ Legacy area and XSAVE header consume 576 bytes} \\
\text{FOR } i \leftarrow 2 \text{ TO } 62 & \quad \text{IF } \text{FORMAT}[i] = 1 \\
& \quad \text{THEN} \\
& \quad \quad \text{IF } \text{TO_BE_RESTORED}[i] = 1 \\
& \quad \quad \quad \text{THEN} \\
& \quad \quad \quad \quad \text{load XSAVE state component } i \text{ at offset NEXT_FEATURE_OFFSET from base of XSAVE area;} \\
& \quad \quad \quad \quad \text{XINUSE}[i] \leftarrow 1; \\
& \quad \quad \quad \quad \text{FI}; \\
& \quad \quad \quad \quad \text{NEXT_FEATURE_OFFSET} = \text{NEXT_FEATURE_OFFSET} + n (n \text{ enumerated by CPUID(EAX=0DH,ECX=i):EAX}); \\
& \quad \quad \text{FI}; \\
& \quad \quad \text{IF } \text{TO_BE_INITIALIZED}[i] = 1 \\
& \quad \quad \quad \text{THEN} \\
& \quad \quad \quad \text{initialize XSAVE state component } i; \\
& \quad \quad \quad \text{XINUSE}[i] \leftarrow 0; \\
& \quad \quad \text{FI}; \\
\text{ENDFOR}; \\
\end{align*}
\]

\[
\begin{align*}
\text{XMODIFIED_BV} & \leftarrow \text{NOT RFBM}; \\
\text{IF in VMX non-root operation} \\
\end{align*}
\]
THEN VMXRN ← 1;
ELSE VMXRN ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← (CPL, VMXRN, LAXA, COMPMASK);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTORS: void _xrstors( void *, unsigned __int64);
XRSTORS64: void _xrstors64( void *, unsigned __int64);

Protected Mode Exceptions
#GP(0) If CPL > 0.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
   If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
   If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
   If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the
   XSTATE_BV field is 1.
   If bytes 63:16 of the XSAVE header are not all zero.
   If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If the LOCK prefix is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
   operand is not aligned on a 64-byte boundary, as described above. If the alignment check
   exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
   vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP
   is signaled in its place. In addition, the width of the alignment check may also vary with imple-
   mentation. For instance, for a given implementation, an alignment check exception might be
   signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalign-
   ments (4-, 8-, or 16-byte misalignments).
Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space, from 0 to FFFFH.
If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
If bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

 Same exceptions as in protected mode.

Compatibility Mode Exceptions

 Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If CPL > 0.
If a memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
If bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSAVE—Save Processor Extended States

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /4 XSAVE mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVE</td>
<td>Save state components specified by EDX:EAX to mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /4 XSAVE64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVE</td>
<td>Save state components specified by EDX:EAX to mem.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRMr/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.7, "Operation of XSAVE," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component \( i \) if and only if \( \text{RFBM}[i] = 1 \).
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If \( \text{RFBM}[i] = 1 \), XSAVE writes \( \text{XSTATE_BV}[i] \) with the value of \( \text{XINUSE}[i] \). ( \( \text{XINUSE} \) is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") If \( \text{RFBM}[i] = 0 \), XSAVE writes \( \text{XSTATE_BV}[i] \) with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

\[
\begin{align*}
\text{RFBM} & \leftarrow \text{XCR0 AND EDX:EAX}; \quad /* \text{bitwise logical AND */} \\
\text{OLD_BV} & \leftarrow \text{XSTATE_BV} \text{ field from XSAVE header;} \\
\text{IF } \text{RFBM}[0] = 1 & \quad \text{THEN store x87 state into legacy region of XSAVE area;} \\
& \quad \text{FI;} \\
\text{IF } \text{RFBM}[1] = 1 & \quad \text{THEN store MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.}
\end{align*}
\]

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.
THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
IF RFBM[i] = 1
THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i);EBX);
FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XSAVE: void _xsave( void *, unsigned __int64);
XSAVE: void _xsave64( void *, unsigned __int64);

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#NM** If CR0.TS[bit 3] = 1.
- **#UD** If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  If CR4.OSXSAVE[bit 18] = 0.
  If the LOCK prefix is used.
- **#AC** If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

**Real-Address Mode Exceptions**

- **#GP** If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
  If any part of the operand lies outside the effective address space from 0 to FFFFH.
- **#NM** If CR0.TS[bit 3] = 1.
- **#UD** If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  If CR4.OSXSAVE[bit 18] = 0.
  If the LOCK prefix is used.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
     If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
     If CR4.OSXSAVE[bit 18] = 0.
     If the LOCK prefix is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
     operand is not aligned on a 64-byte boundary, as described above. If the alignment check
     exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
     vary with implementation, as follows. In all implementations where #AC is not signaled, a
     general protection exception is signaled in its place. In addition, the width of the alignment
     check may also vary with implementation. For instance, for a given implementation, an align-
     ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
     tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
     misalignments).
XSAVEC—Save Processor Extended States with Compaction

**Description**

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.10, "Operation of XSAVEC," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component \(i\) if and only if \(RFBM[i] = 1\) and \(XINUSE[i] = 1\).1 (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.")
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.2,3 (See Section 13.4.2, "XSAVE Header.") XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

**Operation**

\[
\begin{align*}
\text{RFBM} & \leftarrow \text{XCR0 AND EDX:EAX}; \quad /*\text{bitwise logical AND}*/ \\
\text{TO\_BE\_SAVED} & \leftarrow \text{RFBM AND XINUSE}; \quad /*\text{bitwise logical AND}*/ \\
\text{If MXCSR} & \neq 1F80H \text{AND RFBM}[1]
\end{align*}
\]

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.
TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
    ENDIF;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← RFBM OR 80000000_00000000H;

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XSAVEC: void _xsavec( void *, unsigned __int64);
XSAVEC64: void _xsavec64( void *, unsigned __int64);

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
    If CR4.OSXSAVE[bit 18] = 0.
    If the LOCK prefix is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
Real-Address Mode Exceptions

#GP  If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
     If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
     If CR4.OSXSAVE[bit 18] = 0.
     If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If the memory address is in a non-canonical form.
        If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
     If CR4.OSXSAVE[bit 18] = 0.
     If the LOCK prefix is used.
#AC  If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSAVEOPT—Save Processor Extended States Optimized

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /6 XSAVEOPT mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEOPT</td>
<td>Save state components specified by EDX:EAX to mem, optimizing if possible.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /6 XSAVEOPT64 mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEOPT</td>
<td>Save state components specified by EDX:EAX to mem, optimizing if possible.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.9, "Operation of XSAVEOPT,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

• Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
• XSAVEOPT saves state component \( i \) only if \( \text{RFBM}[i] = 1 \) and \( \text{XINUSE}[i] = 1 \). (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVEOPT may optimize and not save state component \( i \) if (1) state component \( i \) has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVE corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
• XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area”).
• XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If \( \text{RFBM}[i] = 1 \), XSAVEOPT writes XSTATE_BV\( [i] \) with the value of XINUSE\( [i] \). If \( \text{RFBM}[i] = 0 \), XSAVEOPT writes XSTATE_BV\( [i] \) with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
• XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

\[
\text{RFBM} \leftarrow \text{XCR0 AND EDX:EAX}; \quad /\ast \text{bitwise logical AND} /\ast
\]

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if \( \text{RFBM}[1] = 1 \) or \( \text{RFBM}[2] = 1 \), regardless of the value of XINUSE.
OLD_BV ← XSTATE_BV field from XSAVE header;
TO_BE_SAVED ← RFBM AND XINUSE;

IF in VMX non-root operation
  THEN VMXNR ← 1;
  ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
IF XRSTOR_INFO = \langle CPL, VMXNR, LAXA, 00000000_00000000H \rangle
  THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
  THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
  THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
  THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
  IF TO_BE_SAVED[i] = 1
    THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
  FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);

**Flags Affected**
None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XSAVEOPT: void _xsaveopt( void * , unsigned __int64);
XSAVEOPT: void _xsaveopt64( void * , unsigned __int64);

**Protected Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP  If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
     If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
     If CR4.OSXSAVE[bit 18] = 0.
     If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
        If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)  If a page fault occurs.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
     If CR4.OSXSAVE[bit 18] = 0.
     If the LOCK prefix is used.
XSAVES—Save Processor Extended States Supervisor

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C7 /5 XSAVES mem</td>
<td>M</td>
<td>V/V</td>
<td>XSS</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /5 XSAVES64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSS</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.11, "Operation of XSAVES," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component i only if RFBM[i] = 1 and XINUSE[i] = 1.¹ (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVES may optimize and not save state component i if (1) state component i has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.² (See Section 13.4.2, "XSAVE Header.") XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

¹. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
². There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.
**Operation**

\[ \text{RFBM} \leftarrow (XCR0 \text{ OR IA32}_{-}\text{XSS}) \text{ AND EDX:EAX}; \quad /* \text{bitwise logical OR and AND} */ \]

IF in VMX non-root operation
   THEN \( \text{VMXNR} \leftarrow 1; \)
   ELSE \( \text{VMXNR} \leftarrow 0; \)
FI;

\( \text{LAXA} \leftarrow \text{linear address of XSAVE area}; \)
\( \text{COMPMASK} \leftarrow \text{RFBM OR 80000000_00000000H}; \)
\( \text{TO\_BE\_SAVED} \leftarrow \text{RFBM AND XINUSE}; \)
IF \( \text{XRSTOR\_INFO} = (\text{CPL}, \text{VMXNR}, \text{LAXA}, \text{COMPMASK}) \)
   THEN \( \text{TO\_BE\_SAVED} \leftarrow \text{TO\_BE\_SAVED} \text{ AND XMODIFIED}; \)
FI;

If \( \text{MXCSR} \neq 1F80H \text{ AND RFBM}[1] \)
   \( \text{TO\_BE\_SAVED}[1] = 1; \)
FI;

IF \( \text{TO\_BE\_SAVED}[0] = 1 \)
   THEN store x87 state into legacy region of XSAVE area;
FI;

IF \( \text{TO\_BE\_SAVED}[1] = 1 \)
   THEN store SSE state into legacy region of XSAVE area; \( // \) this step saves the XMM registers, MXCSR, and MXCSR\_MASK
FI;

\[ \text{NEXT\_FEATURE\_OFFSET} = 576; \quad // \text{Legacy area and XSAVE header consume 576 bytes} \]

FOR \( i \leftarrow 2 \) TO 62
  IF \( \text{RFBM}[i] = 1 \)
    THEN
      IF \( \text{TO\_BE\_SAVED}[i] \)
        THEN
          save XSAVE state component \( i \) at offset \( \text{NEXT\_FEATURE\_OFFSET} \) from base of XSAVE area;
          \( \text{IF} \quad i = 8 \quad // \text{state component } 8 \text{ is for PT state} \)
          \( \text{THEN } \text{IA32\_RTIT\_CTL\_TraceEn}[\text{bit } 0] \leftarrow 0; \)
          \( \text{FI}; \)
        FI;
      \( \text{NEXT\_FEATURE\_OFFSET} = \text{NEXT\_FEATURE\_OFFSET} + n \quad (n \ \text{enumerated by CPUID}(\text{EAX}=0DH, \text{ECX}=i):\text{EAX}); \)
    FI;
  ENDFOR;

\( \text{XSTATE\_BV} \text{ field in XSAVE header} \leftarrow \text{TO\_BE\_SAVED}; \)
\( \text{XCOMP\_BV} \text{ field in XSAVE header} \leftarrow \text{COMPMASK}; \)

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

\( \text{XSAVES:} \quad \text{void } \_\text{xsaves}( \text{void } *, \text{unsigned } \_\text{int64}); \)
\( \text{XSAVES64:} \quad \text{void } \_\text{xsaves64}( \text{void } *, \text{unsigned } \_\text{int64}); \)
Protected Mode Exceptions

#GP(0)  If CPL > 0.
     If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
     If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
     If CR4.OSXSAVE[bit 18] = 0.
     If the LOCK prefix is used.
#AC  If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP  If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
     If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
     If CR4.OSXSAVE[bit 18] = 0.
     If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0)  If CPL > 0.
      If the memory address is in a non-canonical form.
      If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code)  If a page fault occurs.

#NM  If CR0.TS[bit 3] = 1.

#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
      If CR4.OSXSAVE[bit 18] = 0.
      If the LOCK prefix is used.

#AC  If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSETBV—Set Extended Control Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D1</td>
<td>XSETBV</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Write the value in EDX:EAX to the XCR specified by ECX.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, "Enabling the XSAVE Feature Set and XSAVE-Enabled Features," of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

**Operation**

\[
\text{XCR[ECX] \leftarrow EDX:EAX;}
\]

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XSETBV: void _xsetbv( unsigned int, unsigned __int64);

**Protected Mode Exceptions**

#GP(0)  
If the current privilege level is not 0.  
If an invalid XCR is specified in ECX.  
If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.  
If an attempt is made to clear bit 0 of XCR0.  
If an attempt is made to set XCR0[2:1] to 10b.

#UD  
If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
If CR4.OSXSAVE[bit 18] = 0.  
If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP
- If an invalid XCR is specified in ECX.
- If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
- If an attempt is made to clear bit 0 of XCR0.
- If an attempt is made to set XCR0[2:1] to 10b.

#UD
- If CPUID.01H:ECX.XSAVE[bit 26] = 0.
- If CR4.OSXSAVE[bit 18] = 0.
- If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The XSETBV instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
XTEST — Test If in Transactional Execution

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D6 XTEST</td>
<td>A</td>
<td>V/V</td>
<td>HLE or RTM</td>
<td>Test if executing in a transactional region</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

**Operation**

XTEST

IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
THEN
    ZF ← 0
ELSE
    ZF ← 1
FI;

**Flags Affected**

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

**Intel C/C++ Compiler Intrinsic Equivalent**

XTEST:  int _xtest( void );

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

#UD  CPUID.(EAX=7, ECX=0):EBX.HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
      If LOCK prefix is used.
6.1 OVERVIEW

This chapter describes the Safer Mode Extensions (SMX) for the Intel 64 and IA-32 architectures. Safer Mode Extensions (SMX) provide a programming interface for system software to establish a measured environment within the platform to support trust decisions by end users. The measured environment includes:

- Measured launch of a system executive, referred to as a Measured Launched Environment (MLE)\(^1\). The system executive may be based on a Virtual Machine Monitor (VMM), a measured VMM is referred to as MVMM\(^2\).
- Mechanisms to ensure the above measurement is protected and stored in a secure location in the platform.
- Protection mechanisms that allow the VMM to control attempts to modify the VMM.

The measurement and protection mechanisms used by a measured environment are supported by the capabilities of an Intel\(^\circledR\) Trusted Execution Technology (Intel\(^\circledR\) TXT) platform:

- The SMX are the processor's programming interface in an Intel TXT platform.
- The chipset in an Intel TXT platform provides enforcement of the protection mechanisms.
- Trusted Platform Module (TPM) 1.2 in the platform provides platform configuration registers (PCRs) to store software measurement values.

6.2 SMX FUNCTIONALITY

SMX functionality is provided in an Intel 64 processor through the GETSEC instruction via leaf functions. The GETSEC instruction supports multiple leaf functions. Leaf functions are selected by the value in EAX at the time GETSEC is executed. Each GETSEC leaf function is documented separately in the reference pages with a unique mnemonic (even though these mnemonics share the same opcode, 0F 37).

6.2.1 Detecting and Enabling SMX

Software can detect support for SMX operation using the CPUID instruction. If software executes CPUID with 1 in EAX, a value of 1 in bit 6 of ECX indicates support for SMX operation (GETSEC is available), see CPUID instruction for the layout of feature flags of reported by CPUID.01H:ECX.

System software enables SMX operation by setting CR4.SMXE[Bit 14] = 1 before attempting to execute GETSEC. Otherwise, execution of GETSEC results in the processor signaling an invalid opcode exception (#UD).

If the CPUID SMX feature flag is clear (CPUID.01H.ECX[Bit 6] = 0), attempting to set CR4.SMXE[Bit 14] results in a general protection exception.

The IA32FEATURE_CONTROL MSR (at address 03AH) provides feature control bits that configure operation of VMX and SMX. These bits are documented in Table 6-1.

---

2. An MVMM is sometimes referred to as a measured launched environment (MLE). See Intel\(^\circledR\) Trusted Execution Technology Measured Launched Environment Programming Guide.
• Bit 0 is a lock bit. If the lock bit is clear, an attempt to execute VMXON will cause a general-protection exception. Attempting to execute GETSEC[SENTER] when the lock bit is clear will also cause a general-protection exception. If the lock bit is set, WRMSR to the IA32_FEATURE_CONTROL MSR will cause a general-protection exception. Once the lock bit is set, the MSR cannot be modified until a power-on reset. System BIOS can use this bit to provide a setup option for BIOS to disable support for VMX, SMX or both VMX and SMX.
• Bit 1 enables VMX in SMX operation (between executing the SENTER and SEXIT leaves of GETSEC). If this bit is clear, an attempt to execute VMXON in SMX will cause a general-protection exception if executed in SMX operation. Attempts to set this bit on logical processors that do not support both VMX operation (Chapter 6, “Safer Mode Extensions Reference”) and SMX operation cause general-protection exceptions.
• Bit 2 enables VMX outside SMX operation. If this bit is clear, an attempt to execute VMXON will cause a general-protection exception if executed outside SMX operation. Attempts to set this bit on logical processors that do not support VMX operation cause general-protection exceptions.
• Bits 8 through 14 specify enabled functionality of the SENTER leaf function. Each bit in the field represents an enable control for a corresponding SENTER function. Only enabled SENTER leaf functionality can be used when executing SENTER.
• Bits 15 specify global enable of all SENTER functionalities.

### 6.2.2 SMX Instruction Summary

System software must first query for available GETSEC leaf functions by executing GETSEC[CAPABILITIES]. The CAPABILITIES leaf function returns a bit map of available GETSEC leaves. An attempt to execute an unsupported leaf index results in an undefined opcode (#UD) exception.

#### 6.2.2.1 GETSEC[CAPABILITIES]

The SMX functionality provides an architectural interface for newer processor generations to extend SMX capabilities. Specifically, the GETSEC instruction provides a capability leaf function for system software to discover the available GETSEC leaf functions that are supported in a processor. Table 6-2 lists the currently available GETSEC leaf functions.
6.2.2.2 GETSEC[ENTERACCS]

The GETSEC[ENTERACCS] leaf enables authenticated code execution mode. The ENTERACCS leaf function performs an authenticated code module load using the chipset public key as the signature verification. ENTERACCS requires the existence of an Intel® Trusted Execution Technology capable chipset since it unlocks the chipset private configuration register space after successful authentication of the loaded module. The physical base address and size of the authenticated code module are specified as input register values in EBX and ECX, respectively.

While in the authenticated code execution mode, certain processor state properties change. For this reason, the time in which the processor operates in authenticated code execution mode should be limited to minimize impact on external system events.

Upon entry into , the previous paging context is disabled (since the authenticated code module image is specified with physical addresses and can no longer rely upon external memory-based page-table structures).

Prior to executing the GETSEC[ENTERACCS] leaf, system software must ensure the logical processor issuing GETSEC[ENTERACCS] is the boot-strap processor (BSP), as indicated by IA32_APIC_BASE.BSP = 1. System software must ensure other logical processors are in a suitable idle state and not marked as BSP.

The GETSEC[ENTERACCS] leaf may be used by different agents to load different authenticated code modules to perform functions related to different aspects of a measured environment, for example system software and Intel® TXT enabled BIOS may use more than one authenticated code modules.

6.2.2.3 GETSEC(EXITAC)

GETSEC(EXITAC) takes the processor out of . When this instruction leaf is executed, the contents of the authenticated code execution area are scrubbed and control is transferred to the non-authenticated context defined by a near pointer passed with the GETSEC(EXITAC) instruction.

The authenticated code execution area is no longer accessible after completion of GETSEC(EXITAC). RBX (or EBX) holds the address of the near absolute indirect target to be taken.

6.2.2.4 GETSEC[SENTER]

The GETSEC[SENTER] leaf function is used by the initiating logical processor (ILP) to launch an MLE. GETSEC[SENTER] can be considered a superset of the ENTERACCS leaf, because it enters as part of the measured environment launch.

Measured environment startup consists of the following steps:

<table>
<thead>
<tr>
<th>Index (EAX)</th>
<th>Leaf function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CAPABILITIES</td>
<td>Returns the available leaf functions of the GETSEC instruction.</td>
</tr>
<tr>
<td>1</td>
<td>Undefined</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>ENTERACCS</td>
<td>Enter</td>
</tr>
<tr>
<td>3</td>
<td>EXITAC</td>
<td>Exit</td>
</tr>
<tr>
<td>4</td>
<td>SENTER</td>
<td>Launch an MLE.</td>
</tr>
<tr>
<td>5</td>
<td>SEXIT</td>
<td>Exit the MLE.</td>
</tr>
<tr>
<td>6</td>
<td>PARAMETERS</td>
<td>Return SMX related parameter information.</td>
</tr>
<tr>
<td>7</td>
<td>SMCTRL</td>
<td>SMX mode control.</td>
</tr>
<tr>
<td>8</td>
<td>WAKEUP</td>
<td>Wake up sleeping processors in safer mode.</td>
</tr>
<tr>
<td>9 - (4G-1)</td>
<td>Undefined</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
• the ILP rendezvous the responding logical processors (RLPs) in the platform into a controlled state (At the completion of this handshake, all the RLPs except for the ILP initiating the measured environment launch are placed in a newly defined SENTER sleep state).
• Load and authenticate the authenticated code module required by the measured environment, and enter authenticated code execution mode.
• Verify and lock certain system configuration parameters.
• Measure the dynamic root of trust and store into the PCRs in TPM.
• Transfer control to the MLE with interrupts disabled.

Prior to executing the GETSEC[SENTER] leaf, system software must ensure the platform’s TPM is ready for access and the ILP is the boot-strap processor (BSP), as indicated by IA32_APIC_BASE.BSP. System software must ensure other logical processors (RLPs) are in a suitable idle state and not marked as BSP.

System software launching a measurement environment is responsible for providing a proper authenticate code module address when executing GETSEC[SENTER]. The AC module responsible for the launch of a measured environment and loaded by GETSEC[SENTER] is referred to as SINIT. See Intel® Trusted Execution Technology Measured Launched Environment Programming Guide for additional information on system software requirements prior to executing GETSEC[SENTER].

6.2.2.5 GETSEC[SEXIT]

System software exits the measured environment by executing the instruction GETSEC[SEXIT] on the ILP. This instruction rendezvous the responding logical processors in the platform for exiting from the measured environment. External events (if left masked) are unmasked and Intel® TXT-capable chipset's private configuration space is re-locked.

6.2.2.6 GETSEC[PARAMETERS]

The GETSEC[PARAMETERS] leaf function is used to report attributes, options and limitations of SMX operation. Software uses this leaf to identify operating limits or additional options.

The information reported by GETSEC[PARAMETERS] may require executing the leaf multiple times using EBX as an index. If the GETSEC[PARAMETERS] instruction leaf or if a specific parameter field is not available, then SMX operation should be interpreted to use the default limits of respective GETSEC leaves or parameter fields defined in the GETSEC[PARAMETERS] leaf.

6.2.2.7 GETSEC[SMCTRL]

The GETSEC[SMCTRL] leaf function is used for providing additional control over specific conditions associated with the SMX architecture. An input register is supported for selecting the control operation to be performed. See the specific leaf description for details on the type of control provided.

6.2.2.8 GETSEC[WAKEUP]

Responding logical processors (RLPs) are placed in the SENTER sleep state after the initiating logical processor executes GETSEC[SENTER]. The ILP can wake up RLPs to join the measured environment by using GETSEC[WAKEUP]. When the RLPs in SENTER sleep state wake up, these logical processors begin execution at the entry point defined in a data structure held in system memory (pointed to by a chipset register LT.MLE.JOIN) in TXT configuration space.

6.2.3 Measured Environment and SMX

This section gives a simplified view of a representative life cycle of a measured environment that is launched by a system executive using SMX leaf functions. Intel® Trusted Execution Technology Measured Launched Environment Programming Guide provides more detailed examples of using SMX and chipset resources (including chipset registers, Trusted Platform Module) to launch an MVMM.
The life cycle starts with the system executive (an OS, an OS loader, and so forth) loading the MLE and SINIT AC module into available system memory. The system executive must validate and prepare the platform for the measured launch. When the platform is properly configured, the system executive executes GETSEC[SENTER] on the initiating logical processor (ILP) to rendezvous the responding logical processors into an SENTER sleep state, the ILP then enters into using the SINIT AC module. In a multi-threaded or multi-processing environment, the system executive must ensure that other logical processors are already in an idle loop, or asleep (such as after executing HLT) before executing GETSEC[SENTER].

After the GETSEC[SENTER] rendezvous handshake is performed between all logical processors in the platform, the ILP loads the chipset authenticated code module (SINIT) and performs an authentication check. If the check passes, the processor hashes the SINIT AC module and stores the result into TPM PCR 17. It then switches execution context to the SINIT AC module. The SINIT AC module will perform a number of platform operations, including: verifying the system configuration, protecting the system memory used by the MLE from I/O devices capable of DMA, producing a hash of the MLE, storing the hash value in TPM PCR 18, and various other operations. When SINIT completes execution, it executes the GETSEC[EXITAC] instruction and transfers control the MLE at the designated entry point.

Upon receiving control from the SINIT AC module, the MLE must establish its protection and isolation controls before enabling DMA and interrupts and transferring control to other software modules. It must also wakeup the RLPs from their SENTER sleep state using the GETSEC[WAKEUP] instruction and bring them into its protection and isolation environment.

While executing in a measured environment, the MVMM can access the Trusted Platform Module (TPM) in locality 2. The MVMM has complete access to all TPM commands and may use the TPM to report current measurement values or use the measurement values to protect information such that only when the platform configuration registers (PCRs) contain the same value is the information released from the TPM. This protection mechanism is known as sealing.

A measured environment shutdown is ultimately completed by executing GETSEC[SEXIT]. Prior to this step system software is responsible for scrubbing sensitive information left in the processor caches, system memory.

### 6.3 GETSEC LEAF FUNCTIONS

This section provides detailed descriptions of each leaf function of the GETSEC instruction. GETSEC is available only if CPUID.01H:ECX[Bit 6] = 1. This indicates the availability of SMX and the GETSEC instruction. Before GETSEC can be executed, SMX must be enabled by setting CR4.SMXE[Bit 14] = 1.

A GETSEC leaf can only be used if it is shown to be available as reported by the GETSEC[CAPABILITIES] function. Attempts to access a GETSEC leaf index not supported by the processor, or if CR4.SMXE is 0, results in the signaling of an undefined opcode exception.

All GETSEC leaf functions are available in protected mode, including the compatibility sub-mode of IA-32e mode and the 64-bit sub-mode of IA-32e mode. Unless otherwise noted, the behavior of all GETSEC functions and interactions related to the measured environment are independent of IA-32e mode. This also applies to the interpretation of register widths passed as input parameters to GETSEC functions and to register results returned as output parameters.

The GETSEC functions ENTERACCS, SENTER, SEXIT, and WAKEUP require a Intel® TXT capable-chipset to be present in the platform. The GETSEC[CAPABILITIES] returned bit vector in position 0 indicates an Intel® TXT-capable chipset has been sampled present by the processor.

The processor’s operating mode also affects the execution of the following GETSEC leaf functions: SMCTRL, ENTERACCS, EXITAC, SENTER, SEXIT, and WAKEUP. These functions are only allowed in protected mode at CPL = 0. They

1. This chapter uses the 64-bit notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because processors that support SMX also support Intel 64 Architecture. The MVMM can be launched in IA-32e mode or outside IA-32e mode. The 64-bit notation of processor registers also refer to its 32-bit forms if SMX is used in 32-bit environment. In some places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register
2. Sampled present means that the processor sent a message to the chipset and the chipset responded that it (a) knows about the message and (b) is capable of executing SENTER. This means that the chipset CAN support Intel® TXT, and is configured and WILLING to support it.
are not allowed while in SMM in order to prevent potential intra-mode conflicts. Further execution qualifications exist to prevent potential architectural conflicts (for example: nesting of the measured environment or authenticated code execution mode). See the definitions of the GETSEC leaf functions for specific requirements.

For the purpose of performance monitor counting, the execution of GETSEC functions is counted as a single instruction with respect to retired instructions. The response by a responding logical processor (RLP) to messages associated with GETSEC[SENTER] or GTSEC[SEXIT] is transparent to the retired instruction count on the ILP.
GETSEC[CAPABILITIES] - Report the SMX Capabilities

The GETSEC[CAPABILITIES] function returns a bit vector of supported GETSEC leaf functions. The CAPABILITIES leaf of GETSEC is selected with EAX set to 0 at entry. EBX is used as the selector for returning the bit vector field in EAX. GETSEC[CAPABILITIES] may be executed at all privilege levels, but the CR4.SMXE bit must be set or an undefined opcode exception (#UD) is returned.

With EBX = 0 upon execution of GETSEC[CAPABILITIES], EAX returns the a bit vector representing status on the presence of a Intel® TXT-capable chipset and the first 30 available GETSEC leaf functions. The format of the returned bit vector is provided in Table 6-3.

If bit 0 is set to 1, then an Intel® TXT-capable chipset has been sampled present by the processor. If bits in the range of 1-30 are set, then the corresponding GETSEC leaf function is available. If the bit value at a given bit index is 0, then the GETSEC leaf function corresponding to that index is unsupported and attempted execution results in a #UD.

Bit 31 of EAX indicates if further leaf indexes are supported. If the Extended Leafs bit 31 is set, then additional leaf functions are accessed by repeating GETSEC[CAPABILITIES] with EBX incremented by one. When the most significant bit of EAX is not set, then additional GETSEC leaf functions are not supported; indexing EBX to a higher value results in EAX returning zero.

Table 6-3. Getsec Capability Result Encoding (EBX = 0)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chipset Present</td>
<td>0</td>
<td>Intel® TXT-capable chipset is present.</td>
</tr>
<tr>
<td>Undefined</td>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>ENTERACCS</td>
<td>2</td>
<td>GETSEC[ENTERACCS] is available.</td>
</tr>
<tr>
<td>EXITAC</td>
<td>3</td>
<td>GETSEC[EXITAC] is available.</td>
</tr>
<tr>
<td>SENTER</td>
<td>4</td>
<td>GETSEC[SENTER] is available.</td>
</tr>
<tr>
<td>SEXIT</td>
<td>5</td>
<td>GETSEC[SEXIT] is available.</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>6</td>
<td>GETSEC[PARAMETERS] is available.</td>
</tr>
<tr>
<td>SMCTRL</td>
<td>7</td>
<td>GETSEC[SMCTRL] is available.</td>
</tr>
<tr>
<td>WAKEUP</td>
<td>8</td>
<td>GETSEC[WAKEUP] is available.</td>
</tr>
<tr>
<td>Undefined</td>
<td>30:9</td>
<td>Reserved</td>
</tr>
<tr>
<td>Extended Leafs</td>
<td>31</td>
<td>Reserved for extended information reporting of GETSEC capabilities.</td>
</tr>
</tbody>
</table>
Operation
IF (CR4.SMXE=0)
   THEN #UD;
ELSIF (in VMX non-root operation)
   THEN VM Exit (reason="GETSEC instruction");
IF (EBX=0) THEN
   BitVector← 0;
   IF (TXT chipset present)
      BitVector[Chipset present]← 1;
   IF (ENTERACCS Available)
      THEN BitVector[ENTERACCS]← 1;
   IF (EXITAC Available)
      THEN BitVector[EXITAC]← 1;
   IF (INNER Available)
      THEN BitVector[INNER]← 1;
   IF (SEXIT Available)
      THEN BitVector[SEXIT]← 1;
   IF (PARAMETERS Available)
      THEN BitVector[PARAMETERS]← 1;
   IF (SMCTRL Available)
      THEN BitVector[SMCTRL]← 1;
   IF (WAKEUP Available)
      THEN BitVector[WAKEUP]← 1;
   EAX← BitVector;
ELSE
   EAX← 0;
END;;

Flags Affected
None

Use of Prefixes
LOCK Causes #UD
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size Causes #UD
Segment overrides Ignored
Address size Ignored
REX Ignored

Protected Mode Exceptions
#UD IF CR4.SMXE = 0.

Real-Address Mode Exceptions
#UD IF CR4.SMXE = 0.

Virtual-8086 Mode Exceptions
#UD IF CR4.SMXE = 0.

Compatibility Mode Exceptions
#UD IF CR4.SMXE = 0.
64-Bit Mode Exceptions

#UD  IF CR4.SMXE = 0.

VM-exit Condition

Reason (GETSEC)  IF in VMX non-root operation.
GETSEC[ENTERACCS] - Execute Authenticated Chipset Code

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 37</td>
<td>GETSEC[ENTERACCS]</td>
<td>Enter authenticated code execution mode. EBX holds the authenticated code module physical base address. ECX holds the authenticated code module size (bytes).</td>
</tr>
</tbody>
</table>

Description

The GETSEC[ENTERACCS] function loads, authenticates and executes an authenticated code module using an Intel® TXT platform chipset's public key. The ENTERACCS leaf of GETSEC is selected with EAX set to 2 at entry.

There are certain restrictions enforced by the processor for the execution of the GETSEC[ENTERACCS] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled, that is, CR0.CD and CR0.NW bits must be 0.
- For processor packages containing more than one logical processor, CR0.CD is checked to ensure consistency between enabled logical processors.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor must not already be in authenticated code execution mode as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENTER] instruction without a subsequent exiting using GETSEC[EXITAC].
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[ENTERACCS] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_APIC_BASE.BSP (Bit 8).

Failure to conform to the above conditions results in the processor signaling a general protection exception.

Prior to execution of the ENTERACCS leaf, other logical processors, i.e., RLPs, in the platform must be:

- Idle in a wait-for-SIPI state (as initiated by an INIT assertion or through reset for non-BSP designated processors), or
- In the SENTER sleep state as initiated by a GETSEC[SENTER] from the initiating logical processor (ILP).

If other logical processor(s) in the same package are not idle in one of these states, execution of ENTERACCS signals a general protection exception. The same requirement and action applies if the other logical processor(s) of the same package do not have CR0.CD = 0.

A successful execution of ENTERACCS results in the ILP entering an authenticated code execution mode. Prior to reaching this point, the processor performs several checks. These include:

- Establish and check the location and size of the specified authenticated code module to be executed by the processor.
- Inhibit the ILP’s response to the external events: INIT, A20M, NMI and SMI.
- Broadcast a message to enable protection of memory and I/O from other processor agents.
- Load the designated code module into an authenticated code execution area.
- Isolate the contents of the authenticated code execution area from further state modification by external agents.
- Authenticate the authenticated code module.
- Initialize the initiating logical processor state based on information contained in the authenticated code module header.
- Unlock the Intel® TXT-capable chipset private configuration space and TPM locality 3 space.
GETSEC[ENTERACCS] - Execute Authenticated Chipset Code

As an integrity check for proper processor hardware operation, execution of GETSEC[ENTERACCS] will also check exception. condition results in an abort of the authenticated code execution launch and the signaling of a general protection exception. An error verified to be a multiple of 64, that it does not exceed the internal authenticated code execution area capacity (as reported by GETSEC[CAPABILITIES]), and that the top address of the AC module does not exceed 32 bits. An error condition results in an abort of the authenticated code execution launch and the signaling of a general protection exception. As an integrity check for proper processor hardware operation, execution of GETSEC[ENTERACCS] will also check the contents of all the machine check status registers (as reported by the MSRs IA32_MCI_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32_MCG_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must not be asserted, indicating that no machine check exception processing is currently in progress. These checks are performed prior to initiating the load of the authenticated code module. Any outstanding valid uncorrectable machine check error present in these status registers at this point will result in the processor signaling a general protection violation. The ILP masks the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. This masking remains active until optionally unmasked by GETSEC[EXITAC] (this defined unmasking behavior assumes GETSEC[ENTERACCS] was not executed by a prior GETSEC[SENTER]). The purpose of this masking control is to prevent exposure to existing external event handlers that may not be under the control of the authenticated code module. The ILP sets an internal flag to indicate it has entered authenticated code execution mode. The state of the A20M pin is likewise masked and forced internally to a de-asserted state so that any external assertion is not recognized during authenticated code execution mode. To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) access and I/O originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Prior to launching the authenticated execution module using GETSEC[ENTERACCS] or GETSEC[SENTER], the processor’s MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (writeback). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. If the processor detected this requirement is not met, it will signal an Intel® TXT reset condition with an error code during the loading of the authenticated code module. While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default).

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

At the successful completion of GETSEC[ENTERACCS], the architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDTRBasePtr + module base address held in EBX and the GDTR limit is set to the value in the GDTLimit field. The CS selector is initialized to the AC module header SegSel field, while the DS selector is initialized to CS + 8. The segment descriptor fields are implicitly initialized to BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header EntryPoint field + module base address (EBX). The AC module based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.
A summary of the register state initialization after successful completion of GETSEC[ENTERACCS] is given for the processor in Table 6-4. The paging is disabled upon entry into authenticated code execution mode. The authenticated code module is loaded and initially executed using physical addresses. It is up to the system software after execution of GETSEC[ENTERACCS] to establish a new (or restore its previous) paging environment with an appropriate mapping to meet new protection requirements. EBP is initialized to the authenticated code module base physical address for initial execution in the authenticated environment. As a result, the authenticated code can reference EBP for relative address based references, given that the authenticated code module must be position independent.

Table 6-4. Register State Initialization after GETSEC[ENTERACCS]

<table>
<thead>
<tr>
<th>Register State</th>
<th>Initialization Status</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR0</td>
<td>PG←0, AM←0, WP←0: Others unchanged</td>
<td>Paging, Alignment Check, Write-protection are disabled.</td>
</tr>
<tr>
<td>CR4</td>
<td>MCE←0: Others unchanged</td>
<td>Machine Check Exceptions disabled.</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>00000002H</td>
<td>IA-32e mode disabled.</td>
</tr>
<tr>
<td>IA32_EFER</td>
<td>0H</td>
<td>AC.base is in EBX as input to GETSEC[ENTERACCS].</td>
</tr>
<tr>
<td>EIP</td>
<td>AC.base + EntryPoint</td>
<td>Carry forward 64-bit processor state across GETSEC[ENTERACCS].</td>
</tr>
<tr>
<td>ECX</td>
<td>Pre-ENTERACCS state: [31:16]=GDTR.limit; [15:0]=CS.sel</td>
<td>Carry forward 64-bit processor state across GETSEC[ENTERACCS].</td>
</tr>
<tr>
<td>[E</td>
<td>R]DX</td>
<td>Pre-ENTERACCS state: GDTR base</td>
</tr>
<tr>
<td>EBP</td>
<td>AC.base</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Sel=[SegSel], base=0, limit=FFFFFFh, G=1, D=1, AR=9BH</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>Sel=[SegSel] +8, base=0, limit=FFFFFFh, G=1, D=1, AR=93H</td>
<td></td>
</tr>
<tr>
<td>GDTR</td>
<td>Base= AC.base (EBX) + [GDTBasePtr], Limit=[GDTLimit]</td>
<td></td>
</tr>
<tr>
<td>DR7</td>
<td>000000400H</td>
<td></td>
</tr>
<tr>
<td>IA32_DEBUGCTL</td>
<td>0H</td>
<td>The number of initialized fields may change due to processor implementation.</td>
</tr>
<tr>
<td>IA32_MISC_ENABLE</td>
<td>See Table 6-5 for example.</td>
<td></td>
</tr>
</tbody>
</table>

The segmentation related processor state that has not been initialized by GETSEC[ENTERACCS] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in ES, SS, FS, GS, TR, and LDTR might not be valid.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by ENTERACCS. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be reestablished in order to establish IA-32e mode while operating in authenticated code execution mode.

Debug exception and trap related signaling is also disabled as part of GETSEC[ENTERACCS]. This is achieved by resetting DR7, TF in EFLAGS, and the MSR IA32_DEBUGCTL. These debug functions are free to be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly initialized following
entry into authenticated code execution mode. Also, any pending single-step trap condition will have been cleared upon entry into this mode.

The IA32_MISC_ENABLE MSR is initialized upon entry into authenticated execution mode. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings (See the footnote for Table 6-5.). The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. One of the impacts of initializing this MSR is any previous condition established by the MONITOR instruction will be cleared.

To support the possible return to the processor architectural state prior to execution of GETSEC[ENTERACCS], certain critical processor state is captured and stored in the general-purpose registers at instruction completion. [E|R]BX holds effective address ([E|R]IP) of the instruction that would execute next after GETSEC[ENTERACCS], ECX[15:0] holds the CS selector value, ECX[31:16] holds the GDTR limit field, and [E|R]DX holds the GDTR base field. The subsequent authenticated code can preserve the contents of these registers so that this state can be manually restored if needed, prior to exiting authenticated code execution mode with GETSEC[EXITAC]. For the processor state after exiting authenticated code execution mode, see the description of GETSEC[SEXIT].

Table 6-5. IA32_MISC_ENABLE MSR Initialization1 by ENTERACCS and SENTER

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast strings enable</td>
<td>0</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>FOPCODE compatibility mode enable</td>
<td>2</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Thermal monitor enable</td>
<td>3</td>
<td>Set to 1 if other thermal monitor capability is not enabled.2</td>
</tr>
<tr>
<td>Split-lock disable</td>
<td>4</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Bus lock on cache line splits disable</td>
<td>8</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Hardware prefetch disable</td>
<td>9</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>GV1/2 legacy enable</td>
<td>15</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>MONITOR/MWAIT s/m enable</td>
<td>18</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Adjacent sector prefetch disable</td>
<td>19</td>
<td>Clear to 0.</td>
</tr>
</tbody>
</table>

NOTES:
1. The number of IA32_MISC_ENABLE fields that are initialized may vary due to processor implementations.
2. ENTERACCS (and SENTER) initialize the state of processor thermal throttling such that at least a minimum level is enabled. If thermal throttling is already enabled when executing one of these GETSEC leaves, then no change in the thermal throttling control settings will occur. If thermal throttling is disabled, then it will be enabled via setting of the thermal throttle control bit 3 as a result of executing these GETSEC leaves.

The IDTR will also require reloading with a new IDT context after entering authenticated code execution mode, before any exceptions or the external interrupts INTR and NMI can be handled. Since external interrupts are re-enabled at the completion of authenticated code execution mode (as terminated with EXITAC), it is recommended that a new IDT context be established before this point. Until such a new IDT context is established, the programmer must take care in not executing an INT n instruction or any other operation that would result in an exception or trap signaling.

Prior to completion of the GETSEC[ENTERACCS] instruction and after successful authentication of the AC module, the private configuration space of the Intel TXT chipset is unlocked. The authenticated code module alone can gain access to this normally restricted chipset state for the purpose of securing the platform.

Once the authenticated code module is launched at the completion of GETSEC[ENTERACCS], it is free to enable interrupts by setting EFLAGS.IF and enable NMI by execution of IRET. This presumes that it has re-established interrupt handling support through initialization of the IDT, GDT, and corresponding interrupt handling code.
Operation in a Uni-Processor Platform

(* The state of the internal flag ACMODEFLAG persists across instruction boundary *)

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (ACMODEFLAG=1) or (IN_SMM=1))
    THEN #GP(0);

IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF (IA32_MC[I]_STATUS
            =
            uncorrectable error)
            THEN #GP(0);
    OD;
FI;

IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);

ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64 ) ≠ 0 ) or (ACSIZE < minimum module size) OR (ACSIZE > authenticated RAM capacity)) or ((ACBASE+ACSIZE) > (2^32 -1)))
    THEN #GP(0);

IF (secondary thread(s) CR0.CD = 1) or ((secondary thread(s) NOT(wait-for-SIPI)) and
    (secondary thread(s) not in SENTER sleep state)
    THEN #GP(0);

Mask SMI, INIT, A20M, and NMI external pin events;
IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain Outgoing Transactions;
ACMODEFLAG← 1;
SignalTXTMessage(ProcessorHold);
Load the internal ACRAM based on the AC module size;
(* Ensure that all ACRAM loads hit Write Back memory space *)
IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version isnot supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
(* Authenticate the AC Module and shutdown with an error if it fails *)
KEY← GETKEY(ACRAM, ACBASE);
KEYHASH← HASH(KEY);
CSKEYHASH← READ(TXT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LENGTH_CONST - 1 DO
ACRAM[SCRATCH.I] ← SIGNATURE[I];
COMPUTEDSIGNATURE ← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LENGTH_CONST - 1 DO
ACRAM[SCRATCH.SIGNATURE_LENGTH_CONST+I] ← COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL ← ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
(ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) ≥ ACSIZE)
THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
THEN ACEntryPoint ← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
ACEntryPoint ← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint > ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
THEN TXT-SHUTDOWN(#BadACMFormat);
IF (ACRAM[GDTLimit] & FFFF0000h)
THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8))
THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL ≠ 0))
THEN TXT-SHUTDOWN(#BadACMFormat);
CR0.[PG,AM,WP] ← 0;
CR4.MCE ← 0;
EFLAGS ← 00000002h;
IA32_EFER ← 0;
[E|R]BX ← [E|R]IP of the instruction after GETSEC[ENTERACCS];
ECX ← Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
[E|R]DX ← Pre-GETSEC[ENTERACCS] GDT.base;
EBP ← ACBASE;
GDTR.BASE ← ACBASE+ACRAM[GDTBasePtr];
GDTRLIMIT ← ACRAM[GDTLimit];
CS.SEL ← ACRAM[SegSel];
CS.BASE ← 0;
CS.LIMIT ← FFFFH;
CS.G ← 1;
CS.D ← 1;
CS.AR ← 9Bh;
DS.SEL ← ACRAM[SegSel]+8;
DS.BASE ← 0;
DS.LIMIT ← FFFFH;
DS.G ← 1;
DS.D ← 1;
DS.AR ← 93h;
DR7 ← 00000400h;
IA32_DEBUGCTL ← 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP ← ACEntryPoint;
END;
Flags Affected
All flags are cleared.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].
If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
If a Intel® TXT-capable chipset is not present.
If in VMX root operation.
If the initiating processor is not designated as the bootstrap processor via the MSR bit
IA32_APIC_BASE.BSP.
If the processor is already in authenticated code execution mode.
If the processor is in SMM.
If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.
If the authenticated code base is not on a 4096 byte boundary.
If the authenticated code size > processor internal authenticated code area capacity.
If the authenticated code size is not modulo 64.
If other enabled logical processor(s) of the same package CR0.CD = 1.
If other enabled logical processor(s) of the same package are not in the wait-for-SIPI or
ENTER sleep state.

Real-Address Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[ENTERACCS] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[ENTERACCS] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
All protected mode exceptions apply.
#GP If AC code module does not reside in physical address below 2^32 -1.

64-Bit Mode Exceptions
All protected mode exceptions apply.
#GP If AC code module does not reside in physical address below 2^32 -1.
**VM-exit Condition**

Reason (GETSEC)  IF in VMX non-root operation.
GETSEC[EXITAC]—Exit Authenticated Code Execution Mode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 37</td>
<td>GETSEC(EXITAC)</td>
<td>Exit authenticated code execution mode. RBX holds the Near Absolute Indirect jump target and EDX hold the exit parameter flags.</td>
</tr>
</tbody>
</table>

Description

The GETSEC(EXITAC) leaf function exits the ILP out of authenticated code execution mode established by GETSEC[ENTERACCS] or GETSEC[SENTER]. The EXITAC leaf of GETSEC is selected with EAX set to 3 at entry. EBX (or RBX, if in 64-bit mode) holds the near jump target offset for where the processor execution resumes upon exiting authenticated code execution mode. EDX contains additional parameter control information. Currently only an input value of 0 in EDX is supported. All other EDX settings are considered reserved and result in a general protection violation.

GETSEC(EXITAC) can only be executed if the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0. The processor must also be in authenticated code execution mode. To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it is in SMM or in VMX operation. A violation of these conditions results in a general protection violation.

Upon completion of the GETSEC(EXITAC) operation, the processor unmasks responses to external event signals INIT#, NMI#, and SMI#. This unmasking is performed conditionally, based on whether the authenticated code execution mode was entered via execution of GETSEC[SENTER] or GETSEC[ENTERACCS]. If the processor is in authenticated code execution mode due to the execution of GETSEC[SENTER], then these external event signals will remain masked. In this case, A20M is kept disabled in the measured environment until the measured environment executes GETSEC[SEXIT]. INIT# is unconditionally unmasked by EXITAC. Note that any events that are pending, but have been blocked while in authenticated code execution mode, will be recognized at the completion of the GETSEC(EXITAC) instruction if the pin event is unmasked.

The intent of providing the ability to optionally leave the pin events SMI#, and NMI# masked is to support the completion of a measured environment bring-up that makes use of VMX. In this envisioned security usage scenario, these events will remain masked until an appropriate virtual machine has been established in order to field servicing of these events in a safer manner. Details on when and how events are masked and unmasked in VMX operation are described in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C. It should be cautioned that if no VMX environment is to be activated following GETSEC(EXITAC), that these events will remain masked until the measured environment is exited with GETSEC[SEXIT]. If this is not desired then the GETSEC function SMCTRL(0) can be used for unmasking SMI# in this context. NMI# can be correspondingly unmasked by execution of IRET.

A successful exit of the authenticated code execution mode requires the ILP to perform additional steps as outlined below:

- Invalidate the contents of the internal authenticated code execution area.
- Invalidate processor TLBs.
- Clear the internal processor AC Mode indicator flag.
- Re-lock the TPM locality 3 space.
- Unlock the Intel® TXT-capable chipset memory and I/O protections to allow memory and I/O activity by other processor agents.
- Perform a near absolute indirect jump to the designated instruction location.

The content of the authenticated code execution area is invalidated by hardware in order to protect it from further use or visibility. This internal processor storage area can no longer be used or relied upon after GETSEC(EXITAC). Data structures need to be re-established outside of the authenticated code execution area if they are to be referenced after EXITAC. Since addressed memory content formerly mapped to the authenticated code execution area may no longer be coherent with external system memory after EXITAC, processor TLBs in support of linear to physical address translation are also invalidated.

Upon completion of GETSEC(EXITAC) a near absolute indirect transfer is performed with EIP loaded with the contents of EBX (based on the current operating mode size). In 64-bit mode, all 64 bits of RBX are loaded into RIP.
if REX.W precedes GETSEC[EXITAC]. Otherwise RBX is treated as 32 bits even while in 64-bit mode. Conventional CS limit checking is performed as part of this control transfer. Any exception conditions generated as part of this control transfer will be directed to the existing IDT; thus it is recommended that an IDTR should also be established prior to execution of the EXITAC function if there is a need for fault handling. In addition, any segmentation related (and paging) data structures to be used after EXITAC should be re-established or validated by the authenticated code prior to EXITAC.

In addition, any segmentation related (and paging) data structures to be used after EXITAC need to be re-established and mapped outside of the authenticated RAM designated area by the authenticated code prior to EXITAC. Any data structure held within the authenticated RAM allocated area will no longer be accessible after completion by EXITAC.

**Operation**

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or ( (in 64-bit mode) and ( RBX is non-canonical) )
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (ACMODEFLAG=0) or (IN_SMM=1)) or (EDX ≠ 0))
    THEN #GP(0);
IF (OperandSize = 32)
    THEN tempEIP ← EBX;
ELSIF (OperandSize = 64)
    THEN tempEIP ← RBX;
ELSE
    tempEIP ← EBX AND 0000FFFFH;
IF (tempEIP > code segment limit)
    THEN #GP(0);
Invalidate ACRAM contents;
Invalidate processor TLB(s);
Drain outgoing messages;
SignalTXTMsg(CloseLocality3);
SignalTXTMsg(LockSMRAM);
SignalTXTMsg(ProcessorRelease);
Unmask INIT;
IF (SENTERFLAG=0)
    THEN Unmask SMI, INIT, NMI, and A20M pin event;
ELSEIF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ACMODEFLAG ← 0;
EIP ← tempEIP;
END;

**Flags Affected**

None.

**Use of Prefixes**

LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
Segment overrides Ignored.
Address size Ignored.
REX.W Sets 64-bit mode Operand size attribute.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC(EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) If CR0.PE = 0 or CPL>0 or EFLAGS.VM =1.
If in VMX root operation.
If the processor is not currently in authenticated code execution mode.
If the processor is in SMM.
If any reserved bit position is set in the EDX parameter register.

Real-Address Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC(EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC(EXITAC] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC(EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC(EXITAC] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
All protected mode exceptions apply.

64-Bit Mode Exceptions
All protected mode exceptions apply.
#GP(0) If the target address in RBX is not in a canonical form.

VM-Exit Condition
Reason (GETSEC) IF in VMX non-root operation.
GETSEC[ENTER]—Enter a Measured Environment

**Description**

The GETSEC[ENTER] instruction initiates the launch of a measured environment and places the initiating logical processor (ILP) into the authenticated code execution mode. The ENTER leaf of GETSEC is selected with EAX set to 4 at execution. The physical base address of the AC module to be loaded and authenticated is specified in EBX. The size of the module in bytes is specified in ECX. EDX controls the level of functionality supported by the measured environment launch. To enable the full functionality of the protected environment launch, EDX must be initialized to zero.

The authenticated code base address and size parameters (in bytes) are passed to the GETSEC[ENTER] instruction using EBX and ECX respectively. The ILP evaluates the contents of these registers according to the rules for the AC module address in GETSEC[ENTERACCS]. AC module execution follows the same rules, as set by GETSEC[ENTERACCS].

The launching software must ensure that the TPM.ACCESS_0.activeLocality bit is clear before executing the GETSEC[ENTER] instruction.

There are restrictions enforced by the processor for execution of the GETSEC[ENTER] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled using the CR0.CD and NW bits.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not be in authenticated code execution mode or already in a measured environment (as launched by a previous GETSEC[ENTERACCS] or GETSEC[ENTER] instruction).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To insure consistent handling of SIPI messages, the processor executing the GETSEC[ENTER] instruction must also be designated the BSP (boot-strap processor) as defined by A32_APIC_BASE.BSP (Bit 8).
- EDX must be initialized to a setting supportable by the processor. Unless enumeration by the GETSEC[PARAMETERS] leaf reports otherwise, only a value of zero is supported.

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction leaf starts the launch of a measured environment by initiating a rendezvous sequence for all logical processors in the platform. The rendezvous sequence involves the initiating logical processor sending a message (by executing GETSEC[ENTER]) and other responding logical processors (RLPs) acknowledging the message, thus synchronizing the RLP(s) with the ILP.

In response to a message signaling the completion of rendezvous, RLPs clear the bootstrap processor indicator flag (IA32_APIC_BASE.BSP) and enter an SENTER sleep state. In this sleep state, RLPs enter an idle processor condition while waiting to be activated after a measured environment has been established by the system executive. RLPs in the SENTER sleep state can only be activated by the GETSEC leaf function WAKEUP in a measured environment.

A successful launch of the measured environment results in the initiating logical processor entering the authenticated code execution mode. Prior to reaching this point, the ILP performs the following steps internally:

- Inhibit processor response to the external events: INIT, A20M, NMI, and SMI.

---

**Opcode Instruction Description**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 37</td>
<td>GETSEC[ENTER]</td>
<td>Launch a measured environment. EBX holds the SINIT authenticated code module physical base address. ECX holds the SINIT authenticated code module size (bytes). EDX controls the level of functionality supported by the measured environment launch.</td>
</tr>
</tbody>
</table>
• Establish and check the location and size of the authenticated code module to be executed by the ILP.
• Check for the existence of an Intel® TXT-capable chipset.
• Verify the current power management configuration is acceptable.
• Broadcast a message to enable protection of memory and I/O from activities from other processor agents.
• Load the designated AC module into authenticated code execution area.
• Isolate the content of authenticated code execution area from further state modification by external agents.
• Authenticate the AC module.
• Updated the Trusted Platform Module (TPM) with the authenticated code module's hash.
• Initialize processor state based on the authenticated code module header information.
• Unlock the Intel® TXT-capable chipset private configuration register space and TPM locality 3 space.
• Begin execution in the authenticated code module at the defined entry point.

As an integrity check for proper processor hardware operation, execution of GETSEC[SENTER] will also check the contents of all the machine check status registers (as reported by the MSRs IA32_MCI_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32_MCG_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must be not asserted, indicating that no machine check exception processing is currently in-progress. These checks are performed twice: once by the ILP prior to the broadcast of the rendezvous message to RLPs, and later in response to RLPs acknowledging the rendezvous message. Any outstanding valid uncorrectable machine check error condition present in the machine check status registers at the first check point will result in the ILP signaling a general protection violation. If an outstanding valid uncorrectable machine check error condition is present at the second check point, then this will result in the corresponding logical processor signaling the more severe TXT-shutdown condition with an error code of 12.

Before loading and authentication of the target code module is performed, the processor also checks that the current voltage and bus ratio encodings correspond to known good values supportable by the processor. The MSR IA32_PERF_STATUS values are compared against either the processor supported maximum operating target setting, system reset setting, or the thermal monitor operating target. If the current settings do not meet any of these criteria then the SENTER function will attempt to change the voltage and bus ratio select controls in a processor-specific manner. This adjustment may be to the thermal monitor, minimum (if different), or maximum operating target depending on the processor.

This implies that some thermal operating target parameters configured by BIOS may be overridden by SENTER. The measured environment software may need to take responsibility for restoring such settings that are deemed to be safe, but not necessarily recognized by SENTER. If an adjustment is not possible when an out of range setting is discovered, then the processor will abort the measured launch. This may be the case for chipset controlled settings of these values or if the controllability is not enabled on the processor. In this case it is the responsibility of the external software to program the chipset voltage ID and/or bus ratio select settings to known good values recognized by the processor, prior to executing SENTER.

**NOTE**

For a mobile processor, an adjustment can be made according to the thermal monitor operating target. For a quad-core processor the SENTER adjustment mechanism may result in a more conservative but non-uniform voltage setting, depending on the pre-SENTER settings per core.

The ILP and RLPs mask the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. The purpose of this masking control is to prevent exposure to existing external event handlers until a protected handler has been put in place to directly handle these events. Masked external pin events may be unmasked conditionally or unconditionally via the GETSEC[EXITAC], GETSEC[SEXIT], GETSEC[SMCTRL] or for specific VMX related operations such as a VM entry or the VMXOFF instruction (see respective GETSEC leaves and Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C for more details). The state of the A20M pin is masked and forced internally to a de-asserted state so that external assertion is not recognized. A20M masking as set by GETSEC[SENTER] is undone only after taking down the measured environment with the GETSEC[SEXIT] instruction or processor reset. INTR is masked by simply clearing the EFLAGS.IF bit. It is the responsibility of system software to control the processor response to INTR through appropriate management of EFLAGS.
To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) and I/O activities originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Once the authenticated code module has been loaded into the authenticated code execution area, it is protected against further modification from external bus snoops. There is also a requirement that the memory type for the authenticated code module address range be WB (via initialization of the MTRRs prior to execution of this instruction). If this condition is not satisfied, it is a violation of security and the processor will force a TXT system reset (after writing an error code to the chipset LT.ERRORCODE register). This action is referred to as a Intel® TXT reset condition. It is performed when it is considered unreliable to signal an error through the conventional exception reporting mechanism.

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

Once successful authentication has been completed by the ILP, the computed hash is stored in a trusted storage facility in the platform. The following trusted storage facility are supported:

- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 0, the computed hash is stored to the platform’s TPM at PCR17 after this register is implicitly reset. PCR17 is a dedicated register for holding the computed hash of the authenticated code module loaded and subsequently executed by the GETSEC[SENTER]. As part of this process, the dynamic PCRs 18-22 are reset so they can be utilized by subsequently software for registration of code and data modules.
- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 1, the computed hash is stored in a firmware trusted module (FTM) using a modified protocol similar to the protocol used to write to TPM’s PCR17.

After successful execution of SENTER, either PCR17 (if FTM is not enabled) or the FTM (if enabled) contains the measurement of AC code and the SENTER launching parameters.

After authentication is completed successfully, the private configuration space of the Intel® TXT-capable chipset is unlocked so that the authenticated code module and measured environment software can gain access to this normally restricted chipset state. The Intel® TXT-capable chipset private configuration space can be locked later by software writing to the chipset LT.CMD.CLOSE-PRIVATE register or unconditionally using the GETSEC[SEXIT] instruction.

The SENTER leaf function also initializes some processor architecture state for the ILP from contents held in the header of the authenticated code module. Since the authenticated code module is relocatable, all address references are relative to the base address passed in via EBX. The ILP GDTR base value is initialized to EBX + [GDTBasePtr] and GDTR limit set to [GDTLimit]. The CS selector is initialized to the value held in the AC module header field SegSel, while the DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE=0, LIMIT=FFFFFFH, G=1, D=1, P=1, S=1, read/write/accessed for DS, SS, and ES, while execute/read/accessed for CS. Execution in the authenticated code module for the ILP begins with the EIP set to EBX + [EntryPoint]. AC module defined fields used for initializing processor state are consistency checked with a failure resulting in a TXT-shutdown condition.

Table 6-6 provides a summary of processor state initialization for the ILP and RLP(s) after successful completion of GETSEC[SENTER]. For both ILP and RLP(s), paging is disabled upon entry to the measured environment. It is up to the ILP to establish a trusted paging environment, with appropriate mappings, to meet protection requirements established during the launch of the measured environment. RLP state initialization is not completed until a subsequent wake-up has been signaled by execution of the GETSEC[WAKEUP] function by the ILP.
Segmentation related processor state that has not been initialized by GETSEC[SENTER] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in FS, GS, TR, and LDTR may no longer be valid. The IDTR will also require reloading with a new IDT context after launching the measured environment before exceptions or the external interrupts INTR and NMI can be handled. In the meantime, the programmer must take care in not executing an INT n instruction or any other condition that would result in an exception or trap signaling.

Debug exception and trap related signaling is also disabled as part of execution of GETSEC[SENTER]. This is achieved by clearing DR7, TF in EFLAGS, and the MSR IA32_DEBUGCTL as defined in Table 6-6. These can be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly re-initialized following SENTER. Also, any pending single-step trap condition will be cleared at the completion of SENTER for both the ILP and RLP(s).

Performance related counters and counter control registers are cleared as part of execution of SENTER on both the ILP and RLP. This implies any active performance counters at the time of SENTER execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

Since MCE along with all other state bits (with the exception of SMXE) are cleared in CR4 upon execution of SENTER processing, any enabled machine check error condition that occurs will result in the processor performing the TXT-
shutdown action. This also applies to an RLP while in the SENTER sleep state. For each logical processor CR4.MCE must be reestablished with a valid machine check exception handler to otherwise avoid an TXT-shutdown under such conditions.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by SENTER for both the ILP and RLP. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be re-established if it is desired to enable IA-32e mode while operating in authenticated code execution mode.

The miscellaneous feature control MSR, IA32_MISC_ENABLE, is initialized as part of the measured environment launch. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings. See the footnote for Table 6-5 The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. Among the impact of initializing this MSR, any previous condition established by the MONITOR instruction will be cleared.

**Effect of MSR IA32_FEATURE_CONTROL MSR**

Bits 15:8 of the IA32_FEATURE_CONTROL MSR affect the execution of GETSEC[SENTER]. These bits consist of two fields:

- Bit 15: a global enable control for execution of SENTER.
- Bits 14:8: a parameter control field providing the ability to qualify SENTER execution based on the level of functionality specified with corresponding EDX parameter bits 6:0.

The layout of these fields in the IA32_FEATURE_CONTROL MSR is shown in Table 6-1.

Prior to the execution of GETSEC[SENTER], the lock bit of IA32_FEATURE_CONTROL MSR must be bit set to affirm the settings to be used. Once the lock bit is set, only a power-up reset condition will clear this MSR. The IA32_FEATURE_CONTROL MSR must be configured in accordance to the intended usage at platform initialization. Note that this MSR is only available on SMX or VMX enabled processors. Otherwise, IA32_FEATURE_CONTROL is treated as reserved.

The **Intel® Trusted Execution Technology Measured Launched Environment Programming Guide** provides additional details and requirements for programming measured environment software to launch in an Intel TXT platform.

**Operation in a Uni-Processor Platform**

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

**GETSEC[SENTER] (ILP only):**

IF (CR4.SMXE=0)
   THEN #UD;
ELSE IF (in VMX non-root operation)
   THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
   THEN #UD;
ELSE IF (in VMX root operation)
   THEN #UD;
ELSE IF (IA32_FEATURE_CONTROL[0]=0)
   THEN #GP(0);
ELSE IF (IA32_FEATURE_CONTROL[15]=0)
   THEN #GP(0);
ELSE IF (IA32_FEATURE_CONTROL[14:8] & EDX[6:0]) #
   THEN #GP(0);
ELSE IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
   FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
      IF IA32_MC[I].STATUS = uncorrectable error
         THEN #GP(0);
   OD;
If (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64) ≠ 0 ) or (ACSIZE < minimum
  module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2^32 -1)))
  THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENDER);
DO
  WHILE (no SignalSENTER message);

TXT_SENTER__MSG_EVENT (ILP & RLP):
Mask and clear SignalSENTER event;
Unmask SignalSEXIT event;
IF (in VMX operation)
  THEN TXT-SHUTDOWN(#IllegalEvent);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
  IF IA32_MC[I]_STATUS = uncorrectable error
    THEN TXT-SHUTDOWN(#UnrecovMCError);
  FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN TXT-SHUTDOWN(#UnrecovMCError);
IF (Voltage or bus ratio status are NOT at a known good state)
  THEN IF (Voltage select and bus ratio are internally adjustable)
    THEN
      Make product-specific adjustment on operating parameters;
    ELSE
      TXT-SHUTDOWN(#IllegalVIDBRatio);
  FI;
IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain outgoing transactions;
Clear performance monitor counters and control;
SENTERFLAG← 1;
SignalTXTMsg(SENTERAck);
IF (logical processor is not ILP)
  THEN GOTO RLP_SENTER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
  DONE← TXT.READ(LT.STS);
  WHILE (not DONE);
SignalTXTMsg(SENTERContinue);
SignalTXTMsg(ProcessorHold);
FOR I=ACBASE to ACBASE+ACSIZE-1 DO
  ACRAM[I-ACBASE].ADDR← I;
  ACRAM[I-ACBASE].DATA← LOAD(I);
OD;
IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version is not supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
KEY← GETKEY(ACRAM, ACBASE);
KEYHASH← HASH(KEY);
CSKEYHASH← LT.READ(LT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific *)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I]← SIGNATURE[I];
COMPUTEDSIGNATURE← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I]← COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL← ACRAM[CodeControl];
IF ((ACMCONTROL[0] = 0) and (ACMCONTROL[1] = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL[0] = 1) and (ACMCONTROL[1] = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN ACEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    AC EntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) or (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) or (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) or (ACRAM[SegSel].RPL ≠ 0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF (FTM_INTERFACE_ID.[3:0] = 1 ) (* Alternate FTM Interface has been enabled *)
    THEN (* TPM_LOC_CTRL_4 is located at OFED44008H, TPM_DATA_BUFFER_4 is located at OFED44080H *)
        WRITE(TPM_LOC_CTRL_4)← 01H; (* Modified HASH.START protocol *)
        (* Write to firmware storage *)
        WRITE(TPM_DATA_BUFFER_4)← SIGNATURE_LEN_CONST + 4;
        FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
            WRITE(TPM_DATA_BUFFER_4 + 2 + I)← ACRAM[SCRATCH.I];
        WRITE(TPM_DATA_BUFFER_4 + 2 + SIGNATURE_LEN_CONST)← EDX;
        WRITE(FTM_LOC_CTRL)← 06H; (* Modified protocol combining HASH.DATA and HASH.END *)
ELSE IF (FTM_INTERFACE_ID.[3:0] = 0 ) (* Use standard TPM Interface *)
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST]← EDX;
    WRITE(TPM,HASH.START)← 0;
    FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
        WRITE(TPM,HASH.DATA)← ACRAM[SCRATCH.I];
    WRITE(TPM,HASH.END)← 0;
FI;
ACMODEFLAG ← 1;
CR0.[PG.LM.WP] ← 0;
CR4 ← 00000000h;
EFLAGS ← 00000000h;
IA32_EFER ← 0;
EBP ← ACBASE;
GDTR.BASE ← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT ← ACRAM[GDTLimit];
CS.SEL ← ACRAM[SegSel];
CS.BASE ← 0;
CS.LIMIT ← FFFFFFFh;
CS.G ← 1;
CS.D ← 1;
CS.AR ← 9Bh;
DS.SEL ← ACRAM[SegSel]+8;
DS.BASE ← 0;
DS.LIMIT ← FFFFFFFh;
DS.G ← 1;
DS.D ← 1;
DS.AR ← 93h;
SS ← DS;
ES ← DS;
DR7 ← 00000000h;
IA32_DEBUGCTL ← 0;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP ← ACEEntryPoint;
END;

RLP_SENTER_ROUTINE: (RLP only)
Mask SMI, INIT, A20M, and NMI external pin events
Unmask SignalWAKEUP event;
Wait for SignalSENTERContinue message;
IA32_APIC_BASE.BSP ← 0;
GOTO SENTER sleep state;
END;

Flags Affected
All flags are cleared.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPS).
Operand size Causes #UD.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
If in VMX root operation.
If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.
If an Intel® TXT-capable chipset is not present.
If an Intel® TXT-capable chipset interface to TPM is not detected as present.
If a protected partition is already active or the processor is already in authenticated code mode.
If the processor is in SMM.
If a valid uncorrectable machine check error is logged in IA32_MC[I].STATUS.
If the authenticated code base is not on a 4096 byte boundary.
If the authenticated code size > processor's authenticated code execution area storage capacity.
If the authenticated code size is not modulo 64.

Real-Address Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SENTER] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SENTER] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
All protected mode exceptions apply.
#GP IF AC code module does not reside in physical address below 2^32 -1.

64-Bit Mode Exceptions
All protected mode exceptions apply.
#GP IF AC code module does not reside in physical address below 2^32 -1.

VM-Exit Condition
Reason (GETSEC) IF in VMX non-root operation.
GETSEC[SEXIT]—Exit Measured Environment

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 37</td>
<td>GETSEC[SEXIT]</td>
<td>Exit measured environment.</td>
</tr>
</tbody>
</table>

(EAX=5)

Description

The GETSEC[SEXIT] instruction initiates an exit of a measured environment established by GETSEC[SENTER]. The SEXIT leaf of GETSEC is selected with EAX set to 5 at execution. This instruction leaf sends a message to all logical processors in the platform to signal the measured environment exit.

There are restrictions enforced by the processor for the execution of the GETSEC[SEXIT] instruction:

- Execution is not allowed unless the processor is in protected mode (CR0.PE = 1) with CPL = 0 and EFLAGS.VM = 0.
- The processor must be in a measured environment as launched by a previous GETSEC[SENTER] instruction, but not still in authenticated code execution mode.
- To avoid potential inter-operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or in VMX operation.
- To insure consistent handling of SIPI messages, the processor executing the GETSEC[SEXIT] instruction must also be designated the BSP (bootstrap processor) as defined by the register bit IA32_APIC_BASE.BSP (bit 8).

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction initiates a sequence to rendezvous the RLPs with the ILP. It then clears the internal processor flag indicating the processor is operating in a measured environment.

In response to a message signaling the completion of rendezvous, all RLPs restart execution with the instruction that was to be executed at the time GETSEC[SEXIT] was recognized. This applies to all processor conditions, with the following exceptions:

- If an RLP executed HLT and was in this halt state at the time of the message initiated by GETSEC[SEXIT], then execution resumes in the halt state.
- If an RLP was executing MWAIT, then a message initiated by GETSEC[SEXIT] causes an exit of the MWAIT state, falling through to the next instruction.
- If an RLP was executing an intermediate iteration of a string instruction, then the processor resumes execution of the string instruction at the point which the message initiated by GETSEC[SEXIT] was recognized.
- If an RLP is still in the SENTER sleep state (never awakened with GETSEC[WAKEUP]), it will be sent to the wait-for-SIPI state after first clearing the bootstrap processor indicator flag (IA32_APIC_BASE.BSP) and any pending SIPI state. In this case, such RLPs are initialized to an architectural state consistent with having taken a soft reset using the INIT# pin.

Prior to completion of the GETSEC[SEXIT] operation, both the ILP and any active RLPs unmask the response of the external event signals INIT#, A20M, NMI#, and SMI#. This unmasking is performed unconditionally to recognize pin events which are masked after a GETSEC[SENTER]. The state of A20M is unmasked, as the A20M pin is not recognized while the measured environment is active.

On a successful exit of the measured environment, the ILP re-locks the Intel® TXT-capable chipset private configuration space. GETSEC[SEXIT] does not affect the content of any PCR.

At completion of GETSEC[SEXIT] by the ILP, execution proceeds to the next instruction. Since EFLAGS and the debug register state are not modified by this instruction, a pending trap condition is free to be signaled if previously enabled.
Operation in a Uni-Processor Platform
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

GETSEC[SEXIT] (ILP only):
IF (CR4.SMXE=0)
  THEN #UD;
ELSE IF (in VMX non-root operation)
  THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
  THEN #UD;
ELSE IF (in VMX root operation) or
  (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
  (IA32_APIC_BASE.BSP=0) or
  (TXT chipset not present) or
  (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=1)
  THEN #GP(0);
SignalTXTMsg(SEXIT);
DO
  WHILE (no SignalSEXIT message);

TXT_SEXIT_MSG_EVENT (ILP & RLP):
Mask and clear SignalSEXIT event;
Clear MONITOR FSM;
Unmask SignalSENTER event;
IF (in VMX operation)
  THEN TXT-SHUTDOWN(#IllegalEvent);
SignalTXTMsg(SEXITAck);
IF (logical processor is not ILP)
  THEN GOTO RLP_SEXIT_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
  DONE← READ(LT.STS);
  WHILE (NOT DONE);
  SignalTXTMsg(SEXITContinue);
  SignalTXTMsg(ClosePrivate);
  SENTERFLAG← 0;
  Unmask SMI, INIT, A20M, and NMI external pin events;
END;

RLP_SEXIT_ROUTINE (RLPs only):
Wait for SignalSEXITContinue message;
Unmask SMI, INIT, A20M, and NMI external pin events;
IF (prior execution state = HLT)
  THEN reenter HLT state;
IF (prior execution state = SENTER sleep)
  THEN
    IA32_APIC_BASE.BSP← 0;
    Clear pending SIPI state;
    Call INIT_PROCESSOR_STATE;
    Unmask SIPI event;
    GOTO WAIT-FOR-SIPI;
  FI;
END;
Flags Affected
ILP: None.
RLPs: all flags are modified for an RLP. returning to wait-for-SIPI state, none otherwise.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
If in VMX root operation.
If the initiating processor is not designated via the MSR bit IA32_APIC_BASE.BSP.
If an Intel® TXT-capable chipset is not present.
If a protected partition is not already active or the processor is already in authenticated code mode.
If the processor is in SMM.

Real-Address Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SEXIT] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SEXIT] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
All protected mode exceptions apply.

64-Bit Mode Exceptions
All protected mode exceptions apply.

VM-Exit Condition
Reason (GETSEC) IF in VMX non-root operation.
GETSEC[PARAMETERS]—Report the SMX Parameters

**Description**

The GETSEC[PARAMETERS] instruction returns specific parameter information for SMX features supported by the processor. Parameter information is returned in EAX, EBX, and ECX, with the input parameter selected using EBX.

Software retrieves parameter information by searching with an input index for EBX starting at 0, and then reading the returned results in EAX, EBX, and ECX. EAX[4:0] is designated to return a parameter type field indicating if a parameter is available and what type it is. If EAX[4:0] is returned with 0, this designates a null parameter and indicates no more parameters are available.

Table 6-7 defines the parameter types supported in current and future implementations.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 37</td>
<td>GETSEC[PARAMETERS]</td>
<td>Report the SMX parameters. The parameters index is input in EBX with the result returned in EAX, EBX, and ECX.</td>
</tr>
</tbody>
</table>

**Table 6-7. SMX Reporting Parameters Format**

<table>
<thead>
<tr>
<th>Parameter Type EAX[4:0]</th>
<th>Parameter Description</th>
<th>EAX[31:5]</th>
<th>EBX[31:0]</th>
<th>ECX[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>Reserved (0 returned)</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>1</td>
<td>Supported AC module versions</td>
<td>Reserved (0 returned)</td>
<td>Version comparison mask</td>
<td>Version numbers supported</td>
</tr>
<tr>
<td>2</td>
<td>Max size of authenticated code execution area</td>
<td>Multiply by 32 for size in bytes</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>3</td>
<td>External memory types supported during AC mode</td>
<td>Memory type bit mask</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>4</td>
<td>Selective SENTER functionality control</td>
<td>EAX[14:8] correspond to available SENTER function disable controls</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>5</td>
<td>TXT extensions support</td>
<td>TXT Feature Extensions Flags (see Table 6-8)</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>6-31</td>
<td>Undefined</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
</tbody>
</table>
Table 6-8. TXT Feature Extensions Flags

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Processor based S-CRTM</td>
<td>Returns 1 if this processor implements a processor-rooted S-CRTM capability</td>
</tr>
<tr>
<td></td>
<td>S-CRTM support</td>
<td>and 0 if not (S-CRTM is rooted in BIOS). This flag cannot be used to infer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>whether the chipset supports TXT or whether the processor support SMX.</td>
</tr>
<tr>
<td>6</td>
<td>Machine Check Handling</td>
<td>Returns 1 if it machine check status registers can be preserved through</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENTERACCS and SENTER. If this bit is 1, the caller of ENTERACCS and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SENTER is not required to clear machine check error status bits before</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invoking these GETSEC leaves. If this bit returns 0, the caller of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENTERACCS and SENTER must clear all machine check error status bits before</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invoking these GETSEC leaves.</td>
</tr>
<tr>
<td>31:7</td>
<td>Reserved</td>
<td>Reserved for future use. Will return 0.</td>
</tr>
</tbody>
</table>

Supported AC module versions (as defined by the AC module HeaderVersion field) can be determined for a particular SMX capable processor by the type 1 parameter. Using EBX to index through the available parameters reported by GETSEC[PARAMETERS] for each unique parameter set returned for type 1, software can determine the complete list of AC module version(s) supported.

For each parameter set, EBX returns the comparison mask and ECX returns the available HeaderVersion field values supported, after AND'ing the target HeaderVersion with the comparison mask. Software can then determine if a particular AC module version is supported by following the pseudo-code search routine given below:

```
parameter_search_index = 0

while (EAX[4:0] ≠ 0)
{
    EBX = parameter_search_index++
    EAX = 6
    GETSEC
    if (EAX[4:0] = 1) {
        if (((version_query & EBX) = ECX) {  
            version_is_supported = 1
            break
        }
    }
}
```

If only AC modules with a HeaderVersion of 0 are supported by the processor, then only one parameter set of type 1 will be returned, as follows: EAX = 00000001H, EBX = FFFFFFFFH and ECX = 00000000H.

The maximum capacity for an authenticated code execution area supported by the processor is reported with the parameter type of 2. The maximum supported size in bytes is determined by multiplying the returned size in EAX[31:5] by 32. Thus, for a maximum supported authenticated RAM size of 32KBytes, EAX returns with 00008002H.

Supportable memory types for memory mapped outside of the authenticated code execution area are reported with the parameter type of 3. While is active, as initiated by the GETSEC functions SENTER and ENTERACCS and terminated by EXITAC, there are restrictions on what memory types are allowed for the rest of system memory. It is the responsibility of the system software to initialize the memory type range register (MTRR) MSRs and/or the page attribute table (PAT) to only map memory types consistent with the reporting of this parameter. The reporting of supportable memory types of external memory is indicated using a bit map returned in EAX[31:8]. These bit positions correspond to the memory type encodings defined for the MTRR MSR and PAT programming. See Table 6-9.

The parameter type of 4 is used for enumerating the availability of selective GETSEC[SENTER] function disable controls. If a 1 is reported in bits 14:8 of the returned parameter EAX, then this indicates a disable control capa-
bility exists with SENTER for a particular function. The enumerated field in bits 14:8 corresponds to use of the EDX input parameter bits 6:0 for SENTER. If an enumerated field bit is set to 1, then the corresponding EDX input parameter bit of EDX may be set to 1 to disable that designated function. If the enumerated field bit is 0 or this parameter is not reported, then no disable capability exists with the corresponding EDX input parameter for SENTER, and EDX bit(s) must be cleared to 0 to enable execution of SENTER. If no selective disable capability for SENTER exists as enumerated, then the corresponding bits in the IA32_FEATURE_CONTROL MSR bits 14:8 must also be programmed to 1 if the SENTER global enable bit 15 of the MSR is set. This is required to enable future extensibility of SENTER selective disable capability with respect to potentially separate software initialization of the MSR.

Table 6-9. External Memory Types Using Parameter 3

<table>
<thead>
<tr>
<th>EAX Bit Position</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Uncacheable (UC)</td>
</tr>
<tr>
<td>9</td>
<td>Write Combining (WC)</td>
</tr>
<tr>
<td>11:10</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>Write-through (WT)</td>
</tr>
<tr>
<td>13</td>
<td>Write-protected (WP)</td>
</tr>
<tr>
<td>14</td>
<td>Write-back (WB)</td>
</tr>
<tr>
<td>31:15</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

If the GETSEC[PARAMETERS] leaf or specific parameter is not present for a given SMX capable processor, then default parameter values should be assumed. These are defined in Table 6-10.

Table 6-10. Default Parameter Values

<table>
<thead>
<tr>
<th>Parameter Type EAX[4:0]</th>
<th>Default Setting</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0 only</td>
<td>Supported AC module versions.</td>
</tr>
<tr>
<td>2</td>
<td>32 KBytes</td>
<td>Authenticated code execution area size.</td>
</tr>
<tr>
<td>3</td>
<td>UC only</td>
<td>External memory types supported during AC execution mode.</td>
</tr>
<tr>
<td>4</td>
<td>None</td>
<td>Available SENTER selective disable controls.</td>
</tr>
</tbody>
</table>

Operation
(* example of a processor supporting only a 0.0 HeaderVersion, 32K ACRAM size, memory types UC and WC *)

IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
    (* example of a processor supporting a 0.0 HeaderVersion *)
IF (EBX=0)
    THEN
        EAX← 00000001h;
        EBX← FFFFFFFFh;
        ECX← 00000000h;
ELSE IF (EBX=1)
    (* example of a processor supporting a 32K ACRAM size *)
THEN EAX ← 00008002h;
ESE IF (EBX = 2)
   (* example of a processor supporting external memory types of UC and WC *)
   THEN EAX ← 00000303h;
ESE IF (EBX = other value(s) less than unsupported index value)
   (* EAX value varies. Consult Table 6-7 and Table 6-8*)
ELSE (* unsupported index*)
   EAX ← 00000000h;
END;

**Flags Affected**
None.

**Use of Prefixes**
- **LOCK** Causes #UD.
- **REP** Causes #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
- **Operand size** Causes #UD.
- **Segment overrides** Ignored.
- **Address size** Ignored.
- **REX** Ignored.

**Protected Mode Exceptions**
- **#UD** If CR4.SMXE = 0.
  If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

**Real-Address Mode Exceptions**
- **#UD** If CR4.SMXE = 0.
  If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

**Virtual-8086 Mode Exceptions**
- **#UD** If CR4.SMXE = 0.
  If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

**Compatibility Mode Exceptions**
All protected mode exceptions apply.

**64-Bit Mode Exceptions**
All protected mode exceptions apply.

**VM-Exit Condition**
- **Reason (GETSEC)** IF in VMX non-root operation.
GETSEC[SMCTRL]—SMX Mode Control

Description

The GETSEC[SMCTRL] instruction is available for performing certain SMX specific mode control operations. The operation to be performed is selected through the input register EBX. Currently only an input value in EBX of 0 is supported. All other EBX settings will result in the signaling of a general protection violation.

If EBX is set to 0, then the SMCTRL leaf is used to re-enable SMI events. SMI is masked by the ILP executing the GETSEC[SENTER] instruction (SMI is also masked in the responding logical processors in response to SENTER rendezvous messages.). The determination of when this instruction is allowed and the events that are unmasked is dependent on the processor context (See Table 6-11). For brevity, the usage of SMCTRL where EBX=0 will be referred to as GETSEC[SMCTRL(0)].

As part of support for launching a measured environment, the SMI, NMI and INIT events are masked after GETSEC[SENTER], and remain masked after exiting authenticated execution mode. Unmasking these events should be accompanied by securely enabling these event handlers. These security concerns can be addressed in VMX operation by a MVMM.

The VM monitor can choose two approaches:

• In a dual monitor approach, the executive software will set up an SMM monitor in parallel to the executive VMM (i.e. the MVMM), see Chapter 34, “System Management Mode” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C. The SMM monitor is dedicated to handling SMI events without compromising the security of the MVMM. This usage model of handling SMI while a measured environment is active does not require the use of GETSEC[SMCTRL(0)] as event re-enabling after the VMX environment launch is handled implicitly and through separate VMX based controls.

• If a dedicated SMM monitor will not be established and SMIs are to be handled within the measured environment, then GETSEC[SMCTRL(0)] can be used by the executive software to re-enable SMI that has been masked as a result of SENTER.

Table 6-11 defines the processor context in which GETSEC[SMCTRL(0)] can be used and which events will be unmasked. Note that the events that are unmasked are dependent upon the currently operating processor context.

<table>
<thead>
<tr>
<th>ILP Mode of Operation</th>
<th>SMCTRL execution action</th>
</tr>
</thead>
<tbody>
<tr>
<td>In VMX non-root operation</td>
<td>VM exit</td>
</tr>
<tr>
<td>SENTERFLAG = 0</td>
<td>#GP(0), illegal context</td>
</tr>
<tr>
<td>In authenticated code execution mode (ACMODEFLAG = 1)</td>
<td>#GP(0), illegal context</td>
</tr>
<tr>
<td>SENTERFLAG = 1, not in VMX operation, not in SMM</td>
<td>Unmask SMI</td>
</tr>
<tr>
<td>SENTERFLAG = 1, in VMX root operation, not in SMM</td>
<td>Unmask SMI if SMM monitor is not configured, otherwise #GP(0)</td>
</tr>
<tr>
<td>SENTERFLAG = 1, in VMX root operation, in SMM</td>
<td>#GP(0), illegal context</td>
</tr>
</tbody>
</table>
Operation
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason=“GETSEC instruction”);
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) OR (EFLAGS.VM=1))
    THEN #GP(0);
ELSE IF((EBX=0) and (SENDERFLAG=1) and (ACMODEFLAG=0) and (IN_SMM=0) and
    (((in VMX root operation) and (SMM monitor not configured)) or (not in VMX operation))
    THEN unmask SMI;
ELSE
    #GP(0);
END

Flags Affected
None.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.
    If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
    If in VMX root operation.
    If a protected partition is not already active or the processor is currently in authenticated code
    mode.
    If the processor is in SMM.
    If the SMM monitor is not configured.

Real-Address Mode Exceptions
#UD If CR4.SMXE = 0.
    If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SMCTRL] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.
    If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SMCTRL] is not recognized in virtual-8086 mode.
**Compatibility Mode Exceptions**
All protected mode exceptions apply.

**64-Bit Mode Exceptions**
All protected mode exceptions apply.

**VM-exit Condition**
Reason (GETSEC) IF in VMX non-root operation.
GETSEC[_WAKEUP]—Wake up sleeping processors in measured environment

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 37</td>
<td>GETSEC[_WAKEUP]</td>
<td>Wake up the responding logical processors from the SENTER sleep state. (EAX=8)</td>
</tr>
</tbody>
</table>

**Description**

The GETSEC[_WAKEUP] leaf function broadcasts a wake-up message to all logical processors currently in the SENTER sleep state. This GETSEC leaf must be executed only by the ILP, in order to wake-up the RLPs. Responding logical processors (RLPs) enter the SENTER sleep state after completion of the SENTER rendezvous sequence.

The GETSEC[_WAKEUP] instruction may only be executed:

- In a measured environment as initiated by execution of GETSEC[SENTER].
- Outside of authenticated code execution mode.
- Execution is not allowed unless the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0.
- In addition, the logical processor must be designated as the boot-strap processor as configured by setting IA32_APIC_BASE.BSP = 1.

If these conditions are not met, attempts to execute GETSEC[_WAKEUP] result in a general protection violation.

An RLP exits the SENTER sleep state and start execution in response to a WAKEUP signal initiated by ILP’s execution of GETSEC[_WAKEUP]. The RLP retrieves a pointer to a data structure that contains information to enable execution from a defined entry point. This data structure is located using a physical address held in the Intel® TXT-capable chipset configuration register LT.MLE.JOIN. The register is publicly writable in the chipset by all processors and is not restricted by the Intel® TXT-capable chipset configuration register lock status. The format of this data structure is defined in Table 6-12.

**Table 6-12. RLP MVMM JOIN Data Structure**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GDT limit</td>
</tr>
<tr>
<td>4</td>
<td>GDT base pointer</td>
</tr>
<tr>
<td>8</td>
<td>Segment selector initializer</td>
</tr>
<tr>
<td>12</td>
<td>EIP</td>
</tr>
</tbody>
</table>

The MLE JOIN data structure contains the information necessary to initialize RLP processor state and permit the processor to join the measured environment. The GDTR, LIP, and CS, DS, SS, and ES selector values are initialized using this data structure. The CS selector index is derived directly from the segment selector initializer field; DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE = 0, LIMIT = FFFFFH, G = 1, D = 1, P = 1, S = 1; read/write/access for DS, SS, and ES; and execute/read/access for CS. It is the responsibility of external software to establish a GDT pointed to by the MLE JOIN data structure that contains descriptor entries consistent with the implicit settings initialized by the processor (see Table 6-12). Certain states from the content of Table 6-12 are checked for consistency by the processor prior to execution. A failure of any consistency check results in the RLP aborting entry into the protected environment and signaling an Intel® TXT shutdown condition. The specific checks performed are documented later in this section. After successful completion of processor consistency checks and subsequent initialization, RLP execution in the measured environment begins from the entry point at offset 12 (as indicated in Table 6-12).
Operation
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=0) or (in VMX operation) or
        (IA32_APIC_BASE.BSP=0) or (TXT chipset not present))
    THEN #GP(0);
ELSE
    SignalTXTMsg(WAKEUP);
END;

RLP_SIPI_WAKEUP_FROM_SENTER_ROUTINE: (RLP only)
WHILE (no SignalWAKEUP event);
IF (IA32_SMM_MONITOR_CTL[0] ≠ ILP.IA32_SMM_MONITOR_CTL[0])
    THEN TXT-SHUTDOWN(#IllegalEvent)
IF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ELSE
    Mask SMI pin event;
    Mask A20M, and NMI external pin events (unmask INIT);
    Mask SignalWAKEUP event;
    Invalidate processor TLB(s);
    Drain outgoing transactions;
    TempGDTRLIMIT← LOAD(LT.MLE.JOIN);
    TempGDTRBASE← LOAD(LT.MLE.JOIN+4);
    TempSegSel← LOAD(LT.MLE.JOIN+8);
    TempEIP← LOAD(LT.MLE.JOIN+12);
    IF (TempGDTLimit & FFFF0000h)
        THEN TXT-SHUTDOWN(#BadJOINFormat);
    IF ((TempSegSel > TempGDTRLIMIT-15) or (TempSegSel < 8))
        THEN TXT-SHUTDOWN(#BadJOINFormat);
    IF ((TempSegSel.TI=1) or (TempSegSel.RPL ≠ 0))
        THEN TXT-SHUTDOWN(#BadJOINFormat);
    CR0.[PG,CD,NW,AM,WP]← 0;
    CR0.[NE,PE]← 1;
    CR4← 00000400h;
    EFLAGS← 00000002h;
    IA32_EFER← 0;
    GDTR.BASE← TempGDTRBASE;
    GDTR.LIMIT← TempGDTRLIMIT;
    CS.SEL← TempSegSel;
    CS.BASE← 0;
    CS.LIMIT← FFFFh;
    CS.G← 1;
    CS.D← 1;
    CS.AR← 9Bh;
    DS.SEL← TempSegSel+8;
    DS.BASE← 0;
    DS.LIMIT← FFFFh;
    DS.G← 1;
DS.D ← 1;
DS.AR ← 93h;
SS ← DS;
ES ← DS;
DR7 ← 00000400h;
IA32_DEBUGCTL ← 0;
EIP ← TempEIP;
END;

Flags Affected
None.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
If in VMX operation.
If a protected partition is not already active or the processor is currently in authenticated code mode.
If the processor is in SMM.
#UD If CR4.SMXE = 0.
If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[WAKEUP] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[WAKEUP] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
All protected mode exceptions apply.

64-Bit Mode Exceptions
All protected mode exceptions apply.

VM-exit Condition
Reason (GETSEC) IF in VMX non-root operation.
This chapter describes the instruction set that is unique to Intel® Xeon Phi™ Processors based on the Knights Landing and Knights Mill microarchitectures. The set is not supported in any other Intel processors. Included are Intel® AVX-512 instructions. For additional instructions supported on these processors, see Chapter 3, "Instruction Set Reference, A-L", Chapter 4, "Instruction Set Reference, M-U", and Chapter 5, "Instruction Set Reference, V-Z".
PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint

### Description
Fetched the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into ‘Exclusive’ state via a request for ownership) and a locality hint:

- T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHh instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHh instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHh instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHh instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by ‘mem’ into the cache level specified by ‘Level’; a request for exclusive/ownership is done if ‘State’ is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

Prefetch (m8, Level = 1, EXCLUSIVE=1);

### Flags Affected
All flags are affected

### C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch( char const *, int hint= _MM_HINT_ET1);

### Protected Mode Exceptions

#UD If the LOCK prefix is used.
Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
**V4FMADDPS/V4FNMADDPS — Packed Single-Precision Floating-Point Fused Multiply-Add (4-iterations)**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.512.F2.F38.W0 9A /r V4FMADDPS zmm1[k1]{z}, zmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.F2.F38.W0 AA /r V4FNMADDPS zmm1[k1]{z}, zmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply and negate packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1_4X</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction computes 4 sequential packed fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any of the 16 lowest significant mask bits is set to 1 or if a “no masking” encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA (fused multiply and add) boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.
Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

define NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest ← dest

    // reg[] is an array representing the SIMD register file.
    FOR j ← 0 to regs_loaded-1:
        FOR i ← 0 to kl-1:
            IF k1[i] or *no writemask*:
                IF posneg = 0:
                    tmpdest.single[i] ← RoundFPControl_MXCSR(tmpdest.single[i] - reg[src_base + j].single[i] * msrc.single[j])
                ELSE:
                    tmpdest.single[i] ← RoundFPControl_MXCSR(tmpdest.single[i] + reg[src_base + j].single[i] * msrc.single[j])
            ELSE IF *zeroing*:
                tmpdest.single[i] ← 0
    dest ← tmpdest
dest[MAX_VL-1:VL] ← 0

V4FMADDPS and V4FNMADDPS dest{k1}, src1, msrc (AVX512)
KL, VL = (16,512)
regs_loaded ← 4
src_base ← src_reg_id & ~3 // for src1 operand
posneg ← 0 if negative form, 1 otherwise
NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg)

Intel C/C++ Compiler Intrinsic Equivalent

V4FMADDPS __m512 _mm512_4fmadd_ps(__m512, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_mask_4fmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_maskz_4fmadd_ps(__mmask16, __m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_4fnmadd_ps(__m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_mask_4fnmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_maskz_4fnmadd_ps(__mmask16, __m512, __m512x4, __m128 *);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Type E2; additionally

#UD If the EVEX broadcast bit is set to 1.
#UD If the MODRM.mod = 0b11.
### V4FMADDSS/V4FNMADDSS —Scalar Single-Precision Floating-Point Fused Multiply-Add (4-iterations)

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.LLIG.F2.0F38.W0 9B /r V4FMADDSS xmm1[k1][z], xmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LLIG.F2.0F38.W0 AB /r V4FNMADDSS xmm1[k1][z], xmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply and negate scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1_4X</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description
This instruction computes 4 sequential scalar fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if the least significant mask bit is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

### Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```plaintext
define NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest ← dest
    // reg[] is an array representing the SIMD register file.
    IF k1[0] or *no writemask*:
        FOR j ← 0 to regs_loaded - 1:
            IF posneg = 0:
                tmpdest.single[0] ← RoundFPControl_MXCSR(tmpdest.single[0] - reg[src_base + j].single[0] * msrc.single[j])
            ELSE:
                tmpdest.single[0] ← RoundFPControl_MXCSR(tmpdest.single[0] + reg[src_base + j].single[0] * msrc.single[j])
        ELSE IF *zeroing*:
            tmpdest.single[0] ← 0
            dest ← tmpdst
            dest[MAX_VL-1:VL] ← 0
```

---

7-6 Vol. 2D V4FMADDSS/V4FNMADDSS —Scalar Single-Precision Floating-Point Fused Multiply-Add (4-iterations)
V4FMADDSS and V4FN MADDSS dest{k1}, src1, m src (AVX512)
VL = 128

regs_loaded ← 4
src_base ← src_reg_id & ~3 // for src1 operand
posneg ← 0 if negative form, 1 otherwise
NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg)

Intel C/C++ Compiler Intrinsic Equivalent
V4FMADDSS __m128 __m4fmadd_ss(__m128, __m128x4, __m128 *);
V4FMADDSS __m128 __m_mask_4fmadd ss(__m128, __mmask8, __m128x4, __m128 *);
V4FMADDSS __m128 __m_maskz_4fmadd_ss(__mmask8, __m128, __m128x4, __m128 *);
V4FN MADDSS __m128 __m4fnmadd ss(__m128, __m128x4, __m128 *);
V4FN MADDSS __m128 __m_mask_4fnmadd ss(__m128, __mmask8, __m128x4, __m128 *);
V4FN MADDSS __m128 __m_maskz_4fnmadd ss(__mmask8, __m128, __m128x4, __m128 *);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
See Type E2; additionally
#UD If the EVEX broadcast bit is set to 1.
#UD If the MODRM.mod = 0b11.
VEXP2PD—Approximation to the Exponential $2^x$ of Packed Double-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1 C8 /r VEXP2PD zmm1 (k1)[z], zmm2/m512/m64bcst (sae)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the approximate base-2 exponential evaluation of the double-precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than $2^{-23}$ of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


**Operation**

VEXP2PD

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN DEST[i+63:i] ← EXP2_23_DP(SRC[63:0])

ELSE DEST[i+63:i] ← EXP2_23_DP(SRC[i+63:i])

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged* 

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI;

FI;

ENDFOR;
**Table 6-1. Special Values Behavior**

<table>
<thead>
<tr>
<th>Source Input</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>NaN(src)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>+/-0</td>
<td>1.0f</td>
<td><em>Exact result</em></td>
</tr>
<tr>
<td>-∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>Integral value N</td>
<td>2^ (N)</td>
<td><em>Exact result</em></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXP2PD __m512d__mm512_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d__mm512_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d__mm512_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.
VEXP2PS—Approximation to the Exponential $2^x$ of Packed Single-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEXP2PS zmm1 {k1}[z], zmm2/m512/m32bcst (sae)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:r/m (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than $2^{-23}$ of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


**Operation**

VEXP2PS

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

i ← j * 32

IF k1[j] OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN DEST[i+31:i] ← EXP2_23_SP(SRC[31:0])

ELSE DEST[i+31:i] ← EXP2_23_SP(SRC[i+31:i])

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;
Table 6-2. Special Values Behavior

<table>
<thead>
<tr>
<th>Source Input</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(src)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>+\infty</td>
<td>+\infty</td>
<td></td>
</tr>
<tr>
<td>+/-0</td>
<td>1.0f</td>
<td>Exact result</td>
</tr>
<tr>
<td>-\infty</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>Integral value N</td>
<td>$2^{(N)}$</td>
<td>Exact result</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PS __m512 __m512_exp2a23_round_ps (__m512 a, int sae);
VEXP2PS __m512 __mm512_mask_exp2a23_round_ps (__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS __m512 __mm512_maskz_exp2a23_round_ps (__mmask16 m, __m512 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.
VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into a location in the cache hierarchy specified by a locality hint (T0):

- **T0** (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by ‘mem’ into the cache level specified by ‘Level’; a request for exclusive/ownership is done if ‘State’ is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
VGATHERPF0DPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

VGATHERPF0DPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

VGATHERPF0QPS (EVEX encoded version)
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

VGATHERPF0QPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent
VGATHERPF0DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF0QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12NP.
VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

- **T1** (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
VGATHERPF1DPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR

VGATHERPF1DPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR

VGATHERPF1QPS (EVEX encoded version)
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR

VGATHERPF1QPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent
VGATHERPF1DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void _mm512_mask_prefetch_i32gather_ps(__m2512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12NP.
VP4DPwSSDS — Dot Product of Signed Words with Dword Accumulation and Saturation (4-iterations)

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.512.F2.0F38.W0 53 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4VNNIW</td>
<td>Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate the resulting dword results with signed saturation in zmm1.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1_4X</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation and signed saturation. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a “no masking” encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
VP4DPwSSDS dest, src1, src2
(KL,VL) = (16,512)
N ← 4

ORIGDEST ← DEST
src_base ← src_reg_id & ~ (N-1) // for src1 operand

FOR i ← 0 to KL-1:
  IF k1[i] or *no writemask*:
    FOR m ← 0 to N-1:
      t ← SRC2.dword[m]
      p1dword ← reg[src_base+m].word[2*i] * t.word[0]
      p2dword ← reg[src_base+m].word[2*i+1] * t.word[1]
      DEST.dword[i] ← SIGNED_DWORD_SATURATE(DEST.dword[i] + p1dword + p2dword)
  ELSE IF *zeroing*:
    DEST.dword[i] ← 0
  ELSE
    DEST.dword[i] ← ORIGDEST.dword[i]
    DEST[MAX_VL-1:VL] ← 0
```
**Intel C/C++ Compiler Intrinsic Equivalent**

VP4DPWSSDS __m512i _mm512__4dpwssds_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_mask_4dpwssds_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_maskz_4dpwssds_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Type E4; additionally

#UD If the EVEX broadcast bit is set to 1.
#UD If the MODRM.mod = 0b11.
VP4DPWSSD — Dot Product of Signed Words with Dword Accumulation (4-iterations)

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.512.F2.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4VNNIW</td>
<td>Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate resulting signed dwords in zmm1.</td>
</tr>
<tr>
<td>VP4DPwSSD zmm1[k1][z], zmm2+3, m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1_4X</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation; see Figure 7-1 below. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a “no masking” encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

![Figure 7-1. Register Source-Block Dot Product of Two Signed Word Operands with Doubleword Accumulation](image)

**NOTES:**

1. For illustration purposes, one source-block dot product instance is shown out of the four.
**Operation**

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
VP4DPWSSD dest, src1, src2
(KL,VL) = (16,512)
N ← 4
```

ORIGDEST ← DEST
src_base ← src_reg_id & ~ (N-1) // for src1 operand

FOR i ← 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m ← 0 to N-1:
            t ← SRC2.dword[m]
            p1dword ← reg[src_base+m].word[2*i] * t.word[0]
            p2dword ← reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] ← DEST.dword[i] + p1dword + p2dword
        ELSE IF *zeroing*:
            DEST.dword[i] ← 0
        ELSE
            DEST.dword[i] ← ORIGDEST.dword[i]
    DEST[MAX_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VP4DPWSSD __m512i _mm512_4dpwssd_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_mask_4dpwssd_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_maskz_4dpwssd_epi32(__mmask16, __m512i, __m512ix4, __m128i *);
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Type E4; additionally

- #UD If the EVEX broadcast bit is set to 1.
- #UD If the MODRM.mod = 0b11.
VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1 CA /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocals ($&lt; 2^{-28}$ relative error) of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

Usage: VRCP28PD

```
VRCP28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+63:j] ← RCP_28_DP(1.0/SRC[i+63:j]);
            ELSE DEST[i+63:j] ← RCP_28_DP(1.0/SRC[i+63:j]);
            Fl;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:j] remains unchanged*;
                ELSE ; zeroing-masking
                    DEST[i+63:j] ← 0
                    Fl;
            Fl;
        ENDFOR;
```

Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than 2^{-28} Relative Error

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28PD __m512d_mm512_rcp28_round_pd ( __m512d a, int sae);
VRCP28PD __m512d_mm512_mask_rcp28_round_pd( __m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d_mm512_maskz_rcp28_round_pd( __mmask8 m, __m512d b, int sae);

**Intel C/C++ Compiler Intrinsic Equivalent**

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

---

**Table 6-3. VRCP28PD Special Cases**

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2^{-1022}</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2^{-1022} ≤ X ≤ -0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2^{1022}</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2^{1022}</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2^{-n}</td>
<td>2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2^{-n}</td>
<td>-2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>
VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 CB /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocal ($&lt; 2^{-28}$ relative error) of the scalar double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is $\pm\infty$, $\pm0.0$ is returned for that element. Also, if any source element is $\pm0.0$, $\pm\infty$ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.


**Operation**

**VRCP28SD (EVEX encoded versions)**

IF k1[0] OR *no writemask* THEN

DEST[63:0] ← RCP_28_DP(1.0/SRC2[63:0]);
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*;
ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;
ENDFOR;

DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
Table 6-4. VRCP28SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2^-1022</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2^-1022 ≤ X ≤ 0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2^1022</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2^1022</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2^-n</td>
<td>2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2^-n</td>
<td>-2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SD __m128d _mm_rcp28_round_sd (__m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.
VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 [k1][z], zmm2/m512/m32bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocals (&lt; 2^-28 relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results are rounded to < 2^-23 relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1. EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


### Operation

**VRCP28PS (EVEX encoded versions)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] ← RCP_28_SP(1.0/SRC[31:0]);
      ELSE DEST[i+31:i] ← RCP_28_SP(1.0/SRC[i+31:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*;
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI;
  FI;
ENDFOR;
Table 6-5. VRCP28PS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2⁻¹²⁶</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2⁻¹²⁶ &lt; X ≤ 0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2¹²⁶</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2¹²⁶</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2⁻ⁿ</td>
<td>2ⁿ</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2⁻ⁿ</td>
<td>-2ⁿ</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PS_mm512_rcp28_round_ps ( __m512 a, int sae);
VRCP28PS__m512__mm512_mask_rcp28_round_ps( __m512 s, __mmask16 m, __m512 a, int sae);
VRCP28PS__m512__mm512_maskz_rcp28_round_ps( __mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.
VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W0 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocal ($&lt; 2^{-28}$ relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final result is rounded to $< 2^{-23}$ relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is $\pm\infty$, $\pm0.0$ is returned for that element. Also, if any source element is $\pm0.0$, $\pm\infty$ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.


**Operation**

**VRCP28SS ((EVEX encoded versions))**

IF k1[0] OR *no writemask* THEN

DEST[31:0] ← RCP_28_SP(1.0/SRC2[31:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged* 
ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

Fi;

ENDFOR;


DEST[MAXVL-1:128] ← 0
Table 6-6. VRCP28SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2^{-126}</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2^{-126} ≤ X ≤ -0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2^{126}</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2^{126}</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2^{-n}</td>
<td>2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2^{-n}</td>
<td>-2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SS __m128 __mm_rcp28_round_ss (__m128 a, __m128 b, int sae);
VRCP28SS __m128 __mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128 __mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.
VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the Reciprocal square root ($&lt;2^{-28}$ relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of $-0$ must return $-\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $-\infty$ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


Operation

VRSQRT28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

FOR $j \leftarrow 0$ TO KL-1

$\begin{align*}
  i & \leftarrow j \times 64 \\
  \text{IF } & k1[j] \text{ OR } \text{*no writemask* THEN} \\
  \text{IF (EVEX.b = 1) AND (SRC *is memory*)} \\
  \text{THEN } & \text{DEST}[i+63:j] \leftarrow (1.0 / \text{SQRT(SRC[63:0]})); \\
  \text{ELSE } & \text{DEST}[i+63:j] \leftarrow (1.0 / \text{SQRT(SRC[i+63:j]})); \\
  \text{FI;} \\
  \text{ELSE} \\
  \text{IF *merging-masking*} & ; \text{merging-masking} \\
  \text{THEN } & \text{*DEST}[i+63:j] \text{remains unchanged*} \\
  \text{ELSE} & ; \text{zeroing-masking} \\
  \text{DEST}[i+63:j] & \leftarrow 0 \\
  \text{FI;} \\
  \text{FI;} \\
\end{align*}$

ENDFOR;
Table 6-7. VRSQRT28PD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$ or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +0$ or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28PD __m512d __m512_rsqrt28_round_pd(__m512d a, int sae);
VRSQRT28PD __m512d __mm512_rsqrt28_round_pd(__m512d s, __mmask8 m, __m512d a, int sae);
VRSQRT28PD __m512d __mm512_maskz_rsqrt28_round_pd(__mmask8 m, __m512d a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.
VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LG.66.0F38.W1 CD /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximate reciprocal square root ($&lt;2^{-28}$ relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1 with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than $2^{-28}$ of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of $-0$ must return $-\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $-\infty$ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.


### Operation

**VRSQRT28SD (EVEX encoded versions)**

IF $k1[0]$ OR “no writemask” THEN

DEST[63: 0] ← $1.0/ \text{SQRT}(\text{SRC}[63: 0])$;

ELSE

IF “merging-masking” ; merging-masking

THEN *DEST[63: 0] remains unchanged*

ELSE ; zeroing-masking

DEST[63: 0] ← 0

FI;

FI;

ENDIF;

DEST[127:64] ← SRC1[127: 64]

DEST[MAXVL-1:128] ← 0
Table 6-8. VRSQRT28SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNaN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$ or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +0$ or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28SD __m128d __m128d_mm_rsqrt28_round_sd(__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d __m128d_mm_mask_rsqrt28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d __m128d_mm_maskz_rsqrt28_round_sd(__mmask8 m, __m128d a, __m128d b, int rounding);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.
VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 CC /r</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the Reciprocal square root (&lt;2^-28 relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results is rounded to < 2^-23 relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


**Operation**

VRSQRT28PS (EVEX encoded versions)

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

i ← j * 32

IF k1[j] OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN DEST[i+31:i] ← (1.0 / SQRT(SRC[31:0]));

ELSE DEST[i+31:i] ← (1.0 / SQRT(SRC[i+31:i]));

FI;

ELSE

IF *merging-masking*

THEN *DEST[i+31:i] remains unchanged*

ELSE

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;
Table 6-9. VRSQRT28PS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>X = 2⁻²ⁿ</td>
<td>2ⁿ</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0 or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +0 or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PS __m512 __mm512_rsqrt28_round_ps(__m512 a, int sae);
VRSQRT28PS __m512 __mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRSQRT28PS __m512 __mm512_maskz_rsqrt28_round_ps(__mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.
VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LG.66.0F38.W0 CD /r VRSQRT28SS xmm1 (k1){z}, xmm2, xmm3/m32 {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximate reciprocal square root ($&lt;2^{-28}$ relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1 with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final result is rounded to $<2^{-23}$ relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return $-\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $-\infty$ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.


Operation

VRSQRT28SS (EVEX encoded versions)

IF k1[0] OR *no writemask* THEN

DEST[31: 0] ← (1.0/ SQRT(SRC[31: 0]));
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31: 0] remains unchanged*
ELSE ; zeroing-masking

DEST[31: 0] ← 0

Fl;

ENDFOR;
DEST[MAXVL-1:128] ← 0
Table 6-10. VRSQRT28SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$X = 2^{-2^n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$ or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +0$ or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int rounding);
VRSQRT28SS __m128 _mm_mask_rsqrt28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int rounding);
VRSQRT28SS __m128 _mm_maskz_rsqrt28_round_ss(__mmask8 m, __m128 a, __m128 b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.
VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C6 /5 /vsib VSCATTERPF0DPS vm32z (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C7 /5 /vsib VSCATTERPF0QPS vm64z (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C6 /5 /vsib VSCATTERPF0DPD vm32y (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C7 /5 /vsib VSCATTERPF0QPD vm64z (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg (R): VSIBbase, VectorReg(R): VSIBindex</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.
- [PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.
- [PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH('mem', Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
VSCATTERPF0DPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR

VSCATTERPF0DPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR

VSCATTERPF0QPS (EVEX encoded version)
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR

VSCATTERPF0QPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent
VSCATTERPF0DPS void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i32scatter_pd(void *base, __m128i m, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask_prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_mask_prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12NP.
The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by ‘mem’ into the cache level specified by ‘Level’; a request for exclusive/ownership is done if ‘State’ is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
VSCATTERPF1DPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR J ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR

VSCATTERPF1DPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR J ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR

VSCATTERPF1QPS (EVEX encoded version)
(KL, VL) = (8, 512)
FOR J ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR

VSCATTERPF1QPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR J ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent
VSCATTERPF1DPS void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_mask prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_mask prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12NP.
Use the opcode tables in this chapter to interpret IA-32 and Intel 64 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

**NOTE**

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

### A.1 Using Opcode Tables

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined. Cells marked "Reserved-NOP" are also reserved but may behave as NOP on certain processors. Software should not use opcodes corresponding blank cells or cells marked "Reserved-NOP" nor depend on the current behavior of those opcodes.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, "Opcode Look-up Examples for One, Two, and Three-Byte Opcodes."

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

### A.2 Key to Abbreviations

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

#### A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

- **A** Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- **B** The VEX.vvvv field of the VEX prefix selects a general purpose register.
A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).
b Byte, regardless of operand-size attribute.
c Byte or word, depending on operand-size attribute.
d Doubleword, regardless of operand-size attribute.
dq  Double-quadword, regardless of operand-size attribute.
p  32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.
pd 128-bit or 256-bit packed double-precision floating-point data.
pi Quadword MMX technology register (for example: mm0).
ps 128-bit or 256-bit packed single-precision floating-point data.
q  Quadword, regardless of operand-size attribute.
qq Quad-Quadword (256-bits), regardless of operand-size attribute.
s  6-byte or 10-byte pseudo-descriptor.
sd Scalar element of a 128-bit double-precision floating data.
ss Scalar element of a 128-bit single-precision floating data.
si Doubleword integer register (for example: eax).
v  Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
w  Word, regardless of operand-size attribute.
x dq or qq based on the operand-size attribute.
y Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
z Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

A.2.3  Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

A.2.4  Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

A.2.4.1  One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section A.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, “Instruction Format,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. Operand types are listed according to notations listed in Section A.2.
• A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.

• Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSHA; 06H, PUSH ES).

Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

• The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.

• The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.

• The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg(opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem_op, and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.4).

A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

• A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. The operand types are listed according to notations listed in Section A.2.

• A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.

• Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA4050000000003H for a SHLD instruction using Table A-3.

• The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  — Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  — Gv: The reg field of the ModR/M byte selects a general-purpose register.
  — Ib: Immediate data is encoded in the subsequent byte of the instruction.

• The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.

• The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).
• By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:00000000H, EAX, 3.

A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3A. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

• A ModR/M byte is required and is interpreted according to the abbreviations listed in A.1 and Chapter 2, "Instruction Format," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. The operand types are listed according to notations listed in Section A.2.

Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

• 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  — Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
  — Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
  — Ib: Immediate data is encoded in the subsequent byte of the instruction.

• The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.

• The last byte is the immediate byte (08H).

• By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

A.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size/opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a “v”. If the VEX prefix is not present the VEX.vvvv operand is not available and the prefix “v” is dropped from the mnemonic.

A few instructions exist only in VEX form and these are marked with a superscript “v”.

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by ’dq’, 256-bit vectors are indicated by ’qq’, and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by ’x’. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.
A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

### Table A-1. Superscripts Utilized in Opcode Tables

<table>
<thead>
<tr>
<th>Superscript Symbol</th>
<th>Meaning of Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A</td>
<td>Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, “Opcode Extensions For One-Byte And Two-byte Opcodes”).</td>
</tr>
<tr>
<td>1B</td>
<td>Use the 0F0B opcode (UD2 instruction), the 0FB9H opcode (UD1 instruction), or the 0FFFH opcode (UD0 instruction) when deliberately trying to generate an invalid opcode exception (#UD).</td>
</tr>
<tr>
<td>1C</td>
<td>Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.</td>
</tr>
<tr>
<td>i64</td>
<td>The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).</td>
</tr>
<tr>
<td>o64</td>
<td>Instruction is only available when in 64-bit mode.</td>
</tr>
<tr>
<td>d64</td>
<td>When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.</td>
</tr>
<tr>
<td>f64</td>
<td>The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).</td>
</tr>
<tr>
<td>v</td>
<td>VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.</td>
</tr>
<tr>
<td>v1</td>
<td>VEX128 &amp; SSE forms only exist (no VEX256), when can't be inferred from the data size.</td>
</tr>
</tbody>
</table>

A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.
### Table A-2. One-byte Opcode Map: (00H — F7H) *

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td>ADD</td>
<td>EB, GB</td>
<td>EV, Gv</td>
<td>GB, EB</td>
<td>Gv, EV</td>
<td>AL, lb</td>
<td>rAX, lz</td>
</tr>
<tr>
<td><strong>1</strong></td>
<td>ADC</td>
<td>EB, GB</td>
<td>EV, Gv</td>
<td>GB, EB</td>
<td>Gv, EV</td>
<td>AL, lb</td>
<td>rAX, lz</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>AND</td>
<td>EB, GB</td>
<td>EV, Gv</td>
<td>GB, EB</td>
<td>Gv, EV</td>
<td>AL, lb</td>
<td>rAX, lz</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>XOR</td>
<td>EB, GB</td>
<td>EV, Gv</td>
<td>GB, EB</td>
<td>Gv, EV</td>
<td>AL, lb</td>
<td>rAX, lz</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>INC&lt;sup&gt;ES&lt;/sup&gt; general register / REX&lt;sup&gt;ES&lt;/sup&gt; Prefixes</td>
<td>EB, Gb</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>EB, Gb</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, lb</td>
<td>rAX, lz</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>POP&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>ESi64</td>
<td>POP&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>SSi64</td>
<td>POP&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>BND&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>EB, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib</td>
<td>rAX, lz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O</th>
<th>NO</th>
<th>BNAE/C</th>
<th>NB/AE/NC</th>
<th>Z/E</th>
<th>NZ/NE</th>
<th>BE/NA</th>
<th>NBE/A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>8</strong></td>
<td>Immediate Grp 1&lt;sup&gt;EA&lt;/sup&gt;</td>
<td>EB, lb</td>
<td>EV, Iz</td>
<td>EB, lb&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>EV, lb</td>
<td>EB, Gb</td>
<td>EV, Gv</td>
</tr>
<tr>
<td><strong>9</strong></td>
<td>NOP PAUSE(F3)</td>
<td>EB, lb&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>EB, lb&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>EB, lb&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>EB, lb&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>EB, lb&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>EB, lb&lt;sup&gt;ES&lt;/sup&gt;</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>MOV</td>
<td>AL, Ob</td>
<td>rAX, Ox</td>
<td>Ob, AL</td>
<td>Ob, rAX</td>
<td>MOV/S/B</td>
<td>MOV/S/W/D/Q</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>Shift Grp 2&lt;sup&gt;EA&lt;/sup&gt;</td>
<td>EB, lb</td>
<td>EV, lb</td>
<td>near RET&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>near RET&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>LES&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>Gz, Mp</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>Shift Grp 2&lt;sup&gt;EA&lt;/sup&gt;</td>
<td>EB, 1</td>
<td>Ev, 1</td>
<td>EB, CL</td>
<td>Ev, CL</td>
<td>AAM&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>lb</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>LOOPNE&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>LOOPNZ&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>LOOP&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>LOOP&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>LOOP&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>JrcXZ&lt;sup&gt;ES&lt;/sup&gt;</td>
<td>Jb</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>LOCK (Prefix)</td>
<td>INT1</td>
<td>REPNE XACQUIRE (Prefix)</td>
<td>REP/REPE XRELEASE (Prefix)</td>
<td>HLT</td>
<td>CMC</td>
<td>Unary Grp 3&lt;sup&gt;EA&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

---

* * *
### Table A-2. One-byte Opcode Map: (08H — FFH) *

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, EV</td>
<td>AL, LB</td>
<td>rAX, IZ</td>
</tr>
<tr>
<td>1</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, EV</td>
<td>AL, LB</td>
<td>rAX, IZ</td>
</tr>
<tr>
<td>2</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, EV</td>
<td>AL, LB</td>
<td>rAX, IZ</td>
</tr>
<tr>
<td>3</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, EV</td>
<td>AL, LB</td>
<td>rAX, IZ</td>
</tr>
<tr>
<td>4</td>
<td>eAX</td>
<td>eCX</td>
<td>eDX</td>
<td>eBX</td>
<td>eSP</td>
<td>eBP</td>
</tr>
<tr>
<td></td>
<td>REX.W</td>
<td>REX.WB</td>
<td>REX.WX</td>
<td>REX.WB</td>
<td>REX.WR</td>
<td>REX.WR</td>
</tr>
<tr>
<td>5</td>
<td>rAX/8</td>
<td>rCX/9</td>
<td>rDX/10</td>
<td>rBX/11</td>
<td>rSP/12</td>
<td>rBP/13</td>
</tr>
<tr>
<td>6</td>
<td>PUSH&lt;sup&gt;64&lt;/sup&gt;</td>
<td>IMUL</td>
<td>IMUL</td>
<td>INS/</td>
<td>INS/</td>
<td>OUTS/</td>
</tr>
<tr>
<td></td>
<td>IZ</td>
<td>Gv, Ev, IZ</td>
<td>Gv, Ev, IZ</td>
<td>INSB</td>
<td>INSB</td>
<td>OUTSB</td>
</tr>
<tr>
<td>7</td>
<td>S</td>
<td>NS</td>
<td>PIPE</td>
<td>NP/PO</td>
<td>L/NGE</td>
<td>NL/GE</td>
</tr>
<tr>
<td>8</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, EV</td>
<td>MOV</td>
<td>LEA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EV, SW</td>
<td>Gv, M</td>
</tr>
<tr>
<td>9</td>
<td>CBW/</td>
<td>CWD/</td>
<td>far CALL&lt;sup&gt;64&lt;/sup&gt;</td>
<td>FWAIT/</td>
<td>PUSHF/DQ&lt;sup&gt;64&lt;/sup&gt;</td>
<td>POPF/DQ&lt;sup&gt;64&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>CWDE/</td>
<td>CDQ/</td>
<td>Ap</td>
<td>WAIT</td>
<td>Fv</td>
<td>Fv</td>
</tr>
<tr>
<td></td>
<td>CDQE</td>
<td>CQO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>AL, LB</td>
<td>rAX, IZ</td>
<td>STOS/B</td>
<td>STOS/W/D/Q</td>
<td>LODS/B</td>
<td>LODS/W/D/Q</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yb, AL</td>
<td>Yv, rAX</td>
<td>AL, Xb</td>
<td>AL, Xv</td>
</tr>
<tr>
<td>B</td>
<td>rAX/8, IY</td>
<td>rCX/9, IY</td>
<td>rDX/10, IY</td>
<td>rBX/11, IY</td>
<td>rSP/12, IY</td>
<td>rBP/13, IY</td>
</tr>
<tr>
<td>C</td>
<td>ENTER</td>
<td>LEAVE&lt;sup&gt;64&lt;/sup&gt;</td>
<td>far RET</td>
<td>far RET</td>
<td>INT3</td>
<td>INT</td>
</tr>
<tr>
<td></td>
<td>LW, LB</td>
<td></td>
<td>LW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>near CALL&lt;sup&gt;64&lt;/sup&gt;</td>
<td>near&lt;sup&gt;64&lt;/sup&gt;</td>
<td>JMP</td>
<td>JMP</td>
<td>far&lt;sup&gt;64&lt;/sup&gt;</td>
<td>far&lt;sup&gt;64&lt;/sup&gt;</td>
</tr>
<tr>
<td>F</td>
<td>CLC</td>
<td>STC</td>
<td>CLI</td>
<td>STI</td>
<td>CLD</td>
<td>STD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
<table>
<thead>
<tr>
<th>pfx</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grp 6A</td>
<td>Grp 7A</td>
<td>LAR</td>
<td>LSL</td>
<td>SYSCALL</td>
<td>CLTS</td>
<td>SYSRET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vmovups</td>
<td>Wps, Wps</td>
<td>vmovups</td>
<td>Wps, Wps</td>
<td>vunpcklps</td>
<td>Mq, Vq</td>
<td>vunpcklps</td>
<td>Mq, Vq</td>
<td></td>
</tr>
<tr>
<td>vmovpd</td>
<td>Vpd, Wpd</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wx</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wx</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td></td>
</tr>
<tr>
<td>vmovups</td>
<td>Wps, Wps</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td></td>
</tr>
<tr>
<td>.vmov</td>
<td>Vx, Hx, Wp</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td></td>
</tr>
<tr>
<td>vunpcklps</td>
<td>Mq, Vq</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td>vunpcklps</td>
<td>Vq, Hq, Wq</td>
<td></td>
</tr>
</tbody>
</table>

*CMOVcc, (Gv, Ev) - Conditional Move*
Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) *

<table>
<thead>
<tr>
<th>pfx</th>
<th>B</th>
<th>A</th>
<th>2-byte Illegal Opcodes (UD2)</th>
<th>Ev</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INVD</td>
<td>WBINVD</td>
<td>2-byte Illegal Opcodes (UD2)</td>
<td>Ev</td>
</tr>
<tr>
<td>1</td>
<td>Prefetch (Grp 161A)</td>
<td>Reserved-NOP</td>
<td>Reserved-NOP</td>
<td>NOP / Ev</td>
</tr>
<tr>
<td>66</td>
<td>vmovaps Vps, Wps</td>
<td>vmovaps Wps, Vps</td>
<td>vmovntps Mps, Vps</td>
<td>cvtbpzi Ppi, Wps</td>
</tr>
<tr>
<td></td>
<td>vmovapd Vpd, Wpd</td>
<td>vmovapd Wpd, Vpd</td>
<td>cvtbpzip Ppi, Wpd</td>
<td>cvtbpzi Ppi, Wps</td>
</tr>
<tr>
<td></td>
<td>nvtsi2zi Vsd, Hsd, Ey</td>
<td>nvtsi2zi Vsd, Hsd, Ey</td>
<td>nvtsi2zi Gy, Wss</td>
<td>nvtsi2zi Gy, Wss</td>
</tr>
<tr>
<td>3</td>
<td>3-byte escape</td>
<td>3-byte escape</td>
<td>(Table A-4)</td>
<td>(Table A-5)</td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td>NS</td>
<td>PIPE</td>
<td>NP/PO</td>
</tr>
<tr>
<td></td>
<td>vaddss Vss, Hss, Wss</td>
<td>vnuadd Vss, Hss, Wss</td>
<td>vctbts2sd Vsd, Hsd, Ey</td>
<td>vsubss Vss, Hss, Wss</td>
</tr>
<tr>
<td>6</td>
<td>punpckhdq Vq, Qd</td>
<td>punpckhdq Pq, Qd</td>
<td>punpckhhdq Vq, Qd</td>
<td>vpacksdw Pq, Qd</td>
</tr>
<tr>
<td>66</td>
<td>vpunpckhdq Vx, Hx, Wx</td>
<td>vpunpckhdq Vx, Hx, Wx</td>
<td>vpunpckhdq Vx, Hx, Wx</td>
<td>vpunpckhdq Vx, Hx, Wx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>VMREAD Ey, Gy</td>
<td>VMWRITE Gy, Ey</td>
<td>vhaddpd Vpd, Hpd, Wpd</td>
<td>vsubpd Vpd, Hpd, Wpd</td>
</tr>
<tr>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table A-3. Two-byte Opcode Map: 80H — F7H (First Byte is 0FH) *

<table>
<thead>
<tr>
<th>pfx</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>O</td>
<td>NO</td>
<td>B/C/NAE</td>
<td>AE/NB/NC</td>
<td>E/Z</td>
<td>NE/NZ</td>
<td>BE/NA</td>
<td>A/NBE</td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>NO</td>
<td>B/C/NAE</td>
<td>AE/NB/NC</td>
<td>E/Z</td>
<td>NE/NZ</td>
<td>BE/NA</td>
<td>A/NBE</td>
</tr>
<tr>
<td>C</td>
<td>PUSH</td>
<td>FS</td>
<td>POp</td>
<td>FS</td>
<td>CPUID</td>
<td>BT</td>
<td>Ev, Gv</td>
<td>SHLD</td>
</tr>
<tr>
<td>D</td>
<td>CMPXCHG</td>
<td>Eb, Gb</td>
<td>EV</td>
<td>Gv</td>
<td>LSS</td>
<td>Gv, Mp</td>
<td>BTR</td>
<td>Ev, Gv</td>
</tr>
<tr>
<td>E</td>
<td>66</td>
<td>XADD</td>
<td>XADD</td>
<td>Ev, Gv</td>
<td>cmpops</td>
<td>Vps, Hps, Wps, Ib</td>
<td>mvnti</td>
<td>Lss</td>
</tr>
<tr>
<td>F</td>
<td>F3</td>
<td>66</td>
<td>vaddsubpd</td>
<td>Vpd, Hpd, Wpd</td>
<td>vpsrad</td>
<td>Vx, Hx, Wx</td>
<td>vpsadd</td>
<td>Vx, Hx, Wx</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>F3</td>
<td>66</td>
<td>vpsub pd</td>
<td>Vpd, Hpd, Wpd</td>
<td>vpsrad</td>
<td>Vx, Hx, Wx</td>
<td>vpsadd</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>F2</td>
<td>F3</td>
<td>vpavg</td>
<td>Vx, Hx, Wx</td>
<td>vpsrad</td>
<td>Vx, Hx, Wx</td>
<td>vpsadd</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>F2</td>
<td>F2</td>
<td>vpsub ps</td>
<td>Vx, Hx, Wx</td>
<td>vpsrad</td>
<td>Vx, Hx, Wx</td>
<td>vpsadd</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>F2</td>
<td>F3</td>
<td>vpavg</td>
<td>Vx, Hx, Wx</td>
<td>vpsrad</td>
<td>Vx, Hx, Wx</td>
<td>vpsadd</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>F2</td>
<td>F2</td>
<td>vpavg</td>
<td>Vx, Hx, Wx</td>
<td>vpsrad</td>
<td>Vx, Hx, Wx</td>
<td>vpsadd</td>
</tr>
</tbody>
</table>

* Jccff64, Jz - Long-displacement jump on condition
* SETcc, Eb - Byte Set on condition

Grp 91A
**Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH)** *

<table>
<thead>
<tr>
<th>pfx</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
<table>
<thead>
<tr>
<th>pfx</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68 &amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H)**

Vol. 2D A-13
<table>
<thead>
<tr>
<th>pfx</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pabsb</td>
<td>psignb</td>
<td>psignb</td>
<td>psignb</td>
<td>pmulhrsw</td>
<td>vpmaddps</td>
<td>vpshufps</td>
<td>vtestps</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>1</td>
<td>pabsw</td>
<td>psignw</td>
<td>psignw</td>
<td>psignw</td>
<td>pmulhrsw</td>
<td>vpmaddps</td>
<td>vpshufps</td>
<td>vtestps</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>2</td>
<td>vbroadcastss</td>
<td>vbroadcastsd</td>
<td>vbroadcastsf</td>
<td>vbroadcastss</td>
<td>vpackusdw</td>
<td>vmaskmovps</td>
<td>vmaskmovps</td>
<td>vmaskmovps</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wd</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>3</td>
<td>vpminb</td>
<td>vpminb</td>
<td>vpminb</td>
<td>vpminb</td>
<td>vpmaddps</td>
<td>vpmaddps</td>
<td>vpmaddps</td>
<td>vpmaddps</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>4</td>
<td>vpmuldq</td>
<td>vpcmpeqq</td>
<td>vpmull</td>
<td>vpcmpeq</td>
<td>vpmull</td>
<td>vpmull</td>
<td>vpmull</td>
<td>vpmull</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>5</td>
<td>vpminab</td>
<td>vpminab</td>
<td>vpminab</td>
<td>vpminab</td>
<td>vpmaddps</td>
<td>vpmaddps</td>
<td>vpmaddps</td>
<td>vpmaddps</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>6</td>
<td>vpbroadcastbv</td>
<td>vpbroadcastdv</td>
<td>vpbroadcastbw</td>
<td>vpbroadcastbv</td>
<td>vpmuldq</td>
<td>vpmuldq</td>
<td>vpmuldq</td>
<td>vpmuldq</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>7</td>
<td>vpbroadcastbv</td>
<td>vpbroadcastdv</td>
<td>vpbroadcastbw</td>
<td>vpbroadcastbv</td>
<td>vpmuldq</td>
<td>vpmuldq</td>
<td>vpmuldq</td>
<td>vpmuldq</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>8</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
<tr>
<td>9</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
<td>vmmadd132ps</td>
</tr>
<tr>
<td>66</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
<td>Vx, Wx</td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *

<table>
<thead>
<tr>
<th>Pfx</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>66</td>
<td>vpermq⑦ Vqq, Wqq, lb</td>
<td>vpermqd⑦ Vqq, Wqq, lb</td>
<td>vpblendd⑦ Vx, Hx, Wx, lb</td>
<td>vpermilps⑦ Vx, Wx, lb</td>
<td>vpermilpd⑦ Vx, Wx, lb</td>
<td>vperm2f128⑦ Vqq, Hqq, Wqq, lb</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>66</td>
<td>vpextrb Rd/Mb, Vdq, lb</td>
<td>vpextrw Rd/Mw, Vdq, lb</td>
<td>vpextrd/q Ey, Vdq, lb</td>
<td>vextractps Ed, Vdq, lb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>vpinsrb Vdq, Hdq, Rq/Mb, lb</td>
<td>vinsertps Vdq, Hdq, Udq/Md, lb</td>
<td>vpinsrd/q Vdq, Hdq, Ey, lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>66</td>
<td>vdpqs Vx, Hx, Wx, lb</td>
<td>vdpqd Vx, Hdq, Wdq, lb</td>
<td>vmpsb/bw Vx, Hx, Wx, lb</td>
<td>vpcmpu/q Vdq, Hdq, Wdq, lb</td>
<td>vperm2128⑦ Vqq, Hqq, Wqq, lb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>66</td>
<td>vpcmpestrm Vdq, Wdq, lb</td>
<td>vpcmpestri Vdq, Wdq, lb</td>
<td>vpcmpestrm Vdq, Wdq, lb</td>
<td>vpcmpestri Vdq, Wdq, lb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>RORX② Gy, Ey, lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Notes: ⑦ indicates the second byte is 3AH.
### Table A-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH) *

<table>
<thead>
<tr>
<th>pfx</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vroundps Vx,Wx,lb</td>
<td>vroundps Vx,Wx,lb</td>
<td>vroundps Vx,Wx,lb</td>
<td>vroundps Vx,Wx,lb</td>
<td>vroundps Vx,Wx,lb</td>
<td>vroundps Vx,Wx,lb</td>
<td>vroundps Vx,Wx,lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pq, Qq, Ib</td>
<td>Pq, Qq, Ib</td>
<td>Pq, Qq, Ib</td>
<td>Pq, Qq, Ib</td>
<td>Pq, Qq, Ib</td>
<td>Pq, Qq, Ib</td>
<td>Pq, Qq, Ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcvtstps2ph Vx, Vx, Ib</td>
<td>vcvtstps2ph Vx, Vx, Ib</td>
<td>vcvtstps2ph Vx, Vx, Ib</td>
<td>vcvtstps2ph Vx, Vx, Ib</td>
<td>vcvtstps2ph Vx, Vx, Ib</td>
<td>vcvtstps2ph Vx, Vx, Ib</td>
<td>vcvtstps2ph Vx, Vx, Ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vblendps Vx,Hx,Wx,lb</td>
<td>vblendps Vx,Hx,Wx,lb</td>
<td>vblendps Vx,Hx,Wx,lb</td>
<td>vblendps Vx,Hx,Wx,lb</td>
<td>vblendps Vx,Hx,Wx,lb</td>
<td>vblendps Vx,Hx,Wx,lb</td>
<td>vblendps Vx,Hx,Wx,lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vblendpd Vx,Hx,Wx,lb</td>
<td>vblendpd Vx,Hx,Wx,lb</td>
<td>vblendpd Vx,Hx,Wx,lb</td>
<td>vblendpd Vx,Hx,Wx,lb</td>
<td>vblendpd Vx,Hx,Wx,lb</td>
<td>vblendpd Vx,Hx,Wx,lb</td>
<td>vblendpd Vx,Hx,Wx,lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vblendvps Vx,Hx,Wx,Lx</td>
<td>vblendvps Vx,Hx,Wx,Lx</td>
<td>vblendvps Vx,Hx,Wx,Lx</td>
<td>vblendvps Vx,Hx,Wx,Lx</td>
<td>vblendvps Vx,Hx,Wx,Lx</td>
<td>vblendvps Vx,Hx,Wx,Lx</td>
<td>vblendvps Vx,Hx,Wx,Lx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vblendvpdv Vx,Hx,Wx,Lx</td>
<td>vblendvpdv Vx,Hx,Wx,Lx</td>
<td>vblendvpdv Vx,Hx,Wx,Lx</td>
<td>vblendvpdv Vx,Hx,Wx,Lx</td>
<td>vblendvpdv Vx,Hx,Wx,Lx</td>
<td>vblendvpdv Vx,Hx,Wx,Lx</td>
<td>vblendvpdv Vx,Hx,Wx,Lx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vpblendvb Vx,Hx,Wx,Lx</td>
<td>vpblendvb Vx,Hx,Wx,Lx</td>
<td>vpblendvb Vx,Hx,Wx,Lx</td>
<td>vpblendvb Vx,Hx,Wx,Lx</td>
<td>vpblendvb Vx,Hx,Wx,Lx</td>
<td>vpblendvb Vx,Hx,Wx,Lx</td>
<td>vpblendvb Vx,Hx,Wx,Lx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vpalignr Vx,Hx,Wx,lb</td>
<td>vpalignr Vx,Hx,Wx,lb</td>
<td>vpalignr Vx,Hx,Wx,lb</td>
<td>vpalignr Vx,Hx,Wx,lb</td>
<td>vpalignr Vx,Hx,Wx,lb</td>
<td>vpalignr Vx,Hx,Wx,lb</td>
<td>vpalignr Vx,Hx,Wx,lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

<table>
<thead>
<tr>
<th>mod</th>
<th>nnn</th>
<th>R/M</th>
</tr>
</thead>
</table>

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

Example A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3 and Table A-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

A.4.2 Opcode Extension Tables

See Table A-6 below.
### Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Group</th>
<th>Mod 7,6</th>
<th>pfx</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>80-83</td>
<td>1</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1A</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL</td>
<td>2</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6, F7</td>
<td>3</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FE</td>
<td>4</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>5</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF 00</td>
<td>6</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF 01</td>
<td>7</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF BA</td>
<td>8</td>
<td>mem, 11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF C7</td>
<td>9</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF B9</td>
<td>10</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>11</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td>12</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF 71</td>
<td>13</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF 72</td>
<td>14</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)
### Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * (Contd.)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Group</th>
<th>Mod 7,6</th>
<th>pfx</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AE</td>
<td>15</td>
<td>mem</td>
<td></td>
<td>fsave</td>
<td>fxrstor</td>
<td>Idmxcsr</td>
<td>xmmcsr</td>
<td>XSAVE</td>
<td>XRSTOR</td>
<td>XSAVEOPT</td>
<td>clflush</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>RDFSBASE</td>
<td>Ry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11B</td>
<td>RDGSBASE</td>
<td>Ry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WRFSBASE</td>
<td>Ry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WRGSBASE</td>
<td>Ry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F 18</td>
<td>16</td>
<td>mem</td>
<td></td>
<td>prefetch</td>
<td>prefetch</td>
<td>prefetch</td>
<td>prefetch</td>
<td>T2</td>
<td></td>
<td></td>
<td>Reserved NOP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NTA</td>
<td>T0</td>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11B</td>
<td>Reserved NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.0F38 F3</td>
<td>17</td>
<td>mem</td>
<td></td>
<td>BLSR*</td>
<td>BLSMK*</td>
<td>BLS*</td>
<td>By, Ey</td>
<td>By, Ey</td>
<td>By, Ey</td>
<td>By, Ey</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1-and 2-byte opcodes (see A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

Example A-6. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:
- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

Example A-7. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:
- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A.4)</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD single-real</td>
<td>FMUL single-real</td>
<td>FCOM single-real</td>
<td>FCOMP single-real</td>
<td>FSUB single-real</td>
<td>FSUBR single-real</td>
<td>FDIV single-real</td>
<td>FDIVR single-real</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

#### Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD</td>
<td>single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FST</td>
<td>single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP</td>
<td>single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDCW</td>
<td>2 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTENV</td>
<td>14/28 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTCW</td>
<td>2 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Notes:
- All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIADD</td>
<td>dword-integer</td>
<td>FIMUL</td>
<td>dword-integer</td>
<td>FICOM</td>
<td>dword-integer</td>
<td>FICOMP</td>
<td>dword-integer</td>
<td>FISUB</td>
</tr>
<tr>
<td>FLD1</td>
<td>FLD2T</td>
<td>FLDL2E</td>
<td>FLDPI</td>
<td>FLDLG2</td>
<td>FLDLN2</td>
<td>FLDZ</td>
<td>FPREM</td>
<td>FYL2XP1</td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-12 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

#### A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISTTP</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIST</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISTP</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLD</td>
<td>extended-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP</td>
<td>extended-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>FCMOVNB</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
</tr>
<tr>
<td>D</td>
<td>FCMOVNBE</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
</tr>
<tr>
<td>E</td>
<td>FLEX</td>
<td>FINIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>FCOMI</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
</tr>
</tbody>
</table>

### Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>double-real</td>
<td>FMUL</td>
<td>double-real</td>
<td>FCOM</td>
<td>double-real</td>
<td>FCOMP</td>
<td>double-real</td>
<td>FSUB</td>
</tr>
<tr>
<td>FSUBR</td>
<td>double-real</td>
<td>FDIV</td>
<td>double-real</td>
<td>FDIVR</td>
<td>double-real</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.5.2.5 Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>double-real</td>
<td>FMUL</td>
<td>double-real</td>
<td>FCOM</td>
<td>double-real</td>
<td>FCOMP</td>
<td>double-real</td>
<td>FSUB</td>
</tr>
<tr>
<td>FSUBR</td>
<td>double-real</td>
<td>FDIV</td>
<td>double-real</td>
<td>FDIVR</td>
<td>double-real</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD</td>
<td>FLD</td>
<td>FISTTP</td>
<td>FST</td>
<td>FSTP</td>
<td>FRSTOR</td>
<td>FSQ</td>
<td>FSTSW</td>
<td>FSQ</td>
</tr>
<tr>
<td>double-real</td>
<td>integer64</td>
<td>double-real</td>
<td>double-real</td>
<td>double-real</td>
<td>98/108bytes</td>
<td>98/108bytes</td>
<td>2 bytes</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH *  
*

<table>
<thead>
<tr>
<th>C</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>R</td>
<td>E</td>
<td>FREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(0)</td>
<td>ST(1)</td>
<td>ST(2)</td>
<td>ST(3)</td>
<td>ST(4)</td>
<td>ST(5)</td>
<td>ST(6)</td>
<td>ST(7)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(0)</td>
<td>ST(1)</td>
<td>ST(2)</td>
<td>ST(3)</td>
<td>ST(4)</td>
<td>ST(5)</td>
<td>ST(6)</td>
<td>ST(7)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FUCOM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.7 Escape Opcodes with DE as First Byte**

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *  
*

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIADD</td>
<td>word-integer</td>
<td>FIMUL</td>
<td>word-integer</td>
<td>FICO</td>
<td>word-integer</td>
<td>FICOMP</td>
<td>word-integer</td>
<td>FISUB</td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
</tr>
<tr>
<td>1</td>
<td>FADDP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FSUBRP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>FF D</td>
<td>DIVP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CF M</td>
<td>ULP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DF C</td>
<td>COMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>FSUBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>FF D</td>
<td>DIVP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>FSTSW</td>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>FCOMIP</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>FUCOMIP</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
This page was intentionally left blank.
This appendix provides machine instruction formats and encodings of IA-32 instructions. The first section describes the IA-32 architecture’s machine instruction format. The remaining sections show the formats and encoding of general-purpose, MMX, P6 family, SSE/SSE2/SSE3, x87 FPU instructions, and VMX instructions. Those instruction formats also apply to Intel 64 architecture. Instruction formats used in 64-bit mode are provided as supersets of the above.

B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)

The following sections discuss this format.

B.1.1 Legacy Prefixes

The legacy prefixes noted in Figure B-1 include 66H, 67H, F2H and F3H. They are optional, except when F2H, F3H and 66H are used in new instruction extensions. Legacy prefixes must be placed before REX prefixes.

B.1.2  REX Prefixes

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.


B.1.3  Opcode Fields

The primary opcode for an instruction is encoded in one to three bytes of the instruction. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed.

Almost all instructions that refer to a register and/or memory operand have a register and/or address mode byte following the opcode. This byte, the ModR/M byte, consists of the mod field (2 bits), the reg field (3 bits; this field is sometimes an opcode extension), and the R/M field (3 bits). Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. Possible sizes are 8, 16, or 32 bits. If the instruction specifies an immediate value, the immediate value follows any displacement bytes. The immediate, if specified, is always the last field of the instruction.


B.1.4  Special Fields

Table B-1 lists bit fields that appear in certain instructions, sometimes within the opcode bytes. All of these fields (except the d bit) occur in the general-purpose instruction formats in Table B-13.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Number of Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>General-register specifier (see Table B-4 or B-5).</td>
<td>3</td>
</tr>
<tr>
<td>w</td>
<td>Specifies if data is byte or full-sized, where full-sized is 16 or 32 bits (see Table B-6).</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>Specifies sign extension of an immediate field (see Table B-7).</td>
<td>1</td>
</tr>
<tr>
<td>sreg2</td>
<td>Segment register specifier for CS, SS, DS, ES (see Table B-8).</td>
<td>2</td>
</tr>
<tr>
<td>sreg3</td>
<td>Segment register specifier for CS, SS, DS, ES, FS, GS (see Table B-8).</td>
<td>3</td>
</tr>
<tr>
<td>eee</td>
<td>Specifies a special-purpose (control or debug) register (see Table B-9).</td>
<td>3</td>
</tr>
<tr>
<td>ttttn</td>
<td>For conditional instructions, specifies a condition asserted or negated (see Table B-12).</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>Specifies direction of data operation (see Table B-11).</td>
<td>1</td>
</tr>
</tbody>
</table>
### B.1.4.1 Reg Field (reg) for Non-64-Bit Modes

The reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-3 shows the encoding of the reg field when the w bit is present.

#### Table B-2. Encoding of reg Field When w Field is Not Present in Instruction

<table>
<thead>
<tr>
<th>reg Field</th>
<th>Register Selected during 16-Bit Data Operations</th>
<th>Register Selected during 32-Bit Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>100</td>
<td>SP</td>
<td>ESP</td>
</tr>
<tr>
<td>101</td>
<td>BP</td>
<td>EBP</td>
</tr>
<tr>
<td>110</td>
<td>SI</td>
<td>ESI</td>
</tr>
<tr>
<td>111</td>
<td>DI</td>
<td>EDI</td>
</tr>
</tbody>
</table>

#### Table B-3. Encoding of reg Field When w Field is Present in Instruction

<table>
<thead>
<tr>
<th>reg</th>
<th>Register Specified by reg Field During 16-Bit Data Operations</th>
<th>Function of w Field When w = 0</th>
<th>Function of w Field When w = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AL</td>
<td>AX</td>
<td>AL</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>CX</td>
<td>CL</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>DX</td>
<td>DL</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>BX</td>
<td>BL</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>SP</td>
<td>AH</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>BP</td>
<td>CH</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>SI</td>
<td>DH</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>DI</td>
<td>BH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reg</th>
<th>Register Specified by reg Field During 32-Bit Data Operations</th>
<th>Function of w Field When w = 0</th>
<th>Function of w Field When w = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AL</td>
<td>EAX</td>
<td>AL</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>ECX</td>
<td>CL</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>EDX</td>
<td>DL</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>EBX</td>
<td>BL</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>ESP</td>
<td>AH</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>EBP</td>
<td>CH</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>ESI</td>
<td>DH</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>EDI</td>
<td>BH</td>
</tr>
</tbody>
</table>
### B.1.4.2  Reg Field (reg) for 64-Bit Mode

Just like in non-64-bit modes, the reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence of and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-4 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-5 shows the encoding of the reg field when the w bit is present.

**Table B-4. Encoding of reg Field When w Field is Not Present in Instruction**

<table>
<thead>
<tr>
<th>reg Field</th>
<th>Register Selected during 16-Bit Data Operations</th>
<th>Register Selected during 32-Bit Data Operations</th>
<th>Register Selected during 64-Bit Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AX</td>
<td>EAX</td>
<td>RAX</td>
</tr>
<tr>
<td>001</td>
<td>CX</td>
<td>ECX</td>
<td>RCX</td>
</tr>
<tr>
<td>010</td>
<td>DX</td>
<td>EDX</td>
<td>RDX</td>
</tr>
<tr>
<td>011</td>
<td>BX</td>
<td>EBX</td>
<td>RBX</td>
</tr>
<tr>
<td>100</td>
<td>SP</td>
<td>ESP</td>
<td>RSP</td>
</tr>
<tr>
<td>101</td>
<td>BP</td>
<td>EBP</td>
<td>RBP</td>
</tr>
<tr>
<td>110</td>
<td>SI</td>
<td>ESI</td>
<td>RSI</td>
</tr>
<tr>
<td>111</td>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
</tr>
</tbody>
</table>

**Table B-5. Encoding of reg Field When w Field is Present in Instruction**

<table>
<thead>
<tr>
<th>reg</th>
<th>Register Specified by reg Field During 16-Bit Data Operations</th>
<th>Function of w Field</th>
<th>Function of w Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>When w = 0</td>
<td>When w = 1</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>AL</td>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>CX</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>DX</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>AH[^2]</td>
<td>SP</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>DH[^2]</td>
<td>SI</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>BH[^2]</td>
<td>DI</td>
<td></td>
</tr>
</tbody>
</table>

**Table B-6. Encoding of Operand Size (w) Bit**

The current operand-size attribute determines whether the processor is performing 16-bit, 32-bit or 64-bit operations. Within the constraints of the current operand-size attribute, the operand-size bit (w) can be used to indicate operations on 8-bit operands or the full operand size specified with the operand-size attribute. Table B-6 shows the encoding of the w bit depending on the current operand-size attribute.

<table>
<thead>
<tr>
<th>w Bit</th>
<th>Operand Size When Operand-Size Attribute is 16 Bits</th>
<th>Operand Size When Operand-Size Attribute is 32 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 Bits</td>
<td>8 Bits</td>
</tr>
<tr>
<td>1</td>
<td>16 Bits</td>
<td>32 Bits</td>
</tr>
</tbody>
</table>

NOTES:
1. AH, CH, DH, BH can not be encoded when REX prefix is used. Such an expression defaults to the low byte.
B.1.4.4  Sign-Extend (s) Bit

The sign-extend (s) bit occurs in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. See Table B-7.

<table>
<thead>
<tr>
<th>s</th>
<th>Effect on 8-Bit Immediate Data</th>
<th>Effect on 16- or 32-Bit Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Sign-extend to fill 16-bit or 32-bit destination</td>
<td>None</td>
</tr>
</tbody>
</table>

B.1.4.5  Segment Register (sreg) Field

When an instruction operates on a segment register, the reg field in the ModR/M byte is called the sreg field and is used to specify the segment register. Table B-8 shows the encoding of the sreg field. This field is sometimes a 2-bit field (sreg2) and other times a 3-bit field (sreg3).

<table>
<thead>
<tr>
<th>2-Bit sreg2 Field</th>
<th>Segment Register Selected</th>
<th>3-Bit sreg3 Field</th>
<th>Segment Register Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ES</td>
<td>000</td>
<td>ES</td>
</tr>
<tr>
<td>01</td>
<td>CS</td>
<td>001</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>SS</td>
<td>010</td>
<td>SS</td>
</tr>
<tr>
<td>11</td>
<td>DS</td>
<td>011</td>
<td>DS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>FS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101</td>
<td>GS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>Reserved(^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

NOTES:
1. Do not use reserved encodings.

B.1.4.6  Special-Purpose Register (eee) Field

When control or debug registers are referenced in an instruction they are encoded in the eee field, located in bits 5 though 3 of the ModR/M byte (an alternate encoding of the sreg field). See Table B-9.

<table>
<thead>
<tr>
<th>eee</th>
<th>Control Register</th>
<th>Debug Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>CR0</td>
<td>DR0</td>
</tr>
<tr>
<td>001</td>
<td>Reserved(^1)</td>
<td>DR1</td>
</tr>
<tr>
<td>010</td>
<td>CR2</td>
<td>DR2</td>
</tr>
<tr>
<td>011</td>
<td>CR3</td>
<td>DR3</td>
</tr>
<tr>
<td>100</td>
<td>CR4</td>
<td>Reserved</td>
</tr>
<tr>
<td>101</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>110</td>
<td>Reserved</td>
<td>DR6</td>
</tr>
<tr>
<td>111</td>
<td>Reserved</td>
<td>DR7</td>
</tr>
</tbody>
</table>

NOTES:
1. Do not use reserved encodings.
B.1.4.7  Condition Test (tttn) Field

For conditional instructions (such as conditional jumps and set on condition), the condition test field (tttn) is encoded for the condition being tested. The ttt part of the field gives the condition to test and the n part indicates whether to use the condition (n = 0) or its negation (n = 1).

- For 1-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the opcode byte.
- For 2-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the second opcode byte.

Table B-10 shows the encoding of the tttn field.

<table>
<thead>
<tr>
<th>tttn</th>
<th>Mnemonic</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>O</td>
<td>Overflow</td>
</tr>
<tr>
<td>0001</td>
<td>NO</td>
<td>No overflow</td>
</tr>
<tr>
<td>0010</td>
<td>B, NAE</td>
<td>Below, Not above or equal</td>
</tr>
<tr>
<td>0011</td>
<td>NB, AE</td>
<td>Not below, Above or equal</td>
</tr>
<tr>
<td>0100</td>
<td>E, Z</td>
<td>Equal, Zero</td>
</tr>
<tr>
<td>0101</td>
<td>NE, NZ</td>
<td>Not equal, Not zero</td>
</tr>
<tr>
<td>0110</td>
<td>BE, NA</td>
<td>Below or equal, Not above</td>
</tr>
<tr>
<td>0111</td>
<td>NBE, A</td>
<td>Not below or equal, Above</td>
</tr>
<tr>
<td>1000</td>
<td>S</td>
<td>Sign</td>
</tr>
<tr>
<td>1001</td>
<td>NS</td>
<td>Not sign</td>
</tr>
<tr>
<td>1010</td>
<td>P, PE</td>
<td>Parity, Parity Even</td>
</tr>
<tr>
<td>1011</td>
<td>NP, PO</td>
<td>Not parity, Parity Odd</td>
</tr>
<tr>
<td>1100</td>
<td>L, NGE</td>
<td>Less than, Not greater than or equal to</td>
</tr>
<tr>
<td>1101</td>
<td>NL, GE</td>
<td>Not less than, Greater than or equal to</td>
</tr>
<tr>
<td>1110</td>
<td>LE, NG</td>
<td>Less than or equal to, Not greater than</td>
</tr>
<tr>
<td>1111</td>
<td>NLE, G</td>
<td>Not less than or equal to, Greater than</td>
</tr>
</tbody>
</table>

B.1.4.8  Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. See Table B-11.

- When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. Note that this bit does not appear as the symbol "d" in Table B-13; the actual encoding of the bit as 1 or 0 is given.
- When used for floating-point instructions (in Table B-16), the d bit is shown as bit 2 of the first byte of the primary opcode.

Table B-11. Encoding of Operation Direction (d) Bit

<table>
<thead>
<tr>
<th>d</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reg Field</td>
<td>ModR/M or SIB Byte</td>
</tr>
<tr>
<td>1</td>
<td>ModR/M or SIB Byte</td>
<td>reg Field</td>
</tr>
</tbody>
</table>

B.1.5  Other Notes

Table B-12 contains notes on particular encodings. These notes are indicated in the tables shown in the following sections by superscripts.
Table B-12. Notes on Instruction Encoding

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A value of 11B in bits 7 and 6 of the ModR/M byte is reserved.</td>
</tr>
<tr>
<td>B</td>
<td>A value of 01B (or 10B) in bits 7 and 6 of the ModR/M byte is reserved.</td>
</tr>
</tbody>
</table>

### B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES

Table B-13 shows machine instruction formats and encodings for general purpose instructions in non-64-bit modes.

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA – ASCII Adjust after Addition</td>
<td>0011 0111</td>
</tr>
<tr>
<td>AAD – ASCII Adjust AX before Division</td>
<td>1101 0101 : 0000 1010</td>
</tr>
<tr>
<td>AAM – ASCII Adjust AX after Multiply</td>
<td>1101 0100 : 0000 1010</td>
</tr>
<tr>
<td>AAS – ASCII Adjust AL after Subtraction</td>
<td>0011 1111</td>
</tr>
<tr>
<td>ADC – ADD with Carry</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0001 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0001 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>001 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0001 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 010 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 010 r/m : immediate data</td>
</tr>
<tr>
<td>ADD – Add</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0000 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td>AND – Logical AND</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0010 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0010 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0010 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0010 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 100 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 100 r/m : immediate data</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARPL</strong> – Adjust RPL Field of Selector</td>
<td>0110 0011 : 11 reg1 reg2</td>
</tr>
<tr>
<td>from register</td>
<td>0110 0011 : mod reg r/m</td>
</tr>
<tr>
<td>from memory</td>
<td>0110 0010 : modA reg r/m</td>
</tr>
<tr>
<td><strong>BOUND</strong> – Check Array Against Bounds</td>
<td>0110 0010 : modA reg r/m</td>
</tr>
<tr>
<td><strong>BSF</strong> – Bit Scan Forward</td>
<td>0000 1111 : 1011 1100 : 11 reg1 reg2</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 1100 : mod reg r/m</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 1101 : 11 reg1 reg2</td>
</tr>
<tr>
<td><strong>BSR</strong> – Bit Scan Reverse</td>
<td>0000 1111 : 1011 1101 : 11 reg1 reg2</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 1101 : mod reg r/m</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1100 : 1 reg</td>
</tr>
<tr>
<td><strong>BSWAP</strong> – Byte Swap</td>
<td>0000 1111 : 1100 1 reg</td>
</tr>
<tr>
<td><strong>BT</strong> – Bit Test</td>
<td>0000 1111 : 1011 1010 : 11 100 reg: imm8 data</td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : mod 100 r/m : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1011 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTC</strong> – Bit Test and Complement</td>
<td>0000 1111 : 1011 1010 : 11 111 reg: imm8 data</td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : mod 111 r/m : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1011 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTR</strong> – Bit Test and Reset</td>
<td>0000 1111 : 1011 1010 : 11 110 reg: imm8 data</td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : mod 110 r/m : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1011 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTS</strong> – Bit Test and Set</td>
<td>0000 1111 : 1011 1010 : 11 101 reg: imm8 data</td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : mod 101 r/m : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1010 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1010 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>CALL</strong> – Call Procedure (in same segment)</td>
<td>1110 0000 : full displacement</td>
</tr>
<tr>
<td>direct</td>
<td>1110 0100 : unsigned full offset, selector</td>
</tr>
<tr>
<td>register indirect</td>
<td>1111 1111 : 11 010 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>1111 1111 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>CALL</strong> – Call Procedure (in other segment)</td>
<td>1001 0101 : 1 mod 011 r/m</td>
</tr>
<tr>
<td>direct</td>
<td>1111 1111 : mod 011 r/m</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBW – Convert Byte to Word</td>
<td>1001 1000</td>
</tr>
<tr>
<td>CDQ - Convert Doubleword to Qword</td>
<td>1001 1001</td>
</tr>
<tr>
<td>CLC – Clear Carry Flag</td>
<td>1111 1000</td>
</tr>
<tr>
<td>CLD – Clear Direction Flag</td>
<td>1111 1100</td>
</tr>
<tr>
<td>CLI – Clear Interrupt Flag</td>
<td>1111 1010</td>
</tr>
<tr>
<td>CLTS – Clear Task-Switched Flag in CR0</td>
<td>0000 1111 : 0000 0110</td>
</tr>
<tr>
<td>CMC – Complement Carry Flag</td>
<td>1111 0101</td>
</tr>
<tr>
<td>CMP – Compare Two Operands</td>
<td></td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0011 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 with register1</td>
<td>0011 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory with register</td>
<td>0011 100w : mod reg r/m</td>
</tr>
<tr>
<td>register with memory</td>
<td>0011 101w : mod reg r/m</td>
</tr>
<tr>
<td>immediate with register</td>
<td>1000 00sw : 11 111 reg : immediate data</td>
</tr>
<tr>
<td>immediate with AL, AX, or EAX</td>
<td>0011 110w : immediate data</td>
</tr>
<tr>
<td>immediate with memory</td>
<td>1000 00sw : mod 111 r/m : immediate data</td>
</tr>
<tr>
<td>CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands</td>
<td>1010 011w</td>
</tr>
<tr>
<td>CMPXCHG – Compare and Exchange</td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 000w : mod reg r/m</td>
</tr>
<tr>
<td>CPUID – CPU Identification</td>
<td>0000 1111 : 1010 0010</td>
</tr>
<tr>
<td>CWD – Convert Word to Doubleword</td>
<td>1001 1001</td>
</tr>
<tr>
<td>CWDE – Convert Word to Doubleword</td>
<td>1001 1000</td>
</tr>
<tr>
<td>DAA – Decimal Adjust AL after Addition</td>
<td>0010 0111</td>
</tr>
<tr>
<td>DAS – Decimal Adjust AL after Subtraction</td>
<td>0010 1111</td>
</tr>
<tr>
<td>DEC – Decrement by 1</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1111 111w : 11 001 reg</td>
</tr>
<tr>
<td>register (alternate encoding)</td>
<td>0100 1 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 111w : mod 001 r/m</td>
</tr>
<tr>
<td>DIV - Unsigned Divide</td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>1111 011w : 11 110 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>1111 011w : mod 110 r/m</td>
</tr>
<tr>
<td>HLT – Halt</td>
<td>1111 0100</td>
</tr>
<tr>
<td>IDIV – Signed Divide</td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>1111 011w : 11 111 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>1111 011w : mod 111 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td><strong>IMUL – Signed Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>1111 011w : 11 101 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>1111 011w : mod 101 reg</td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0000 1111 : 1010 1111 : 11 : reg1 reg2</td>
</tr>
<tr>
<td>register with memory</td>
<td>0000 1111 : 1010 1111 : mod reg r/m</td>
</tr>
<tr>
<td>register1 with immediate to register2</td>
<td>0110 101s : 11 reg1 reg2 : immediate data</td>
</tr>
<tr>
<td>memory with immediate to register</td>
<td>0110 101s : mod reg r/m : immediate data</td>
</tr>
<tr>
<td><strong>IN – Input From Port</strong></td>
<td></td>
</tr>
<tr>
<td>fixed port</td>
<td>1110 010w : port number</td>
</tr>
<tr>
<td>variable port</td>
<td>1110 110w</td>
</tr>
<tr>
<td><strong>INC – Increment by 1</strong></td>
<td></td>
</tr>
<tr>
<td>reg</td>
<td>1111 111w : 11 000 reg</td>
</tr>
<tr>
<td>reg (alternate encoding)</td>
<td>0100 0 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 111w : mod 000 r/m</td>
</tr>
<tr>
<td><strong>INS – Input from DX Port</strong></td>
<td>0110 110w</td>
</tr>
<tr>
<td><strong>INT n – Interrupt Type n</strong></td>
<td>1100 1101 : type</td>
</tr>
<tr>
<td><strong>INT – Single-Step Interrupt 3</strong></td>
<td>1100 1100</td>
</tr>
<tr>
<td><strong>INTO – Interrupt 4 on Overflow</strong></td>
<td>1100 1110</td>
</tr>
<tr>
<td><strong>INVD – Invalidate Cache</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 0000 1000</td>
</tr>
<tr>
<td><strong>INVLP – Invalidate TLB Entry</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 0000 0001 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>INVPCID – Invalidate Process-Context Identifier</strong></td>
<td>0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m</td>
</tr>
<tr>
<td><strong>IRET/IRETD – Interrupt Return</strong></td>
<td>1100 1111</td>
</tr>
<tr>
<td>jcc – Jump if Condition is Met</td>
<td></td>
</tr>
<tr>
<td>8-bit displacement</td>
<td>0111 tttn : 8-bit displacement</td>
</tr>
<tr>
<td>full displacement</td>
<td>0000 1111 : 1000 ttttn : full displacement</td>
</tr>
<tr>
<td><strong>JCXZ/JECXZ – Jump on CX/ECX Zero</strong></td>
<td></td>
</tr>
<tr>
<td>Address-size prefix differentiates JCXZ and JECXZ</td>
<td>1110 0011 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>JMP – Unconditional Jump (to same segment)</strong></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>1110 1011 : 8-bit displacement</td>
</tr>
<tr>
<td>direct</td>
<td>1110 1001 : full displacement</td>
</tr>
<tr>
<td>register indirect</td>
<td>1111 1111 : 11 100 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>1111 1111 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>JMP – Unconditional Jump (to other segment)</strong></td>
<td></td>
</tr>
<tr>
<td>direct intersegment</td>
<td>1110 1010 : unsigned full offset, selector</td>
</tr>
<tr>
<td>indirect intersegment</td>
<td>1111 1111 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>LAHF – Load Flags into AHRegister</strong></td>
<td>1001 1111</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LAR – Load Access Rights Byte</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0000 1111 : 0000 0010 : 11 reg1 reg2</td>
</tr>
<tr>
<td>from memory</td>
<td>0000 1111 : 0000 0010 : mod reg r/m</td>
</tr>
<tr>
<td><strong>LDS – Load Pointer to DS</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100 0101 : modA,B reg r/m</td>
</tr>
<tr>
<td><strong>LEA – Load Effective Address</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000 1101 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LEAVE – High Level Procedure Exit</strong></td>
<td>1100 1001</td>
</tr>
<tr>
<td><strong>LES – Load Pointer to ES</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100 0100 : modA,B reg r/m</td>
</tr>
<tr>
<td><strong>LFS – Load Pointer to FS</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 1011 0100 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LGDT – Load Global Descriptor Table Register</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 0000 0001 : modA 010 r/m</td>
</tr>
<tr>
<td><strong>LGS – Load Pointer to GS</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 1011 0101 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LIDT – Load Interrupt Descriptor Table Register</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 0000 0001 : modA 011 r/m</td>
</tr>
<tr>
<td><strong>LLDT – Load Local Descriptor Table Register</strong></td>
<td></td>
</tr>
<tr>
<td>LDTR from register</td>
<td>0000 1111 : 0000 0000 : 11 010 reg</td>
</tr>
<tr>
<td>LDTR from memory</td>
<td>0000 1111 : 0000 0000 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>LMSW – Load Machine Status Word</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0000 1111 : 0000 0001 : 11 110 reg</td>
</tr>
<tr>
<td>from memory</td>
<td>0000 1111 : 0000 0001 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>LOCK – Assert LOCK# Signal Prefix</strong></td>
<td>1111 0000</td>
</tr>
<tr>
<td><strong>LODS/LODSB/LODSW/LODSD – Load String Operand</strong></td>
<td>1010 110w</td>
</tr>
<tr>
<td><strong>LOOP – Loop Count</strong></td>
<td>1110 0010 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOOPZ/LOOPE – Loop Count while Zero/Equal</strong></td>
<td>1110 0001 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOOPNZ/LOOPNE – Loop Count while not Zero/Equal</strong></td>
<td>1110 0000 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>LSL – Load Segment Limit</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0000 1111 : 0000 0011 : 11 reg1 reg2</td>
</tr>
<tr>
<td>from memory</td>
<td>0000 1111 : 0000 0011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>LSS – Load Pointer to SS</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 1011 0010 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LTR – Load Task Register</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0000 1111 : 0000 0000 : 11 011 reg</td>
</tr>
<tr>
<td>from memory</td>
<td>0000 1111 : 0000 0000 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>MOV – Move Data</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>1000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>1000 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to reg</td>
<td>1000 101w : mod reg r/m</td>
</tr>
<tr>
<td>reg to memory</td>
<td>1000 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1100 011w : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate to register (alternate encoding)</td>
<td>1011 w reg : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1100 011w : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td>memory to AL, AX, or EAX</td>
<td>1010 000w : full displacement</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL, AX, or EAX to memory</td>
<td>1010 001w : full displacement</td>
</tr>
<tr>
<td><strong>MOV - Move to/from Control Registers</strong></td>
<td></td>
</tr>
<tr>
<td>CR0 from register</td>
<td>0000 1111 : 0010 0010 : -- 000 reg</td>
</tr>
<tr>
<td>CR2 from register</td>
<td>0000 1111 : 0010 0010 : -- 010 reg</td>
</tr>
<tr>
<td>CR3 from register</td>
<td>0000 1111 : 0010 0010 : -- 011 reg</td>
</tr>
<tr>
<td>CR4 from register</td>
<td>0000 1111 : 0010 0010 : -- 100 reg</td>
</tr>
<tr>
<td>register from CR0-CR4</td>
<td>0000 1111 : 0010 0000 : -- eee reg</td>
</tr>
<tr>
<td><strong>MOV - Move to/from Debug Registers</strong></td>
<td></td>
</tr>
<tr>
<td>DR0-DR3 from register</td>
<td>0000 1111 : 0010 0011 : -- eee reg</td>
</tr>
<tr>
<td>DR4-DR5 from register</td>
<td>0000 1111 : 0010 0011 : -- eee reg</td>
</tr>
<tr>
<td>DR6-DR7 from register</td>
<td>0000 1111 : 0010 0011 : -- eee reg</td>
</tr>
<tr>
<td>register from DR6-DR7</td>
<td>0000 1111 : 0010 0001 : -- eee reg</td>
</tr>
<tr>
<td>register from DR4-DR5</td>
<td>0000 1111 : 0010 0001 : -- eee reg</td>
</tr>
<tr>
<td>register from DR0-DR3</td>
<td>0000 1111 : 0010 0001 : -- eee reg</td>
</tr>
<tr>
<td><strong>MOV - Move to/from Segment Registers</strong></td>
<td></td>
</tr>
<tr>
<td>register to segment register</td>
<td>1000 1110 : 11 sreg3 reg</td>
</tr>
<tr>
<td>register to SS</td>
<td>1000 1110 : 11 sreg3 reg</td>
</tr>
<tr>
<td>memory to segment reg</td>
<td>1000 1110 : mod sreg3 r/m</td>
</tr>
<tr>
<td>memory to SS</td>
<td>1000 1110 : mod sreg3 r/m</td>
</tr>
<tr>
<td>segment register to register</td>
<td>1000 1100 : 11 sreg3 reg</td>
</tr>
<tr>
<td>segment register to memory</td>
<td>1000 1100 : mod sreg3 r/m</td>
</tr>
<tr>
<td><strong>MOVBE - Move data after swapping bytes</strong></td>
<td></td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 1111 : 0011 1011 0000 : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 1111 : 0011 1001 1000 : mod reg r/m</td>
</tr>
<tr>
<td><strong>MOVS/MOVSB/MOVSW/MOVSD - Move Data from String to String</strong></td>
<td>1010 010w</td>
</tr>
<tr>
<td><strong>MOVSX - Move with Sign-Extend</strong></td>
<td></td>
</tr>
<tr>
<td>memory to reg</td>
<td>0000 1111 : 1011 111w : mod reg r/m</td>
</tr>
<tr>
<td><strong>MOVZX - Move with Zero-Extend</strong></td>
<td></td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 1111 : 1011 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 1111 : 1011 011w : mod reg r/m</td>
</tr>
<tr>
<td><strong>MUL - Unsigned Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>1111 011w : 11 100 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>1111 011w : mod 100 r/m</td>
</tr>
<tr>
<td><strong>NEG - Two's Complement Negation</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1111 011w : 11 011 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 011w : mod 011 r/m</td>
</tr>
<tr>
<td><strong>NOP - No Operation</strong></td>
<td>1001 0000</td>
</tr>
</tbody>
</table>
Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP – Multi-byte No Operation¹</td>
<td>register 0000 1111 0001 1111 : 11 000 reg</td>
</tr>
<tr>
<td></td>
<td>memory 0000 1111 0001 1111 : mod 000 r/m</td>
</tr>
<tr>
<td>NOT – One's Complement Negation</td>
<td>register 1111 011w : 11 010 reg</td>
</tr>
<tr>
<td></td>
<td>memory 1111 011w : mod 010 r/m</td>
</tr>
<tr>
<td>OR – Logical Inclusive OR</td>
<td>register1 to register2 0000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td></td>
<td>register2 to register1 0000 101w : 11 reg2 reg1</td>
</tr>
<tr>
<td></td>
<td>memory to register 0000 101w : mod reg r/m</td>
</tr>
<tr>
<td></td>
<td>register to memory 0000 100w : mod reg r/m</td>
</tr>
<tr>
<td></td>
<td>immediate to register 1000 00sw : 11 001 reg : immediate data</td>
</tr>
<tr>
<td></td>
<td>immediate to AL, AX, or EAX 0000 110w : immediate data</td>
</tr>
<tr>
<td></td>
<td>immediate to memory 1000 00sw : mod 001 r/m : immediate data</td>
</tr>
<tr>
<td>OUT – Output to Port</td>
<td>fixed port 1110 011w : port number</td>
</tr>
<tr>
<td></td>
<td>variable port 1110 111w</td>
</tr>
<tr>
<td>OUTS – Output to DX Port</td>
<td>0110 111w</td>
</tr>
<tr>
<td>POP – Pop a Word from the Stack</td>
<td>register 1000 1111 : 11 000 reg</td>
</tr>
<tr>
<td></td>
<td>register (alternate encoding) 0101 1 reg</td>
</tr>
<tr>
<td></td>
<td>memory 1000 1111 : mod 000 r/m</td>
</tr>
<tr>
<td>POP – Pop a Segment Register from the Stack</td>
<td>segment register DS, ES 000 sreg2 111</td>
</tr>
<tr>
<td></td>
<td>segment register SS 000 sreg2 111</td>
</tr>
<tr>
<td></td>
<td>segment register FS, GS 0000 1111 : 10 sreg3 001</td>
</tr>
<tr>
<td>(Note: CS cannot be sreg2 in this usage.)</td>
<td>POPA/POPAD – Pop All General Registers 0110 0001</td>
</tr>
<tr>
<td></td>
<td>POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register 1001 1101</td>
</tr>
<tr>
<td>PUSH – Push Operand onto the Stack</td>
<td>register 1111 1111 : 11 110 reg</td>
</tr>
<tr>
<td></td>
<td>register (alternate encoding) 0101 0 reg</td>
</tr>
<tr>
<td></td>
<td>memory 1111 1111 : mod 110 r/m</td>
</tr>
<tr>
<td></td>
<td>immediate 0110 10s0 : immediate data</td>
</tr>
<tr>
<td>PUSH – Push Segment Register onto the Stack</td>
<td>segment register CS,DS,ES,SS 000 sreg2 110</td>
</tr>
<tr>
<td></td>
<td>segment register FS,GS 0000 1111 : 10 sreg3 000</td>
</tr>
</tbody>
</table>
### TABLE B-13. GENERAL PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUSHA/PUSHAD</strong> – Push All General Registers</td>
<td>0110 0000</td>
</tr>
<tr>
<td><strong>PUSHF/PUSHFD</strong> – Push Flags Register onto the Stack</td>
<td>1001 1100</td>
</tr>
<tr>
<td><strong>RCL</strong> – Rotate thru Carry Left</td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 010 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 010 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 010 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 010 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 010 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 010 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>RCR</strong> – Rotate thru Carry Right</td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 011 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 011 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 011 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 011 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 011 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 011 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>RDMSR</strong> – Read from Model-Specific Register</td>
<td>0000 1111 : 0011 0010</td>
</tr>
<tr>
<td><strong>RDPMC</strong> – Read Performance Monitoring Counters</td>
<td>0000 1111 : 0011 0011</td>
</tr>
<tr>
<td><strong>RDTSC</strong> – Read Time-Stamp Counter</td>
<td>0000 1111 : 0011 0001</td>
</tr>
<tr>
<td><strong>RDTSCP</strong> – Read Time-Stamp Counter and Processor ID</td>
<td>0000 1111 : 0000 0001 : 1111 1001</td>
</tr>
<tr>
<td><strong>REP INS</strong> – Input String</td>
<td>1111 0011 : 0110 110w</td>
</tr>
<tr>
<td><strong>REP LODS</strong> – Load String</td>
<td>1111 0011 : 1010 110w</td>
</tr>
<tr>
<td><strong>REP MOVS</strong> – Move String</td>
<td>1111 0011 : 1010 010w</td>
</tr>
<tr>
<td><strong>REP OUTS</strong> – Output String</td>
<td>1111 0011 : 0110 111w</td>
</tr>
<tr>
<td><strong>REP STOS</strong> – Store String</td>
<td>1111 0011 : 1010 110w</td>
</tr>
<tr>
<td><strong>REPE CMPS</strong> – Compare String</td>
<td>1111 0011 : 1010 011w</td>
</tr>
<tr>
<td><strong>REPE SCAS</strong> – Scan String</td>
<td>1111 0011 : 1010 111w</td>
</tr>
<tr>
<td><strong>REPNE CMPS</strong> – Compare String</td>
<td>1111 0010 : 1010 011w</td>
</tr>
<tr>
<td><strong>REPNE SCAS</strong> – Scan String</td>
<td>1111 0010 : 1010 111w</td>
</tr>
<tr>
<td><strong>RET</strong> – Return from Procedure (to same segment)</td>
<td></td>
</tr>
<tr>
<td>no argument</td>
<td>1100 0011</td>
</tr>
<tr>
<td>adding immediate to SP</td>
<td>1100 0010 : 16-bit displacement</td>
</tr>
<tr>
<td><strong>RET</strong> – Return from Procedure (to other segment)</td>
<td></td>
</tr>
<tr>
<td>intersegment</td>
<td>1100 1011</td>
</tr>
<tr>
<td>adding immediate to SP</td>
<td>1100 1010 : 16-bit displacement</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROL – Rotate Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 000 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 000 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 000 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 000 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 000 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 000 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>ROR – Rotate Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 001 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 001 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 001 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 001 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 001 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 001 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>RSM – Resume from System Management Mode</strong></td>
<td>0000 1111 : 1010 1010</td>
</tr>
<tr>
<td><strong>SAHF – Store AH into Flags</strong></td>
<td>1001 1110</td>
</tr>
<tr>
<td><strong>SAL – Shift Arithmetic Left</strong></td>
<td>same instruction as SHL</td>
</tr>
<tr>
<td><strong>SAR – Shift Arithmetic Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 111 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 111 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 111 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 111 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 111 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 111 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>SBB – Integer Subtraction with Borrow</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0001 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0001 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0001 101w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0001 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 011 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 110w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 011 r/m : immediate data</td>
</tr>
<tr>
<td><strong>SCAS/SCASB/SCASW/SCASD – Scan String</strong></td>
<td>1010 111w</td>
</tr>
<tr>
<td><strong>SETcc – Byte Set on Condition</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 1001 tttt : 11 000 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 1001 tttt : mod 000 r/m</td>
</tr>
<tr>
<td><strong>SGDT – Store Global Descriptor Table Register</strong></td>
<td>0000 1111 : 0000 0001 : modA 000 r/m</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHL – Shift Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 100 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 100 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 100 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 100 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 100 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 100 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>SHLD – Double Precision Shift Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0000 1111 : 1010 0100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0000 1111 : 1010 0101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0000 1111 : 1010 0101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>SHR – Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 101 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 101 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 101 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 101 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 101 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 101 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>SHRD – Double Precision Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0000 1111 : 1010 1100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0000 1111 : 1010 1101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0000 1111 : 1010 1101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>SIDT – Store Interrupt Descriptor Table Register</strong></td>
<td>0000 1111 : 0000 0001 : modA 001 r/m</td>
</tr>
<tr>
<td><strong>SLDT – Store Local Descriptor Table Register</strong></td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>0000 1111 : 0000 0000 : 11 000 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0000 1111 : 0000 0000 : mod 000 r/m</td>
</tr>
<tr>
<td><strong>SMSW – Store Machine Status Word</strong></td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>0000 1111 : 0000 0001 : 11 100 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0000 1111 : 0000 0001 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>STC – Set Carry Flag</strong></td>
<td>1111 1001</td>
</tr>
<tr>
<td><strong>STD – Set Direction Flag</strong></td>
<td>1111 1101</td>
</tr>
<tr>
<td><strong>STI – Set Interrupt Flag</strong></td>
<td>1111 1011</td>
</tr>
<tr>
<td><strong>STOS/STOSB/STOSW/STOSD – Store String Data</strong></td>
<td>1010 101w</td>
</tr>
<tr>
<td><strong>STR – Store Task Register</strong></td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>0000 1111 : 0000 0000 : 11 001 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0000 1111 : 0000 0000 : mod 001 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>SUB – Integer Subtraction</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0010 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0010 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0010 101w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0010 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 101 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 110w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 101 r/m : immediate data</td>
</tr>
<tr>
<td>TEST – Logical Compare</td>
<td></td>
</tr>
<tr>
<td>register1 and register2</td>
<td>1000 010w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory and register</td>
<td>1000 010w : mod reg r/m</td>
</tr>
<tr>
<td>immediate and register</td>
<td>1111 011w : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate and AL, AX, or EAX</td>
<td>1010 100w : immediate data</td>
</tr>
<tr>
<td>immediate and memory</td>
<td>1111 011w : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td>UD0 – Undefined instruction</td>
<td>0000 1111 : 1111 1111</td>
</tr>
<tr>
<td>UD1 – Undefined instruction</td>
<td>0000 1111 : 0000 1011</td>
</tr>
<tr>
<td>UD2 – Undefined instruction</td>
<td>0000 FFFF : 0000 1011</td>
</tr>
<tr>
<td>VERR – Verify a Segment for Reading</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 0000 0000 : 11 100 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 0000 0000 : mod 100 r/m</td>
</tr>
<tr>
<td>VERW – Verify a Segment for Writing</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 0000 0000 : 11 101 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 0000 0000 : mod 101 r/m</td>
</tr>
<tr>
<td>WAIT – Wait</td>
<td>1001 1011</td>
</tr>
<tr>
<td>WBINVD – Writeback and Invalidate Data Cache</td>
<td>0000 1111 : 0000 1001</td>
</tr>
<tr>
<td>WRMSR – Write to Model-Specific Register</td>
<td>0000 1111 : 0011 0000</td>
</tr>
<tr>
<td>XADD – Exchange and Add</td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1100 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1100 000w : mod reg r/m</td>
</tr>
<tr>
<td>XCHG – Exchange Register/Memory with Register</td>
<td></td>
</tr>
<tr>
<td>register1 with register2</td>
<td>1000 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>AX or EAX with reg</td>
<td>1001 0 reg</td>
</tr>
<tr>
<td>memory with reg</td>
<td>1000 011w : mod reg r/m</td>
</tr>
<tr>
<td>XLAT/XLATB – Table Look-up Translation</td>
<td>1101 0111</td>
</tr>
<tr>
<td>XOR – Logical Exclusive OR</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0011 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0011 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0011 001w : mod reg r/m</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register to memory</td>
<td>0011 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 110 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0011 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 110 r/m : immediate data</td>
</tr>
</tbody>
</table>

Prefix Bytes

<table>
<thead>
<tr>
<th>Prefix Bytes</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>address size</td>
<td>0110 0111</td>
</tr>
<tr>
<td>LOCK</td>
<td>1111 0000</td>
</tr>
<tr>
<td>operand size</td>
<td>0110 0110</td>
</tr>
<tr>
<td>CS segment override</td>
<td>0010 1110</td>
</tr>
<tr>
<td>DS segment override</td>
<td>0011 1110</td>
</tr>
<tr>
<td>ES segment override</td>
<td>0010 0110</td>
</tr>
<tr>
<td>FS segment override</td>
<td>0110 0100</td>
</tr>
<tr>
<td>GS segment override</td>
<td>0110 0101</td>
</tr>
<tr>
<td>SS segment override</td>
<td>0011 0110</td>
</tr>
</tbody>
</table>

NOTES:
1. The multi-byte NOP instruction does not alter the content of the register and will not issue a memory operation.

B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode

Table B-15 shows machine instruction formats and encodings for general purpose instructions in 64-bit mode.

Table B-14. Special Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>If the value of REX.W. is 1, it overrides the presence of 66H.</td>
</tr>
<tr>
<td>w</td>
<td>The value of bit W. in REX is has no effect.</td>
</tr>
</tbody>
</table>

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC – ADD with Carry</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B : 0001 0001 : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B : 0001 0001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0001 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B : 0001 0011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0001 001w : mod reg r/m</td>
</tr>
<tr>
<td>memory to qwordregister</td>
<td>0100 1RXB : 0001 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0001 000w : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory</td>
<td>0100 1RXB : 0001 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 000B : 1000 000w : 11 010 reg : immediate</td>
</tr>
<tr>
<td>immediate to qwordregister</td>
<td>0100 100B : 1000 0001 : 11 010 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate to qwordregister</td>
<td>0100 1R0B : 1000 0011 : 11 010 qwordreg : imm8</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 010w : immediate data</td>
</tr>
<tr>
<td>immediate to RAX</td>
<td>0100 1000 : 0000 0101 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB : 1000 00sw : mod 010 r/m : immediate</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB : 1000 0001 : mod 010 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB : 1000 0031 : mod 010 r/m : imm8</td>
</tr>
<tr>
<td><strong>ADD – Add</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B : 0000 0000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0000 0000 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0000 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0000 0010 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 001w : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB : 0000 0000 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0000 000w : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB : 0000 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 0000B : 1000 00sw : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 100B : 1000 0001 : 11 010 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 010w : immediate8</td>
</tr>
<tr>
<td>immediate to RAX</td>
<td>0100 1000 : 0000 0101 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB : 1000 00sw : mod 000 r/m : immediate</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB : 1000 0001 : mod 010 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB : 1000 0011 : mod 010 r/m : imm8</td>
</tr>
<tr>
<td><strong>AND – Logical AND</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B 0010 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0010 0001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0010 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 1R0B 0010 0011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB 0010 001w : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB : 0010 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0010 000w : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB : 0010 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 0000B : 1000 00sw : 11 100 reg : immediate</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 100B 1000 0001 : 11 100 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 010w : immediate</td>
</tr>
<tr>
<td>immediate32 to RAX</td>
<td>0100 1000 0101 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB : 1000 00sw : mod 100 r/m : immediate</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB : 1000 0001 : mod 100 r/m : immediate32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB : 1000 0011 : mod 100 r/m : imm8</td>
</tr>
</tbody>
</table>

### BSF – Bit Scan Forward
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1011 1100 : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1011 1100 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 1100 : mod reg r/m</td>
</tr>
<tr>
<td>memory64, qwordregister</td>
<td>0100 1RXB 0000 1111 : 1011 1100 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BSR – Bit Scan Reverse**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1011 1101 : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1011 1101 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 1101 : mod reg r/m</td>
</tr>
<tr>
<td>memory64, qwordregister</td>
<td>0100 1RXB 0000 1111 : 1011 1101 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BSwap – Byte Swap**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSwap – Byte Swap</td>
<td>0000 1111 : 1100 1 reg</td>
</tr>
</tbody>
</table>

**BT – Bit Test**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register, immediate</td>
<td>0100 000B 0000 1111 : 1011 1010 : 11 100 reg: imm8</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
<td>0100 100B 1111 : 1011 1010 : 11 100 qwordreg: imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0100 00XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8</td>
</tr>
<tr>
<td>memory64, immediate8</td>
<td>0100 10XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1010 0011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1010 0011 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0100 0RXB 0000 1111 : 1010 0011 : mod reg r/m</td>
</tr>
<tr>
<td>memory, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1010 0011 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BTC – Bit Test and Complement**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register, immediate</td>
<td>0100 000B 0000 1111 : 1011 1010 : 11 111 reg: imm8</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
<td>0100 100B 1111 : 1011 1010 : 11 111 qwordreg: imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0100 00XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>memory64, immediate8</td>
<td>0100 10XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1011 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1011 1011 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 1011 : mod reg r/m</td>
</tr>
<tr>
<td>memory, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1011 1011 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BTR – Bit Test and Reset**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register, immediate</td>
<td>0100 000B 0000 1111 : 1011 1010 : 11 110 reg: imm8</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
<td>0100 100B 1111 : 1011 1010 : 11 110 qwordreg: imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0100 00XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8</td>
</tr>
<tr>
<td>memory64, immediate8</td>
<td>0100 10XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8 data</td>
</tr>
</tbody>
</table>
Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1ROB 0000 1111 : 1011 0011 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 0011 : mod reg r/m</td>
</tr>
<tr>
<td>memory64, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1011 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>BTS – Bit Test and Set</strong></td>
<td></td>
</tr>
<tr>
<td>register, immediate</td>
<td>0100 000B 0000 1111 : 1011 1010 : 11 101 reg: imm8</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
<td>0100 100B 0000 1111 : 1011 1010 : 11 101 qwordreg: imm8</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0100 00XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>memory64, immediate8</td>
<td>0100 10XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0ROB 0000 1111 : 1010 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1ROB 0000 1111 : 1010 1011 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1010 1011 : mod reg r/m</td>
</tr>
<tr>
<td>memory64, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1010 1011 : mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>CALL – Call Procedure (in same segment)</strong></td>
<td></td>
</tr>
<tr>
<td>direct</td>
<td>1110 1000 : displacement32</td>
</tr>
<tr>
<td>register indirect</td>
<td>0100 WR00w 1111 1111 : 11 010 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>0100 WDXBwr 1111 1111 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>CALL – Call Procedure (in other segment)</strong></td>
<td></td>
</tr>
<tr>
<td>indirect</td>
<td>1111 1111 : mod 011 r/m</td>
</tr>
<tr>
<td>indirect</td>
<td>0100 10XB 0100 1000 1111 1111 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>CBW – Convert Byte to Word</strong></td>
<td>1001 1000</td>
</tr>
<tr>
<td><strong>CDQ – Convert Doubleword to Qword+</strong></td>
<td>1001 1001</td>
</tr>
<tr>
<td><strong>CDQE – RAX, Sign-Extend of EAX</strong></td>
<td>0100 1000 1001 1001</td>
</tr>
<tr>
<td><strong>CLC – Clear Carry Flag</strong></td>
<td>1111 1000</td>
</tr>
<tr>
<td><strong>CLD – Clear Direction Flag</strong></td>
<td>1111 1100</td>
</tr>
<tr>
<td><strong>CLI – Clear Interrupt Flag</strong></td>
<td>1111 1010</td>
</tr>
<tr>
<td><strong>CLTS – Clear Task-Switched Flag in CR0</strong></td>
<td>0000 1111 : 0000 0110</td>
</tr>
<tr>
<td><strong>CMC – Complement Carry Flag</strong></td>
<td>1111 0101</td>
</tr>
<tr>
<td><strong>CMP – Compare Two Operands</strong></td>
<td></td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0100 0R0B 0011 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 with qwordregister2</td>
<td>0100 1R0B 0011 1001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 with register1</td>
<td>0100 0R0B 0011 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister2 with qwordregister1</td>
<td>0100 1R0B 0011 101w : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory with register</td>
<td>0100 0RXB 0011 100w : mod reg r/m</td>
</tr>
<tr>
<td>memory64 with qwordregister</td>
<td>0100 1RXB 0011 1001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register with memory</td>
<td>0100 0RXB 0011 101w : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister with memory64</td>
<td>0100 1RXB 0011 101w1 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate with register</td>
<td>0100 000B 1000 00sw : 11 1111 reg : imm</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate32 with qwordregister</td>
<td>0100 100B 1000 0001 : 11 111 qwordreg : imm64</td>
</tr>
<tr>
<td>immediate with AL, AX, or EAX</td>
<td>0011 110w : imm</td>
</tr>
<tr>
<td>immediate32 with RAX</td>
<td>0100 1000 0111 1101 : imm32</td>
</tr>
<tr>
<td>immediate with memory</td>
<td>0100 00XB 1000 00sw : mod 111 r/m : imm</td>
</tr>
<tr>
<td>immediate32 with memory64</td>
<td>0100 1RXB 1000 0001 : mod 111 r/m : imm64</td>
</tr>
<tr>
<td>immediate8 with memory64</td>
<td>0100 1RXB 1000 0011 : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>CMPS/CMPSB/CMPSW/CMPSD/CMPSQ – Compare String Operands</td>
<td></td>
</tr>
<tr>
<td>compare string operands [ X at DS:(E)SI with Y at ES:(E)DI ]</td>
<td>1010 011w</td>
</tr>
<tr>
<td>qword at address RSI with qword at address RDI</td>
<td>0100 1000 1010 0111</td>
</tr>
<tr>
<td>CMPXCHG – Compare and Exchange</td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>byteregister1, byteregister2</td>
<td>0100 000B 0000 1111 : 1011 0000 : 11 bytereg2 reg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 100B 0000 1111 : 1011 0001 : 11 qwordreg2 reg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 000w : mod reg r/m</td>
</tr>
<tr>
<td>memory8, byteregister</td>
<td>0100 00XB 0000 1111 : 1011 0000 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64, qwordregister</td>
<td>0100 10XB 0000 1111 : 1011 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>CPUID – CPU Identification</td>
<td>0000 1111 : 1010 0010</td>
</tr>
<tr>
<td>CQO – Sign-Extend RAX</td>
<td>0100 1000 1001 1001</td>
</tr>
<tr>
<td>CWD – Convert Word to Doubleword</td>
<td>1001 1000</td>
</tr>
<tr>
<td>CWDE – Convert Word to Doubleword</td>
<td>1001 1000</td>
</tr>
<tr>
<td>DEC – Decrement by 1</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0100 000B 1111 111w : 11 001 reg</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 100B 1111 1111 : 11 001 qwordreg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB 1111 111w : mod 001 r/m</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 10XB 1111 1111 : mod 001 r/m</td>
</tr>
<tr>
<td>DIV – Unsigned Divide</td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>0100 000B 1111 011w : 11 110 reg</td>
</tr>
<tr>
<td>Divide RDX:RAX by qwordregister</td>
<td>0100 100B 1111 0111 : 11 110 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>0100 00XB 1111 011w : mod 110 r/m</td>
</tr>
<tr>
<td>Divide RDX:RAX by memory64</td>
<td>0100 10XB 1111 0111 : mod 110 r/m</td>
</tr>
<tr>
<td>ENTER – Make Stack Frame for High Level Procedure</td>
<td>1100 1000 : 16-bit displacement : 8-bit level (L)</td>
</tr>
<tr>
<td>HLT – Halt</td>
<td>1111 0100</td>
</tr>
<tr>
<td>IDIV – Signed Divide</td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>0100 000B 1111 111w : 11 111 reg</td>
</tr>
<tr>
<td>RDX:RAX by qwordregister</td>
<td>0100 100B 1111 0111 : 11 111 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>0100 00XB 1111 111w : mod 111 r/m</td>
</tr>
<tr>
<td>RDX:RAX by memory64</td>
<td>0100 10XB 1111 1111 : mod 111 r/m</td>
</tr>
<tr>
<td>IMUL – Signed Multiply</td>
<td></td>
</tr>
</tbody>
</table>
Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL, AX, or EAX with register</td>
<td>0100 000B 1111 011w : 11 101 reg</td>
</tr>
<tr>
<td>RDX:RAX ← RAX with qwordregister</td>
<td>0100 100B 1111 0111 : 11 101 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>0100 00Xb 1111 011w : mod 101 r/m</td>
</tr>
<tr>
<td>RDX:RAX ← RAX with memory64</td>
<td>0100 10Xb 1111 0111 : mod 101 r/m</td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0000 1111 : 1010 1111 : 11 : reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 ← qwordregister1 with qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1010 1111 : 11 : qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register with memory</td>
<td>0100 0RXb 0000 1111 : 1010 1111 : mod r/m</td>
</tr>
<tr>
<td>qwordregister ← qwordregister with memory64</td>
<td>0100 1RXb 0000 1111 : 1010 1111 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register1 with immediate to register2</td>
<td>0100 0R0B 0110 10s1 : 11 reg1 reg2 : imm</td>
</tr>
<tr>
<td>qwordregister1 ← qwordregister2 with sign-extended</td>
<td>0100 1R0B 0110 1011 : 11 qwordreg1 qwordreg2 : imm8</td>
</tr>
<tr>
<td>immediate8</td>
<td></td>
</tr>
<tr>
<td>qwordregister1 ← qwordregister2 with immediate32</td>
<td>0100 1R0B 0110 1001 : 11 qwordreg1 qwordreg2 : imm32</td>
</tr>
<tr>
<td>memory with immediate to register</td>
<td>0100 0RXb 0110 10s1 : mod reg r/m : imm</td>
</tr>
<tr>
<td>qwordregister ← memory64 with sign-extended immediate8</td>
<td>0100 1RXb 0110 1011 : mod qwordreg r/m : imm8</td>
</tr>
<tr>
<td>qwordregister ← memory64 with immediate32</td>
<td>0100 1RXb 0110 1001 : mod qwordreg r/m : imm32</td>
</tr>
</tbody>
</table>

IN – Input From Port

| fixed port                                          | 1110 010w : port number                       |
| variable port                                       | 1110 110w                                    |

INC – Increment by 1

| reg                                                | 0100 000B 1111 111w : 11 000 reg                |
| qwordreg                                           | 0100 100B 1111 1111 : 11 000 qwordreg           |
| memory                                             | 0100 00Xb 1111 111w : mod 000 r/m                |
| memory64                                           | 0100 10Xb 1111 1111 : mod 000 r/m                |

INT – Input from DX Port

| INT n – Interrupt Type n                           | 1100 1101 : type                              |
| INT – Single-Step Interrupt 3                      | 1100 1100                                    |
| INTO – Interrupt 4 on Overflow                      | 1100 1110                                    |

INVD – Invalidate Cache

| 0000 1111 : 0000 1000                                |

INVLPG – Invalidate TLB Entry

| 0000 1111 : 0000 0001 : mod 111 r/m                  |

INVP CID – Invalidate Process-Context Identifier

| 0110 0110:0000 1111:0011 1000:1000 0010: mod r/m       |

IRETO – Interrupt Return

| 1100 1111                                           |

Jcc – Jump if Condition is Met

| 8-bit displacement                                  | 0111 ttn : 8-bit displacement                  |
| displacements (excluding 16-bit relative offsets)   | 0000 1111 : 1000 ttn : displacement32           |

JCXZ/JECXZ – Jump on CX/ECX Zero

| Address-size prefix differentiates JCXZ and JECXZ   | 1110 0011 : 8-bit displacement                  |

JMP – Unconditional Jump (to same segment)

| short                                              | 1110 1011 : 8-bit displacement                  |
Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct</td>
<td>1110 1001 : displacement32</td>
</tr>
<tr>
<td>register indirect</td>
<td>0100 W00B™ : 1111 1111 : 11 100 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>0100 W0XB™ : 1111 1111 : mod 100 r/m</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JMP – Unconditional Jump (to other segment)</strong></td>
<td></td>
</tr>
<tr>
<td>indirect intersegment</td>
<td>0100 00XB : 1111 1111 : mod 101 r/m</td>
</tr>
<tr>
<td>64-bit indirect intersegment</td>
<td>0100 10XB : 1111 1111 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>LAR – Load Access Rights Byte</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0100 0R0B : 0000 1111 : 0000 0010 : 11 reg1 reg2</td>
</tr>
<tr>
<td>from dwordregister to qwordregister, masked by 00FxFF00H</td>
<td>0100 WR0B : 0000 1111 : 0000 0010 : 11 qwordreg1 dwordreg2</td>
</tr>
<tr>
<td>from memory</td>
<td>0100 0RXB : 0000 1111 : 0000 0010 : mod reg r/m</td>
</tr>
<tr>
<td>from memory32 to qwordregister, masked by 00FxFF00H</td>
<td>0100 WRXB 0000 1111 : 0000 0010 : mod r/m</td>
</tr>
<tr>
<td><strong>LEA – Load Effective Address</strong></td>
<td></td>
</tr>
<tr>
<td>in wordregister/dwordregister</td>
<td>0100 0RXB : 1000 1101 : mod A reg r/m</td>
</tr>
<tr>
<td>in qwordregister</td>
<td>0100 1RXB : 1000 1101 : mod A qwordreg r/m</td>
</tr>
<tr>
<td><strong>LEAVE – High Level Procedure Exit</strong></td>
<td>1100 1001</td>
</tr>
<tr>
<td><strong>LFS – Load Pointer to FS</strong></td>
<td></td>
</tr>
<tr>
<td>FSr16/r32 with far pointer from memory</td>
<td>0100 0RXB : 0000 1111 : 1011 0100 : modA reg r/m</td>
</tr>
<tr>
<td>FSr64 with far pointer from memory</td>
<td>0100 1RXB : 0000 1111 : 1011 0100 : modA qwordreg r/m</td>
</tr>
<tr>
<td><strong>LGDT – Load Global Descriptor Table Register</strong></td>
<td>0100 10XB : 0000 1111 : 0000 0001 : modA 010 r/m</td>
</tr>
<tr>
<td><strong>LGS – Load Pointer to GS</strong></td>
<td></td>
</tr>
<tr>
<td>GSr16/r32 with far pointer from memory</td>
<td>0100 0RXB : 0000 1111 : 1011 0101 : modA reg r/m</td>
</tr>
<tr>
<td>GSr64 with far pointer from memory</td>
<td>0100 1RXB : 0000 1111 : 1011 0101 : modA qwordreg r/m</td>
</tr>
<tr>
<td><strong>LIDT – Load Interrupt Descriptor Table Register</strong></td>
<td>0100 10XB : 0000 1111 : 0000 0001 : modA 011 r/m</td>
</tr>
<tr>
<td><strong>LLDT – Load Local Descriptor Table Register</strong></td>
<td></td>
</tr>
<tr>
<td>LDTR from register</td>
<td>0100 000B : 0000 1111 : 0000 0000 : 11 010 reg</td>
</tr>
<tr>
<td>LDTR from memory</td>
<td>0100 00XB : 0000 1111 : 0000 0000 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>LMSW – Load Machine Status Word</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0100 000B : 0000 1111 : 0000 0001 : 11 110 reg</td>
</tr>
<tr>
<td>from memory</td>
<td>0100 00XB : 0000 1111 : 0000 0001 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>LOCK – Assert LOCK# Signal Prefix</strong></td>
<td>1111 0000</td>
</tr>
<tr>
<td><strong>LODS/LODSB/LODSW/LODSD/LODSQ – Load String Operand</strong></td>
<td></td>
</tr>
<tr>
<td>at DS:(E)SI to AL/EAX/EAX</td>
<td>1010 110w</td>
</tr>
<tr>
<td>at (R)SI to RAX</td>
<td>0100 1000 1010 1101</td>
</tr>
<tr>
<td><strong>LOOP – Loop Count</strong></td>
<td></td>
</tr>
<tr>
<td>if count ≠ 0, 8-bit displacement</td>
<td>1110 0010</td>
</tr>
<tr>
<td>if count ≠ 0, RIP + 8-bit displacement sign-extended to 64-bits</td>
<td>0100 1000 1110 0010</td>
</tr>
<tr>
<td><strong>LOOPE – Loop Count while Zero/Equal</strong></td>
<td></td>
</tr>
<tr>
<td>if count ≠ 0 &amp; ZF = 1, 8-bit displacement</td>
<td>1110 0001</td>
</tr>
<tr>
<td>if count ≠ 0 &amp; ZF = 1, RIP + 8-bit displacement sign-extended to 64-bits</td>
<td>0100 1000 1110 0001</td>
</tr>
</tbody>
</table>
# Instruction Formats and Encodings

## Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOOPNE/LOOPNZ – Loop Count while not Zero/Equal</strong></td>
<td></td>
</tr>
<tr>
<td>If count ≠ 0 &amp; ZF = 0, 8-bit displacement</td>
<td>1110 0000</td>
</tr>
<tr>
<td>If count ≠ 0 &amp; ZF = 0, RIP + 8-bit displacement sign-extended to 64-bits</td>
<td>0100 1000 1110 0000</td>
</tr>
<tr>
<td><strong>LSL – Load Segment Limit</strong></td>
<td></td>
</tr>
<tr>
<td>From register</td>
<td>0000 1111 : 0000 0011 : 11 reg1 reg2</td>
</tr>
<tr>
<td>From qwordregister</td>
<td>0100 1R00 0000 1111 : 0000 0111 : 11 qwordreg1 reg2</td>
</tr>
<tr>
<td>From memory16</td>
<td>0000 1111 : 0000 0111 : mod reg r/m</td>
</tr>
<tr>
<td>From memory64</td>
<td>0100 1RXB 0000 1111 : 0000 0111 : mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>LSS – Load Pointer to SS</strong></td>
<td></td>
</tr>
<tr>
<td>SS:r16/r32 with far pointer from memory</td>
<td>0100 0RXB 0000 1111 : 1011 0010 : modA reg r/m</td>
</tr>
<tr>
<td>SS:r64 with far pointer from memory</td>
<td>0100 1WXB 0000 1111 : 1011 0010 : modA qwordreg r/m</td>
</tr>
<tr>
<td><strong>LTR – Load Task Register</strong></td>
<td></td>
</tr>
<tr>
<td>From register</td>
<td>0100 0R00 : 0000 1111 : 0000 0000 : 11 011 reg</td>
</tr>
<tr>
<td>From memory</td>
<td>0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>MOV – Move Data</strong></td>
<td></td>
</tr>
<tr>
<td>Register1 to register2</td>
<td>0100 0R0B : 1000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>Qwordregister1 to qwordregister2</td>
<td>0100 1R0B 1000 1001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>Register2 to register1</td>
<td>0100 0R0B : 1000 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>Qwordregister2 to qwordregister1</td>
<td>0100 1R0B 1000 1011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>Memory to reg</td>
<td>0100 0RXB : 1000 101w : mod reg r/m</td>
</tr>
<tr>
<td>Memory64 to qwordregister</td>
<td>0100 1RXB 1000 1011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>Reg to memory</td>
<td>0100 0RXB : 1000 100w : mod reg r/m</td>
</tr>
<tr>
<td>Qwordregister to memory64</td>
<td>0100 1RXB 1000 1001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>Immediate to register</td>
<td>0100 000B : 1100 011w : 11 000 reg : imm</td>
</tr>
<tr>
<td>Immediate32 to qwordregister (zero extend)</td>
<td>0100 100B 1100 0111 : 11 000 qwordreg : imm32</td>
</tr>
<tr>
<td>Immediate32 to qwordregister (alternate encoding)</td>
<td>0100 000B : 1011 w reg : imm</td>
</tr>
<tr>
<td>Immediate64 to qwordregister (alternate encoding)</td>
<td>0100 100B 1011 1000 reg : imm64</td>
</tr>
<tr>
<td>Immediate to memory</td>
<td>0100 00XB : 1100 011w : mod 000 r/m : imm</td>
</tr>
<tr>
<td>Immediate32 to memory64 (zero extend)</td>
<td>0100 10XB 1100 0111 : 11 000 r/m : imm32</td>
</tr>
<tr>
<td>Memory to AL, AX, or EAX</td>
<td>0100 0000 : 1010 000w : displacement</td>
</tr>
<tr>
<td>Memory64 to RAX</td>
<td>0100 1000 1010 0001 : displacement64</td>
</tr>
<tr>
<td>AL, AX, or EAX to memory</td>
<td>0100 0000 : 1010 001w : displacement</td>
</tr>
<tr>
<td>RAX to memory64</td>
<td>0100 1000 1010 0011 : displacement64</td>
</tr>
<tr>
<td><strong>MOV – Move to/from Control Registers</strong></td>
<td></td>
</tr>
<tr>
<td>CR0-CR4 from register</td>
<td>0100 0R0B : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#)</td>
</tr>
<tr>
<td>CRx from qwordregister</td>
<td>0100 1R0B : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#)</td>
</tr>
<tr>
<td>Register from CR0-CR4</td>
<td>0100 0R0B : 0000 1111 : 0010 0000 : 11 eee reg (eee = CR#)</td>
</tr>
</tbody>
</table>
## Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>qwordregister from CRx</td>
<td>0100 100B 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)</td>
</tr>
<tr>
<td>MOV – Move to/from Debug Registers</td>
<td></td>
</tr>
<tr>
<td>DR0-DR7 from register</td>
<td>0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)</td>
</tr>
<tr>
<td>DR0-DR7 from quadregister</td>
<td>0100 100B 0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)</td>
</tr>
<tr>
<td>register from DR0-DR7</td>
<td>0000 1111 : 0010 0001 : 11 eee reg (eee = DR#)</td>
</tr>
<tr>
<td>quadregister from DR0-DR7</td>
<td>0100 100B 0000 1111 : 0010 0001 : 11 eee quadreg (eee = DR#)</td>
</tr>
<tr>
<td>MOV – Move to/from Segment Registers</td>
<td></td>
</tr>
<tr>
<td>register to segment register</td>
<td>0100 W00Bw : 1000 1110 : 11 sreg reg</td>
</tr>
<tr>
<td>register to SS</td>
<td>0100 000B : 1000 1110 : 11 sreg reg</td>
</tr>
<tr>
<td>memory to segment register</td>
<td>0100 00XB : 1000 1110 : mod sreg r/m</td>
</tr>
<tr>
<td>memory64 to segment register (lower 16 bits)</td>
<td>0100 10XB 1000 1110 : mod sreg r/m</td>
</tr>
<tr>
<td>memory to SS</td>
<td>0100 00XB : 1000 1110 : mod sreg r/m</td>
</tr>
<tr>
<td>segment register to register</td>
<td>0100 000B : 1000 1100 : 11 sreg reg</td>
</tr>
<tr>
<td>segment register to quadregister (zero extended)</td>
<td>0100 100B 1000 1100 : 11 sreg qwordreg</td>
</tr>
<tr>
<td>segment register to memory</td>
<td>0100 00XB : 1000 1100 : mod sreg r/m</td>
</tr>
<tr>
<td>segment register to memory64 (zero extended)</td>
<td>0100 10XB 1000 1100 : mod sreg3 r/m</td>
</tr>
<tr>
<td>MOVBE – Move data after swapping bytes</td>
<td></td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 1111 : 0011 100010:0000 : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to quadregister</td>
<td>0100 1RXB : 0000 1111 : 0011 100010:0000 : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0000 1111 : 0011 100010:0001 : mod reg r/m</td>
</tr>
<tr>
<td>quadregister to memory64</td>
<td>0100 1RXB : 0000 1111 : 0011 100010:0001 : mod reg r/m</td>
</tr>
<tr>
<td>MOVS/MOVSB/MOVSW/MOVSD/MOVSDQ – Move Data from String to String</td>
<td></td>
</tr>
<tr>
<td>Move data from string to string</td>
<td>1010 010w</td>
</tr>
<tr>
<td>Move data from string to string (qword)</td>
<td>0100 1000 1010 0101</td>
</tr>
<tr>
<td>MOVSX/MOVSXD – Move with Sign-Extend</td>
<td></td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0000 1111 : 1011 1111w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 to quadregister1 (sign-extend)</td>
<td>0100 1R0B 0000 1111 : 1011 1110 : 11 quadreg1 bytereg2</td>
</tr>
<tr>
<td>wordregister2 to quadregister1</td>
<td>0100 1R0B 0000 1111 : 1011 1111 : 11 quadreg1 wordreg2</td>
</tr>
<tr>
<td>dwordregister2 to quadregister1</td>
<td>0100 1R0B 0110 0011 : 11 quadreg1 dwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 1111 : 1011 1111w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 to quadregister (sign-extend)</td>
<td>0100 1RXB 0000 1111 : 1011 1110 : mod quadreg r/m</td>
</tr>
<tr>
<td>memory16 to quadregister</td>
<td>0100 1RXB 0000 1111 : 1011 1111 : mod quadreg r/m</td>
</tr>
<tr>
<td>memory32 to quadregister</td>
<td>0100 1RXB 0110 0011 : mod quadreg r/m</td>
</tr>
<tr>
<td>MOVZX – Move with Zero-Extend</td>
<td></td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0000 1111 : 1011 0111w : 11 reg1 reg2</td>
</tr>
<tr>
<td>dwordregister2 to quadregister1</td>
<td>0100 1R0B 0000 1111 : 1011 0111 : 11 quadreg1 dwordreg2</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 1111 : 1011 011w : mod reg r/m</td>
</tr>
<tr>
<td>memory32 to qwordregister</td>
<td>0100 1RXB 0000 1111 : 1011 0111 : mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>MUL – Unsigned Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>0100 000B : 1111 011w : 11 100 reg</td>
</tr>
<tr>
<td>RAX with qwordregister (to RDX:RAX)</td>
<td>0100 100B 1111 0111 : 11 100 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>0100 00XB 1111 011w : mod 100 r/m</td>
</tr>
<tr>
<td>RAX with memory64 (to RDX:RAX)</td>
<td>0100 10XB 1111 0111 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>NEG – Two’s Complement Negation</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0100 000B : 1111 011w : 11 011 reg</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 100B 1111 0111 : 11 010 qwordreg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB : 1111 011w : mod 010 r/m</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 10XB 1111 0111 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>NOP – No Operation</strong></td>
<td>1001 0000</td>
</tr>
<tr>
<td><strong>NOT – One’s Complement Negation</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0100 000B : 1111 011w : 11 010 reg</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 000B 1111 0111 : 11 010 qwordreg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB : 1111 011w : mod 010 r/m</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 1RXB 1111 0111 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>OR – Logical Inclusive OR</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister1 to byteregister2</td>
<td>0100 0R0B 0000 1000 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0000 1001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 to byteregister1</td>
<td>0100 0R0B 0000 1010 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister2 to qwordregister1</td>
<td>0100 1R0B 0000 1011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 101w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 to byteregister</td>
<td>0100 0RXB 0000 1010 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory8 to qwordregister</td>
<td>0100 0RXB 0000 1011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 100w : mod reg r/m</td>
</tr>
<tr>
<td>byteregister to memory8</td>
<td>0100 0RXB 0000 1000 : mod bytereg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB 0000 1001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 001 reg : imm</td>
</tr>
<tr>
<td>immediate8 to byteregister</td>
<td>0100 000B 1000 0000 : 11 001 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 000B 1000 0001 : 11 001 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate8 to qwordregister</td>
<td>0100 000B 1000 0011 : 11 001 qwordreg : imm8</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 110w : imm</td>
</tr>
<tr>
<td>immediate64 to RAX</td>
<td>0100 1000 0000 1101 : imm64</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 001 r/m : imm</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate8 to memory8</td>
<td>0100 00XB 1000 0000 : mod 001 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 00XB 1000 0001 : mod 001 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 00XB 1000 0011 : mod 001 r/m : imm8</td>
</tr>
<tr>
<td><strong>OUT – Output to Port</strong></td>
<td></td>
</tr>
<tr>
<td>fixed port</td>
<td>1110 011w : port number</td>
</tr>
<tr>
<td>variable port</td>
<td>1110 111w</td>
</tr>
<tr>
<td><strong>OUTS – Output to DX Port</strong></td>
<td></td>
</tr>
<tr>
<td>output to DX Port</td>
<td>0110 111w</td>
</tr>
<tr>
<td><strong>POP – Pop a Value from the Stack</strong></td>
<td></td>
</tr>
<tr>
<td>wordregister</td>
<td>0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 W00B5 : 1000 1111 : 11 000 reg64</td>
</tr>
<tr>
<td>wordregister (alternate encoding)</td>
<td>0101 0101 : 0100 000B : 0101 1 reg16</td>
</tr>
<tr>
<td>qwordregister (alternate encoding)</td>
<td>0100 W00B : 0101 1 reg64</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 W0XB : 1000 1111 : mod 000 r/m</td>
</tr>
<tr>
<td>memory16</td>
<td>0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m</td>
</tr>
<tr>
<td><strong>POP – Pop a Segment Register from the Stack</strong> (Note: CS cannot be sreg2 in this usage.)</td>
<td></td>
</tr>
<tr>
<td>segment register FS, GS</td>
<td>0000 1111 : 10 sreg3 001</td>
</tr>
<tr>
<td><strong>POPF/POPFQ – Pop Stack into FLAGS/RFLAGS Register</strong></td>
<td></td>
</tr>
<tr>
<td>pop stack to FLAGS register</td>
<td>0101 0101 : 1001 1101</td>
</tr>
<tr>
<td>pop Stack to RFLAGS register</td>
<td>0100 1000 1001 1101</td>
</tr>
<tr>
<td><strong>PUSH – Push Operand onto the Stack</strong></td>
<td></td>
</tr>
<tr>
<td>wordregister</td>
<td>0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 W00B5 : 1111 1111 : 11 110 reg64</td>
</tr>
<tr>
<td>wordregister (alternate encoding)</td>
<td>0101 0101 : 0100 000B : 0101 0 reg16</td>
</tr>
<tr>
<td>qwordregister (alternate encoding)</td>
<td>0100 W00B5 : 0101 0 reg64</td>
</tr>
<tr>
<td>memory16</td>
<td>0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 W00B5 : 1111 1111 : mod 110 r/m</td>
</tr>
<tr>
<td>immediate8</td>
<td>0110 1010 : imm8</td>
</tr>
<tr>
<td>immediate16</td>
<td>0101 0101 : 0110 1000 : imm16</td>
</tr>
<tr>
<td>immediate64</td>
<td>0110 1000 : imm64</td>
</tr>
<tr>
<td><strong>PUSH – Push Segment Register onto the Stack</strong></td>
<td></td>
</tr>
<tr>
<td>segment register FS,GS</td>
<td>0000 1111 : 10 sreg3 000</td>
</tr>
<tr>
<td><strong>PUSHF/PUSHFD – Push Flags Register onto the Stack</strong></td>
<td>1001 1100</td>
</tr>
<tr>
<td><strong>RCL – Rotate thru Carry Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B : 1101 000w : 11 010 reg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 010 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB : 1101 000w : mod 010 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 010 r/m</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register by CL</td>
<td>0100 000B : 1101 001w : 11 010 reg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 010 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB : 1101 001w : mod 010 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 010 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B : 1100 000w : 11 010 reg : imm</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 010 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB : 1100 000w : mod 010 r/m : imm</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 010 r/m : imm8</td>
</tr>
<tr>
<td><strong>RCR – Rotate thru Carry Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B : 1101 000w : 11 011 reg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 011 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB : 1101 000w : mod 011 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 011 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B : 1101 001w : 11 011 reg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0010 : 11 011 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB : 1101 001w : mod 011 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 011 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B : 1100 000w : 11 011 reg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 011 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB : 1100 000w : mod 011 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 011 r/m : imm8</td>
</tr>
<tr>
<td><strong>RDMSR – Read from Model-Specific Register</strong></td>
<td></td>
</tr>
<tr>
<td>load ECX-specified register into EDX:EAX</td>
<td>0000 1111 : 0011 0010</td>
</tr>
<tr>
<td><strong>RDPMC – Read Performance Monitoring Counters</strong></td>
<td></td>
</tr>
<tr>
<td>load ECX-specified performance counter into EDX:EAX</td>
<td>0000 1111 : 0011 0011</td>
</tr>
<tr>
<td><strong>RDTSC – Read Time-Stamp Counter</strong></td>
<td></td>
</tr>
<tr>
<td>read time-stamp counter into EDX:EAX</td>
<td>0000 1111 : 0011 0001</td>
</tr>
<tr>
<td><strong>RDTSCP – Read Time-Stamp Counter and Processor ID</strong></td>
<td></td>
</tr>
<tr>
<td>read time-stamp counter and processor ID into EDX:EAX</td>
<td>0000 1111 : 0000 0001 : 1111 1001</td>
</tr>
<tr>
<td><strong>REP INS – Input String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REP LODS – Load String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REP MOV5 – Move String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REP OUTS – Output String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REP STOS – Store String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REPE CMPS – Compare String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REPE SCAS – Scan String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REPNE CMPS – Compare String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>REPNE SCAS – Scan String</strong></td>
<td></td>
</tr>
<tr>
<td><strong>RET – Return from Procedure (to same segment)</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>no argument</td>
<td>1100 0011</td>
</tr>
<tr>
<td>adding immediate to SP</td>
<td>1100 0010: 16-bit displacement</td>
</tr>
<tr>
<td>RET – Return from Procedure (to other segment)</td>
<td></td>
</tr>
<tr>
<td>intersegment</td>
<td>1100 1011</td>
</tr>
<tr>
<td>adding immediate to SP</td>
<td>1100 1010: 16-bit displacement</td>
</tr>
<tr>
<td>ROL – Rotate Left</td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 000 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 000 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 000 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB 1101 000w : mod 000 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 00XB 1101 0000 : mod 000 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 000 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 000 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 000 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 000 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 1101 001w : mod 000 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 00XB 1101 0010 : mod 000 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 000 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 000 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 000w : 11 000 bytereg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 000 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 000 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 000w : mod 000 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 000 r/m : imm8</td>
</tr>
<tr>
<td>ROR – Rotate Right</td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 001 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 001 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 001 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB 1101 000w : mod 001 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 00XB 1101 0000 : mod 001 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 001 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 001 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 001 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 001 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 1101 001w : mod 001 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 00XB 1101 0010 : mod 001 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 001 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B 1100 000w : 11 001 reg : imm8</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 001 reg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 001 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 1100 000w : mod 001 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 001 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 001 r/m : imm8</td>
</tr>
</tbody>
</table>

**RSM - Resume from System Management Mode**

0000 1111 : 1010 1010

**SAL - Shift Arithmetic Left**

same instruction as SHL

**SAR - Shift Arithmetic Right**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 111 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 111 byteregreg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 111 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB 1101 000w : mod 111 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 00XB 1101 0000 : mod 111 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 111 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 111 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 111 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 111 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 1101 001w : mod 111 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 00XB 1101 0010 : mod 111 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 111 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B 1100 000w : 11 111 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 111 bytereg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 111 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 1100 000w : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 111 r/m : imm8</td>
</tr>
</tbody>
</table>

**SBB - Integer Subtraction with Borrow**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B 0001 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister1 to byteregister2</td>
<td>0100 0R0B 0001 1000 : 11 byteregreg</td>
</tr>
<tr>
<td>quadregister1 to quadregister2</td>
<td>0100 1R0B 0001 1001 : 11 quadreg1 quadreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B 0001 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 to byteregister1</td>
<td>0100 0R0B 0001 1010 : 11 reg1 bytereg</td>
</tr>
<tr>
<td>byteregister2 to byteregister1</td>
<td>0100 1R0B 0001 1011 : 11 reg1 bytereg</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB 0001 101w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 to byteregister</td>
<td>0100 0RXB 0001 1010 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64 to byteregister</td>
<td>0100 1RXB 0001 1011 : mod quadreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB 0001 100w : mod reg r/m</td>
</tr>
<tr>
<td>byteregister to memory8</td>
<td>0100 0RXB 0001 1000 : mod reg r/m</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>quadregister to memory64</td>
<td>0100 1RXB 0001 1001 : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 000B 1000 00sw : 11 011 reg : imm</td>
</tr>
<tr>
<td>immediate8 to byteregister</td>
<td>0100 000B 1000 0000 : 11 011 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 100B 1000 0001 : 11 011 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate8 to qwordregister</td>
<td>0100 100B 1000 0011 : 11 011 qwordreg : imm8</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0100 000B 0001 110w : imm</td>
</tr>
<tr>
<td>immediate32 to RAL</td>
<td>0100 1000 0001 1101 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB 1000 00sw : mod 011 r/m : imm</td>
</tr>
<tr>
<td>immediate8 to memory8</td>
<td>0100 00XB 1000 0000 : mod 011 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB 1000 0001 : mod 011 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB 1000 0011 : mod 011 r/m : imm8</td>
</tr>
</tbody>
</table>

### SCAS/SCASB/SCASW/SCASD – Scan String

- scan string | 1010 111w |
- scan string (compare AL with byte at RDI) | 0100 1000 1010 1110 |
- scan string (compare RAX with qword at RDI) | 0100 1000 1010 1111 |

### SETcc – Byte Set on Condition

- register | 0100 000B 0000 1111 : 1001 tttt : 11 000 reg |
- register | 0100 0000 0000 1111 : 1001 tttt : 11 000 reg |
- memory | 0100 00XB 0000 1111 : 1001 tttt : mod 000 r/m |
- memory | 0100 0000 0000 1111 : 1001 tttt : mod 000 r/m |

### SGDT – Store Global Descriptor Table Register

- SGDT | 0000 1111 : 0000 0001 : modA 000 r/m |

### SHL – Shift Left

- register by 1 | 0100 000B 1101 000w : 11 100 reg |
- byteregister by 1 | 0100 000B 1101 0000 : 11 100 bytereg |
- qwordregister by 1 | 0100 100B 1101 0001 : 11 100 qwordreg |
- memory by 1 | 0100 00XB 1101 000w : mod 100 r/m |
- memory8 by 1 | 0100 00XB 1101 0000 : mod 100 r/m |
- memory64 by 1 | 0100 10XB 1101 0001 : mod 100 r/m |
- register by CL | 0100 000B 1101 001w : 11 100 reg |
- byteregister by CL | 0100 000B 1101 0010 : 11 100 bytereg |
- qwordregister by CL | 0100 100B 1101 0011 : 11 100 qwordreg |
- memory by CL | 0100 00XB 1101 001w : mod 100 r/m |
- memory8 by CL | 0100 00XB 1101 0010 : mod 100 r/m |
- memory64 by CL | 0100 10XB 1101 0011 : mod 100 r/m |
- register by immediate count | 0100 000B 1100 000w : 11 100 reg : imm8 |
- byteregister by immediate count | 0100 000B 1100 0000 : 11 100 bytereg : imm8 |
- quadregister by immediate count | 0100 100B 1100 0001 : 11 100 quadreg : imm8 |
- memory by immediate count | 0100 00XB 1100 000w : mod 100 r/m : imm8 |
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 100 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 100 r/m : imm8</td>
</tr>
<tr>
<td><strong>SHLD – Double Precision Shift Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 0R0B 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate8</td>
<td>0100 1R0B 0000 1111 : 1010 0100 : 11 qwordreg2 qwordreg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 0RXB 0000 1111 : 1010 0100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate8</td>
<td>0100 1RXB 0000 1111 : 1010 0100 : mod qwordreg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 0R0B 0000 1111 : 1010 0101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>quadregister by CL</td>
<td>0100 1R0B 0000 1111 : 1010 0101 : 11 quadreg2 quadreg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 0000 1111 : 1010 0101 : mod reg r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 1RXB 0000 1111 : 1010 0101 : mod quadreg r/m</td>
</tr>
<tr>
<td><strong>SHR – Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 101 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 101 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 101 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB 1101 000w : mod 101 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 00XB 1101 0000 : mod 101 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 101 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 101 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 101 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 101 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 1101 001w : mod 101 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 00XB 1101 0010 : mod 101 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 101 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B 1100 000w : 11 101 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 101 bytereg</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 101 reg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 1100 000w : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td><strong>SHRD – Double Precision Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 0R0B 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate8</td>
<td>0100 1R0B 0000 1111 : 1010 1100 : 11 qwordreg2 qwordreg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 0000 1111 : 1010 1100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate8</td>
<td>0100 1RXB 0000 1111 : 1010 1100 : mod qwordreg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 0000 1111 : 1010 1101 : 11 reg2 reg1</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>qwordregister by CL</td>
<td>0100 1R0B 0000 1111 : 1010 1101 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0000 1111 : 1010 1101 : mod reg r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 1RXB 0000 1111 : 1010 1101 : mod qwordreg r/m</td>
</tr>
<tr>
<td>SIDT – Store Interrupt Descriptor Table Register</td>
<td>0000 1111 : 0000 0001 : mod^A 001 r/m</td>
</tr>
<tr>
<td>SLDT – Store Local Descriptor Table Register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 000 reg</td>
</tr>
<tr>
<td></td>
<td>0100 00XB 0000 1111 : 0000 0000 : mod 000 r/m</td>
</tr>
<tr>
<td>SMSW – Store Machine Status Word</td>
<td>1111 1001</td>
</tr>
<tr>
<td></td>
<td>1111 1101</td>
</tr>
<tr>
<td></td>
<td>1111 1011</td>
</tr>
<tr>
<td>STOS/STOSB/STOSw/STOSD/STOSQ – Store String Data</td>
<td>1010 101w</td>
</tr>
<tr>
<td>store string data</td>
<td>0100 1000 1010 1011</td>
</tr>
<tr>
<td>to register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 001 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0100 00XB 0000 1111 : 0000 0000 : mod 001 r/m</td>
</tr>
<tr>
<td>STR – Store Task Register</td>
<td>1111 1001</td>
</tr>
<tr>
<td></td>
<td>1111 1011</td>
</tr>
<tr>
<td>STC – Set Carry Flag</td>
<td>1111 1001</td>
</tr>
<tr>
<td>STD – Set Direction Flag</td>
<td>1111 1101</td>
</tr>
<tr>
<td>STI – Set Interrupt Flag</td>
<td>1111 1011</td>
</tr>
<tr>
<td>SUB – Integer Subtraction</td>
<td>0100 0R0B 0010 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register1 from register2</td>
<td>0100 0R0B 0010 1000 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>byteregister1 from byteregister2</td>
<td>0100 1R0B 0010 1000 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 from register1</td>
<td>0100 0R0B 0010 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 from byteregister1</td>
<td>0100 0R0B 0010 1010 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister2 from qwordregister1</td>
<td>0100 1R0B 0010 1011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory from register</td>
<td>0100 00XB 0010 101w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 from byteregister</td>
<td>0100 0RXB 0010 1010 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64 from qwordregister</td>
<td>0100 1RXB 0010 1011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register from memory</td>
<td>0100 0RXB 0010 100w : mod reg r/m</td>
</tr>
<tr>
<td>byteregister from memory8</td>
<td>0100 0RXB 0010 1000 : mod bytereg r/m</td>
</tr>
<tr>
<td>qwordregister from memory8</td>
<td>0100 1RXB 0010 1000 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate from register</td>
<td>0100 000B 1000 00sw : 11 101 reg : imm</td>
</tr>
<tr>
<td>immediate8 from byteregister</td>
<td>0100 000B 1000 0000 : 11 101 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 from qwordregister</td>
<td>0100 100B 1000 0001 : 11 101 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate8 from qwordregister</td>
<td>0100 100B 1000 0011 : 11 101 qwordreg : imm8</td>
</tr>
<tr>
<td>immediate from AL, AX, or EAX</td>
<td>0100 000B 0010 110w : imm</td>
</tr>
<tr>
<td>immediate32 from RAX</td>
<td>0100 1000 0010 1101 : imm32</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate from memory</td>
<td>0100 00XB 1000 00sw : mod 101 r/m : imm</td>
</tr>
<tr>
<td>immediate8 from memory8</td>
<td>0100 00XB 1000 0000 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 from memory64</td>
<td>0100 10XB 1000 0001 : mod 101 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 from memory64</td>
<td>0100 10XB 1000 0011 : mod 101 r/m : imm8</td>
</tr>
</tbody>
</table>

**SwAPGS – Swap GS Base Register**

Exchanges the current GS base register value for value in MSR C0000102H

<table>
<thead>
<tr>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111 0000 0001 1111 1000</td>
</tr>
</tbody>
</table>

**SYSCALL – Fast System Call**

Fast call to privilege level 0 system procedures

<table>
<thead>
<tr>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111 0000 0101</td>
</tr>
</tbody>
</table>

**SYSRET – Return From Fast System Call**

Return from fast system call

<table>
<thead>
<tr>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111 0000 0111</td>
</tr>
</tbody>
</table>

**TEST – Logical Compare**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1 and register2</td>
<td>0100 0R0B 1000 010w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister1 and byteregister2</td>
<td>0100 0R0B 1000 0100 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister1 and qwordregister2</td>
<td>0100 1R0B 1000 0101 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory and register</td>
<td>0100 0R0B 1000 010w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 and byteregister</td>
<td>0100 0RXB 1000 0100 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64 and qwordregister</td>
<td>0100 1RXB 1000 0101 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate and register</td>
<td>0100 000B 1111 011w : 11 000 reg : imm</td>
</tr>
<tr>
<td>immediate8 and byteregister</td>
<td>0100 000B 1111 0110 : 11 000 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 and qwordregister</td>
<td>0100 100B 1111 0111 : 11 000 bytereg : imm8</td>
</tr>
<tr>
<td>immediate and AL, AX, or EAX</td>
<td>0100 000B 1010 100w : imm</td>
</tr>
<tr>
<td>immediate32 and RAX</td>
<td>0100 1000 1010 1001 : imm32</td>
</tr>
<tr>
<td>immediate and memory</td>
<td>0100 00XB 1111 011w : mod 000 r/m : imm</td>
</tr>
<tr>
<td>immediate8 and memory8</td>
<td>0100 1000 1111 0110 : mod 000 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 and memory64</td>
<td>0100 1000 1111 0111 : mod 000 r/m : imm32</td>
</tr>
</tbody>
</table>

**UD2 – Undefined Instruction**

<table>
<thead>
<tr>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 FFFF : 0000 1011</td>
</tr>
</tbody>
</table>

**VERR – Verify a Segment for Reading**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 100 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m</td>
</tr>
</tbody>
</table>

**VERw – Verify a Segment for Writing**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 101 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m</td>
</tr>
</tbody>
</table>

**WAIT – Wait**

<table>
<thead>
<tr>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 1011</td>
</tr>
</tbody>
</table>

**WBINVD – Writeback and Invalidate Data Cache**

<table>
<thead>
<tr>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111 : 0000 1001</td>
</tr>
</tbody>
</table>

**WRMSR – Write to Model-Specific Register**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>write EDX:EAX to ECX specified MSR</td>
<td>0000 1111 : 0011 0000</td>
</tr>
<tr>
<td>write RDX[31:0]:RAX[31:0] to RCX specified MSR</td>
<td>0100 1000 0000 1111 : 0011 0000</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>XADD - Exchange and Add</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1100 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>byteregister1, byteregister2</td>
<td>0100 0R0B 0000 1111 : 1100 0000 : 11 bytereg2 bytereg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 0R0B 0000 1111 : 1100 0001 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1100 000w : mod reg r/m</td>
</tr>
<tr>
<td>memory8,bytereg</td>
<td>0100 1RXB 0000 1111 : 1100 0000 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1100 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>XCHG - Exchange Register/Memory with Register</strong></td>
<td></td>
</tr>
<tr>
<td>register1 with register2</td>
<td>1000 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>AX or EAX with register</td>
<td>1001 0 reg</td>
</tr>
<tr>
<td>memory with register</td>
<td>1000 011w : mod reg r/m</td>
</tr>
<tr>
<td><strong>XLAT/XLATB - Table Look-up Translation</strong></td>
<td></td>
</tr>
<tr>
<td>AL to byte DS[(E)BX + unsigned AL]</td>
<td>1101 0111</td>
</tr>
<tr>
<td>AL to byte DS[(RBX + unsigned AL]</td>
<td>0100 1000 1101 0111</td>
</tr>
<tr>
<td><strong>XOR - Logical Exclusive OR</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 0RXB 0011 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister1 to byteregister2</td>
<td>0100 0RXB 0011 0000 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1RXB 0011 0001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B 0011 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 to byteregister1</td>
<td>0100 0R0B 0011 0010 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister2 to qwordregister1</td>
<td>0100 1R0B 0011 0011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB 0011 001w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 to byteregister</td>
<td>0100 0RXB 0011 0010 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB 0011 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB 0011 000w : mod reg r/m</td>
</tr>
<tr>
<td>byteregister to memory8</td>
<td>0100 0RXB 0011 0000 : mod bytereg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB 0011 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 000B 1000 00sw : 11 110 reg : imm</td>
</tr>
<tr>
<td>immediate8 to byteregister</td>
<td>0100 000B 1000 0000 : 11 110 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 100B 1000 0001 : 11 110 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate8 to qwordregister</td>
<td>0100 100B 1000 0011 : 11 110 qwordreg : imm8</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0100 000B 0011 010w : imm</td>
</tr>
<tr>
<td>immediate to RAX</td>
<td>0100 1000 0011 0101 : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB 1000 00sw : mod 110 r/m : imm</td>
</tr>
<tr>
<td>immediate8 to memory8</td>
<td>0100 00XB 1000 0000 : mod 110 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB 1000 0001 : mod 110 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB 1000 0011 : mod 110 r/m : imm8</td>
</tr>
</tbody>
</table>
The following table shows formats and encodings introduced by the Pentium processor family.

### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix Bytes</td>
<td></td>
</tr>
<tr>
<td>address size</td>
<td>0110 0111</td>
</tr>
<tr>
<td>LOCK</td>
<td>1111 0000</td>
</tr>
<tr>
<td>operand size</td>
<td>0110 0110</td>
</tr>
<tr>
<td>CS segment override</td>
<td>0010 1110</td>
</tr>
<tr>
<td>DS segment override</td>
<td>0011 1110</td>
</tr>
<tr>
<td>ES segment override</td>
<td>0010 0110</td>
</tr>
<tr>
<td>FS segment override</td>
<td>0110 0100</td>
</tr>
<tr>
<td>GS segment override</td>
<td>0110 0101</td>
</tr>
<tr>
<td>SS segment override</td>
<td>0011 0110</td>
</tr>
</tbody>
</table>

### Table B-16. Pentium Processor Family Instruction Formats and Encodings, Non-64-Bit Modes

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPXCHG8B - Compare and Exchange 8 Bytes</td>
<td></td>
</tr>
<tr>
<td>EDX:EAX with memory64</td>
<td>0000 1111 : 1100 0111 : mod 001 r/m</td>
</tr>
</tbody>
</table>

### Table B-17. Pentium Processor Family Instruction Formats and Encodings, 64-Bit Mode

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPXCHG8B/CMPXCHG16B - Compare and Exchange Bytes</td>
<td></td>
</tr>
<tr>
<td>EDX:EAX with memory64</td>
<td>0000 1111 : 1100 0111 : mod 001 r/m</td>
</tr>
<tr>
<td>RDX:RAX with memory128</td>
<td>0100 10XB 0000 1111 : 1100 0111 : mod 001 r/m</td>
</tr>
</tbody>
</table>

### B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS

Non-64-bit mode instruction encodings for MMX Technology, SSE, SSE2, and SSE3 are covered by applying these rules to Table B-19 through Table B-31. Table B-34 lists special encodings (instructions that do not follow the rules below).

1. The REX instruction has no effect:
   - On immediates.
   - If both operands are MMX registers.
   - On MMX registers and XMM registers.
   - If an MMX register is encoded in the reg field of the ModR/M byte.

2. If a memory operand is encoded in the r/m field of the ModR/M byte, REX.X and REX.B may be used for encoding the memory operand.
3. If a general-purpose register is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding and REX.W may be used to encode the 64-bit operand size.

4. If an XMM register operand is encoded in the reg field of the ModR/M byte, REX.R may be used for register encoding. If an XMM register operand is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding.

B.5 MMX INSTRUCTION FORMATS AND ENCODINGS

MMX instructions, except the EMMS instruction, use a format similar to the 2-byte Intel Architecture integer format. Details of subfield encodings within these formats are presented below.

B.5.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-18 shows the encoding of the gg field.

<table>
<thead>
<tr>
<th>gg</th>
<th>Granularity of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Packed Bytes</td>
</tr>
<tr>
<td>01</td>
<td>Packed Words</td>
</tr>
<tr>
<td>10</td>
<td>Packed Doublewords</td>
</tr>
<tr>
<td>11</td>
<td>Quadword</td>
</tr>
</tbody>
</table>

B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg)

When MMX technology registers (mmxreg) are used as operands, they are encoded in the ModR/M byte in the reg field (bits 5, 4, and 3) and/or the R/M field (bits 2, 1, and 0).

If an MMX instruction operates on a general-purpose register (reg), the register is encoded in the R/M field of the ModR/M byte.

B.5.3 MMX Instruction Formats and Encodings Table

Table B-19 shows the formats and encodings of the integer instructions.

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMMS – Empty MMX technology state</td>
<td>0000 1111:01110111</td>
</tr>
<tr>
<td>MOVQ – Move quadword</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:0110 1111:11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:0111 1111:11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mem to mmxreg</td>
<td>0000 1111:0110 1111:mod mmxreg r/m</td>
</tr>
</tbody>
</table>
### Table B-19. MMX Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem from mmxreg</td>
<td>0000 1111:0111 1111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PACKSSDW1 – Pack dword to word data (signed with saturation)</td>
<td>0000 1111:0110 1011: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:0110 1011: mod mmxreg r/m</td>
</tr>
<tr>
<td>PACKSSWB1 – Pack word to byte data (signed with saturation)</td>
<td>0000 1111:0110 0111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:0110 0111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PACKUSWB1 – Pack word to byte data (unsigned with saturation)</td>
<td>0000 1111:0110 0111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:0110 0111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PADD – Add with wrap-around</td>
<td>0000 1111:1111 11gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1111 11gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PADDS – Add signed with saturation</td>
<td>0000 1111:1110 11gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1110 11gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PADDUS – Add unsigned with saturation</td>
<td>0000 1111:1101 11gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1101 11gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PAND – Bitwise And</td>
<td>0000 1111:1101 1011: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1101 1011: mod mmxreg r/m</td>
</tr>
<tr>
<td>PANDN – Bitwise AndNot</td>
<td>0000 1111:1101 1111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1101 1111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PCMPLEQ – Packed compare for equality</td>
<td>0000 1111:0111 01gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg1 with mmxreg2</td>
<td>0000 1111:0111 01gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg with mmxreg</td>
<td>0000 1111:0111 01gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PCMPGT – Packed compare greater (signed)</td>
<td>0000 1111:0110 01gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg1 with mmxreg2</td>
<td>0000 1111:0110 01gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg with mmxreg</td>
<td>0000 1111:0110 01gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PMADDUD – Packed multiply add</td>
<td>0000 1111:1111 0101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1111 0101: mod mmxreg r/m</td>
</tr>
<tr>
<td>PMULHUW – Packed multiplication, store high word (unsigned)</td>
<td>0000 1111:1110 0100: 11 mmxreg1 mmxreg2</td>
</tr>
</tbody>
</table>
### Table B-19. MMX Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111: 1110 0100: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PMULHW</strong> – Packed multiplication, store high word</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:1110 0101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1110 0101: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PMULLW</strong> – Packed multiplication, store low word</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:1101 0101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1101 0101: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>POR</strong> – Bitwise Or</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:1110 1011: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1110 1011: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PSLL2</strong> – Packed shift left logical</td>
<td></td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:1111 00gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg by memory</td>
<td>0000 1111:1111 00gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:0111 00gg: 11 110 mmxreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSRA2</strong> – Packed shift right arithmetic</td>
<td></td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:1111 00gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg by memory</td>
<td>0000 1111:1111 00gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:0111 00gg: 11 100 mmxreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSRL2</strong> – Packed shift right logical</td>
<td></td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:1101 00gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg by memory</td>
<td>0000 1111:1101 00gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:0111 00gg: 11 010 mmxreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSUB</strong> – Subtract with wrap-around</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:1111 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:1111 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PSUBS</strong> – Subtract signed with saturation</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:1110 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:1110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PSUBUS</strong> – Subtract unsigned with saturation</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:1101 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:1101 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PUNPCKH</strong> – Unpack high data to next larger type</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:0110 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:0110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PUNPCKL</strong> – Unpack low data to next larger type</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:0110 00gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:0110 00gg: mod mmxreg r/m</td>
</tr>
</tbody>
</table>
B.6 PROCESSOR EXTENDED STATE INSTRUCTION FORMATS AND ENCODINGS

Table B-20 shows the formats and encodings for several instructions that relate to processor extended state management.

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PXOR – Bitwise Xor</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:1110 1111:11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1110 1111: mod mmxreg r/m</td>
</tr>
</tbody>
</table>

NOTES:
1. The pack instructions perform saturation from signed packed data of one type to signed or unsigned data of the next smaller type.
2. The format of the shift instructions has one additional format to support shifting by immediate shift-counts. The shift operations are not supported equally for all data types.

B.7 P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS

Table B-20 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

Table B-20. Formats and Encodings of XSAVE/XRSTOR/XGETBV/XSETBV Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGETBV – Get Value of Extended Control Register</td>
<td>0000 1111:0000 0001:1101 0000</td>
</tr>
<tr>
<td>XRSTOR – Restore Processor Extended States$^1$</td>
<td>0000 1111:1010 1110: mod$^A$ 101 r/m</td>
</tr>
<tr>
<td>XSAVE – Save Processor Extended States$^1$</td>
<td>0000 1111:1010 1110: mod$^A$ 100 r/m</td>
</tr>
<tr>
<td>XSETBV – Set Extended Control Register</td>
<td>0000 1111:0000 0001:1101 0001</td>
</tr>
</tbody>
</table>

NOTES:
1. For XSAVE and XRSTOR, “mod = 11” is reserved.
INSTRUCTION FORMATS AND ENCODINGS

B.8 SSE INSTRUCTION FORMATS AND ENCODINGS

The SSE instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables (Tables B-22, B-23, and B-24) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively. Some SSE instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. Mandatory prefixes are included in the tables.

Table B-22. Formats and Encodings of SSE Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS—Add Packed Single-Precision Floating-Point Values</td>
<td>0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>ADDSS—Add Scalar Single-Precision Floating-Point Values</td>
<td>0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values</td>
<td>0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values</td>
<td>0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>CMPPS—Compare Packed Single-Precision Floating-Point Values</td>
<td>0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>CMPSS—Compare Scalar Single-Precision Floating-Point Values</td>
<td>0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>
Table B-22. Formats and Encodings of SSE Floating-Point Instructions  (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>1111 0011:0000 1111:1100 0010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0010 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0010 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTP2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>0000 1111:0010 1010:11 xmmreg1 mmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0010 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0000 1111:0010 1101:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0010 1101: mod mmreg r/m</td>
</tr>
<tr>
<td>CVTTS2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>1111 0011:0000 1111:00101010:11 xmmreg1 r32</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:00101010: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>1111 0011:0000 1111:0010 1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>1111 0011:0000 1111:0010 1101: mod r32 r/m</td>
</tr>
<tr>
<td>CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0000 1111:0010 1100:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0010 1100: mod mmreg r/m</td>
</tr>
<tr>
<td>CVTTTP2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>1111 0011:0000 1111:0010 1100:11 r32 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>1111 0011:0000 1111:0010 1100: mod r32 r/m</td>
</tr>
<tr>
<td>DIVPS—Divide Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td>DIVSS—Divide Scalar Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td>LDMXCSR—Load MXCSR Register State</td>
<td></td>
</tr>
<tr>
<td>m32 to MXCSR</td>
<td>0000 1111:1010 1110:mod 010 mem</td>
</tr>
<tr>
<td>MAXPS—Return Maximum Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1111:11 xmmreg2 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MINPS—Return Minimum Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1101:11 xmmreg2 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MINSS—Return Minimum Scalar Double-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0010 1000:11 xmmreg2 xmmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0010 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0000 1111:0010 1001:11 xmmreg1 xmmreg2 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>0000 1111:0010 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0001 0010:11 xmmreg1 xmmreg2 xmmreg2</td>
</tr>
<tr>
<td><strong>MOVHPS—Move High Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0001 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:00010110:11 xmmreg1 xmmreg2 xmmreg2</td>
</tr>
<tr>
<td><strong>MOVLPS—Move Low Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0001 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0000 1111:0001 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0000 1111:0101 0000:11 r32 xmmreg</td>
</tr>
<tr>
<td><strong>MOVSS—Move Scalar Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0001 0000:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>1111 0011:0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>1111 0011:0000 1111:0001 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>1111 0011:0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0001 0000:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0000 1111:0001 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MULPS—Multiply Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MULSS—Multiply Scalar Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0101 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0101 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 0010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>1111 0011:0000 1111:0101 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0101 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 0010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>1111 0011:0000 1111:0101 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0000 1111:1100 0110: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Table B-22.  Formats and Encodings of SSE Floating-Point Instructions  (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0001: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value**

| xmmreg2 to xmmreg1     | 1111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 1111 0011:0000 1111:0101 1110: mod xmmreg r/m |

**STMXCSR—Store MXCSR Register State**

| MXCSR to mem           | 0000 1111:1010 1110:mod 111 mem |

**SUBPS—Subtract Packed Single-Precision Floating-Point Values**

| xmmreg2 to xmmreg1     | 0000 1111:0101 1100:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 0000 1111:0101 1100: mod xmmreg r/m |

**SUBSS—Subtract Scalar Single-Precision Floating-Point Values**

| xmmreg2 to xmmreg1     | 1111 0011:0000 1111:0101 1100:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 1111 0011:0000 1111:0101 1100: mod xmmreg r/m |

**UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS**

| xmmreg2 to xmmreg1     | 0000 1111:0010 1110:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 0000 1111:0010 1110: mod xmmreg r/m |

**UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values**

| xmmreg2 to xmmreg1     | 0000 1111:0001 0101:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 0000 1111:0001 0101: mod xmmreg r/m |

**UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values**

| xmmreg2 to xmmreg1     | 0000 1111:0001 0100:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 0000 1111:0001 0100: mod xmmreg r/m |

**XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values**

| xmmreg2 to xmmreg1     | 0000 1111:0101 0111:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 0000 1111:0101 0111: mod xmmreg r/m |
### Table B-23. Formats and Encodings of SSE Integer Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAVGB/PAVGw—Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 0000:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 0011:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>PEXTRw—Extract Word</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to reg32, imm8</td>
<td>0000 1111:1110 0101:11 r32 mmreg:imm8</td>
</tr>
<tr>
<td><strong>PINSRW—Insert Word</strong></td>
<td></td>
</tr>
<tr>
<td>reg32 to mmreg, imm8</td>
<td>0000 1111:1100 0100:11 mmreg r32:imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>0000 1111:1110 0100:11 mod mmreg r/m:imm8</td>
</tr>
<tr>
<td><strong>PMAXSw—Maximum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 1110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 1110:11 mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXUB—Maximum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 1010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 1010:11 mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMINSw—Minimum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 1110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 1110:11 mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMINUB—Minimum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 1010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 1010:11 mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMOVMSKB—Move Byte Mask To Integer</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to reg32</td>
<td>0000 1111:1110 0111:11 r32 mmreg</td>
</tr>
<tr>
<td><strong>PMULHUW—Multiply Packed Unsigned Integers and Store High Result</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 0100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 0100:11 mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PSADBW—Compute Sum of Absolute Differences</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1111 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1111 0110:11 mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PSHUFw—Shuffle Packed Words</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1, imm8</td>
<td>0000 1111:0111 0000:11 mmreg1 mmreg2:imm8</td>
</tr>
<tr>
<td>mem to mmreg, imm8</td>
<td>0000 1111:0111 0000:11 mod mmreg r/m:imm8</td>
</tr>
</tbody>
</table>
B.9 SSE2 INSTRUCTION FORMATS AND ENCODINGS

The SSE2 instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables show the formats and encodings for the SSE2 SIMD floating-point, SIMD integer, and cacheability instructions, respectively. Some SSE2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

B.9.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-25 shows the encoding of this gg field.

<table>
<thead>
<tr>
<th>gg</th>
<th>Granularity of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Packed Bytes</td>
</tr>
<tr>
<td>01</td>
<td>Packed Words</td>
</tr>
<tr>
<td>10</td>
<td>Packed Doublewords</td>
</tr>
<tr>
<td>11</td>
<td>Quadword</td>
</tr>
</tbody>
</table>

Table B-24. Format and Encoding of SSE Cacheability & Memory Ordering Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASKMOVQ—Store Selected Bytes of Quadword</td>
<td>0000 1111:1111 0111:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</td>
<td>0000 1111:0010 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVNTQ—Store Quadword Using Non-Temporal Hint</td>
<td>0000 1111:1110 0111: mod mmreg r/m</td>
</tr>
<tr>
<td>PREFETCHT0—Prefetch Temporal to All Cache Levels</td>
<td>0000 1111:0001 1000:modA 001 mem</td>
</tr>
<tr>
<td>PREFETCHT1—Prefetch Temporal to First Level Cache</td>
<td>0000 1111:0001 1000:modA 010 mem</td>
</tr>
<tr>
<td>PREFETCHT2—Prefetch Temporal to Second Level Cache</td>
<td>0000 1111:0001 1000:modA 011 mem</td>
</tr>
<tr>
<td>PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels</td>
<td>0000 1111:0001 1000:modA 000 mem</td>
</tr>
<tr>
<td>SFENCE—Store Fence</td>
<td>0000 1111:1010 1110:11 111 000</td>
</tr>
</tbody>
</table>
### Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD—Add Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>ADDSD—Add Scalar Double-Precision Floating-Point Values</td>
<td>1111 0010:0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>CMPFD—Compare Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0010 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>CMPFD—Compare Scalar Double-Precision Floating-Point Values</td>
<td>1111 0010:0000 1111:0010 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>1111 0010:0000 1111:0010 1010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>CMPSD—Compare Scalar Double-Precision Floating-Point Values</td>
<td>11110 010:0000 1111:0010 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>COMPS—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0010 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0010 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTIPD2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1001:11 xmmreg1 mmreg1</td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>0110 0110:0000 1111:0101 1001: mod mmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td>0110 0110:0000 1111:0010 1111:11 xmmreg1 mmreg1</td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0110 0110:0000 1111:0010 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td></td>
</tr>
<tr>
<td>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</td>
<td>1111 0010:0000 1111:0010 1010:11 xmmreg r32</td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>1111 0010:0000 1111:0010 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg to r32</td>
<td>1111 0010:0000 1111:0010 1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>1111 0010:0000 1111:0010 1101:mod r32 r/m</td>
</tr>
<tr>
<td><strong>CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0110 0110:0000 1111:0010 1100:11 mmreg xmmreg</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0110 0110:0000 1111:0010 1100: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTS2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>1111 0010:0000 1111:0010 1100:11 r32 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>1111 0010:0000 1111:0010 1100:mod r32 r/m</td>
</tr>
<tr>
<td><strong>CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:1110 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:1110 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1110 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1110 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>CVTDQ2PD</strong>—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</td>
<td>1111 0011:0000 1111:1110 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:1110 0110:11</td>
</tr>
<tr>
<td><strong>CVTPS2DQ</strong>—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td>0110 0110:0000 1111:0101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1011:11</td>
</tr>
<tr>
<td><strong>CVTTPS2DQ</strong>—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td>1111 0011:0000 1111:0101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1011:11</td>
</tr>
<tr>
<td><strong>CVTDQ2PS</strong>—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</td>
<td>0000 1111:0101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1011:11</td>
</tr>
<tr>
<td><strong>DIVPD</strong>—Divide Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1110:11</td>
</tr>
<tr>
<td><strong>DIVSD</strong>—Divide Scalar Double-Precision Floating-Point Values</td>
<td>1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1111:11</td>
</tr>
<tr>
<td><strong>MAXPD</strong>—Return Maximum Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1111:11</td>
</tr>
<tr>
<td><strong>MAXSD</strong>—Return Maximum Scalar Double-Precision Floating-Point Value</td>
<td>1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1111:11</td>
</tr>
<tr>
<td><strong>MINPD</strong>—Return Minimum Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1101:11</td>
</tr>
<tr>
<td><strong>MINSD</strong>—Return Minimum Scalar Double-Precision Floating-Point Value</td>
<td>1111 0010:0000 1111:0101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1101:11</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0010 1001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0110 0110:0000 1111:0010 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>0110 0110:0000 1111:0010 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0110 0110:0000 1111:0010 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVHPD—Move High Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0001 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVLPD—Move Low Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0110 0110:0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask</td>
<td>0110 0110:0000 1111:0101 0000:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0110 0110:0000 1111:0101 0000:11 r32 xmmreg</td>
</tr>
<tr>
<td>MOVSD—Move Scalar Double-Precision Floating-Point Values</td>
<td>1111 0010:0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>1111 0010:0000 1111:0001 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>1111 0010:0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0001 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>MULPD—Multiply Packed Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 1001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>MULSD—Multiply Scalar Double-Precision Floating-Point Values</td>
<td>1111 0010:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</td>
<td>0110 0110:0000 1111:0101 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 0110: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHUFPD—Shuffle Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:1100 0110: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td><strong>SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>SUBPD—Subtract Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1100: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>SUBSD—Subtract Scalar Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1100: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0010 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0001 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0001 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>MOVD—Move Doubleword</strong></td>
<td></td>
</tr>
<tr>
<td>reg to xmmreg</td>
<td>0110 0110:0000 1111:0110 1110: 11 xmmreg reg</td>
</tr>
<tr>
<td>reg from xmmreg</td>
<td>0110 0110:0000 1111:0111 1110: 11 xmmreg reg</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>0110 0110:0000 1111:0111 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVDQA—Move Aligned Double Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0110 0110:0000 1111:0111 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>0110 0110:0000 1111:0111 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVDQU—Move Unaligned Double Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0110 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>1111 0011:0000 1111:0111 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0110 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>1111 0011:0000 1111:0111 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVQ2DQ—Move Quadword from MMX to XMM Register</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>MOVQ2DQ—Move Quadword from XMM to MMX Register</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>MOVQ—Move Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0111 1110: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0110 0110:0000 1111:1101 0110: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0111 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>0110 0110:0000 1111:1101 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PACKSSDW1—Pack Dword To Word Data (signed with saturation)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 1011: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:0110 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PACKSSWB—Pack Word To Byte Data (signed with saturation)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 0011: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:0110 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PACKUSWB—Pack Word To Byte Data (unsigned with saturation)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 0111: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:0110 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PADDQ—Add Packed Quadword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1101 0100:11 mmreg1 mmreg2</td>
</tr>
</tbody>
</table>
### Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to mmreg</td>
<td>0000:1111:1101:0100: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1101:0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110:0110:0000:1111:1101:0100: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PADD—Add With Wrap-around</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1101:11 gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110:0110:0000:1111:1111:11 gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PADDS—Add Signed With Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1110:11 gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110:0110:0000:1111:1110:11 gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PADDUS—Add Unsigned With Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1101:11 gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110:0110:0000:1111:1101:11 gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PAND—Bitwise And</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1100:1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110:0110:0000:1111:1101:11 gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PANDN—Bitwise AndNot</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1101:11 gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110:0110:0000:1111:1110:11 gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PAVGB—Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1110:0011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110:0110:0000:1111:1110:0011:1100000 mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PAVGW—Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1110:0011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110:0110:0000:1111:1111:0110 mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPEQ—Packed Compare For Equality</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 with xmmreg2</td>
<td>0110:0110:0000:1111:0111:01 gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg with memory</td>
<td>0110:0110:0000:1111:0111:01 gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPGT—Packed Compare Greater (signed)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 with xmmreg2</td>
<td>0110:0110:0000:1111:0110:01 gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg with memory</td>
<td>0110:0110:0000:1111:0110:01 gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PEXTW—Extract Word</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PINSW—Insert Word</strong></td>
<td></td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0110:0110:0000:1111:1100:1000: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMADDW—Packed Multiply Add</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110:0110:0000:1111:1111:0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110:0110:0000:1111:1111:0101: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

#### PMAXSW—Maximum of Packed Signed Word Integers
- xmmreg2 to xmmreg1
- mem to xmmreg

#### PMAXUB—Maximum of Packed Unsigned Byte Integers
- xmmreg2 to xmmreg1
- mem to xmmreg

#### PMINSW—Minimum of Packed Signed Word Integers
- xmmreg2 to xmmreg1
- mem to xmmreg

#### PMINUB—Minimum of Packed Unsigned Byte Integers
- xmmreg2 to xmmreg1
- mem to xmmreg

#### PMOVMSKB—Move Byte Mask To Integer
- xmmreg to reg32

#### PMULHUW—Packed multiplication, store high word (unsigned)
- xmmreg2 to xmmreg1
- memory to xmmreg

#### PMULHW—Packed Multiplication, store high word
- xmmreg2 to xmmreg1
- memory to xmmreg

#### PMULLW—Packed Multiplication, store low word
- xmmreg2 to xmmreg1
- memory to xmmreg

#### PMULUDQ—Multiply Packed Unsigned Doubleword Integers
- mmreg2 to mmreg1
- mem to mmreg

#### POR—Bitwise Or
- xmmreg2 to xmmreg1
- mem to xmmreg

#### PSADBW—Compute Sum of Absolute Differences
- xmmreg2 to xmmreg1
- mem to xmmreg

#### PSHUFLW—Shuffle Packed Low Words

---

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMAXSW—Maximum of Packed Signed Word Integers</td>
<td>0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:1101110: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMAXUB—Maximum of Packed Unsigned Byte Integers</td>
<td>0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1101 1110:11 mod xmmreg r/m</td>
</tr>
<tr>
<td>PMINSW—Minimum of Packed Signed Word Integers</td>
<td>0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1101 1110:11 mod xmmreg r/m</td>
</tr>
<tr>
<td>PMINUB—Minimum of Packed Unsigned Byte Integers</td>
<td>0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1101 1110:11 mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVMSKB—Move Byte Mask To Integer</td>
<td>0110 0110:0000 1111:1101 0111:11 r32 xmmreg</td>
</tr>
<tr>
<td>PMULHUW—Packed multiplication, store high word (unsigned)</td>
<td>0110 0110:0000 1111:1110 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:1110 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMULHW—Packed Multiplication, store high word</td>
<td>0110 0110:0000 1111:1110 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:1110 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMULLW—Packed Multiplication, store low word</td>
<td>0110 0110:0000 1111:1101 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:1101 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMULUDQ—Multiply Packed Unsigned Doubleword Integers</td>
<td>0000 1111:1111 0100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1111 0100: mod mmreg r/m</td>
</tr>
<tr>
<td>POR—Bitwise Or</td>
<td>0110 0110:0000 1111:1110 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1110 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td>PSADBW—Compute Sum of Absolute Differences</td>
<td>0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1111 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td>PSHUFLW—Shuffle Packed Low Words</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>1111 0010:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>1111 0010:0000 1111:0111 0000:11 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PSHUFHW—Shuffle Packed High Words</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>1111 0011:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>1111 0011:0000 1111:0111 0000: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PSHUFD—Shuffle Packed Doublewords</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0111 0000: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PSLLDQ—Shift Double Quadword Left Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg, imm8</td>
<td>0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8</td>
</tr>
<tr>
<td><strong>PSLL—Packed Shift Left Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 by xmmreg2</td>
<td>0110 0110:0000 1111:1111 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg by memory</td>
<td>0110 0110:0000 1111:1111 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg by immediate</td>
<td>0110 0110:0000 1111:0111 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSRA—Packed Shift Right Arithmetic</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 by xmmreg2</td>
<td>0110 0110:0000 1111:1110 00gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg by memory</td>
<td>0110 0110:0000 1111:1110 00gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg by immediate</td>
<td>0110 0110:0000 1111:0111 00gg: 11 100 xmmreg: imm8</td>
</tr>
<tr>
<td><strong>PSRLDQ—Shift Double Quadword Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg, imm8</td>
<td>0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8</td>
</tr>
<tr>
<td><strong>PSRL—Packed Shift Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 by xmmreg2</td>
<td>0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg by memory</td>
<td>0110 0110:0000 1111:1101 00gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg by immediate</td>
<td>0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8</td>
</tr>
<tr>
<td><strong>PSUBQ—Subtract Packed Quadword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1111 011:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1111 1011: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1111 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1111 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSUB—Subtract With Wrap-around</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0110 0110:0000 1111:1111 10gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory from xmmreg</td>
<td>0110 0110:0000 1111:1111 10gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSUBS—Subtract Signed With Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0110 0110:0000 1111:1110 10gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory from xmmreg</td>
<td>0110 0110:0000 1111:1110 10gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSUBUS—Subtract Unsigned With Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0000 1111:1101 10gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory from xmmreg</td>
<td>0000 1111:1101 10gg: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUNPCKH—Unpack High Data To Next Larger Type</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0110 10gg:11 xmmreg1 xmmreg2 0110 0110:0000 1111:0110 10gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PUNPCKHQDQ—Unpack High Data</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0110 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td>PUNPCKL—Unpack Low Data To Next Larger Type</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0110 00gg:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 00gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PUNPCKLQDQ—Unpack Low Data</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0110 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 1100: mod xmmreg r/m</td>
</tr>
<tr>
<td>PXOR—Bitwise Xor</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1110 1111: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1110 1111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

Table B-28. Format and Encoding of SSE2 Cacheability Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASKMOVDQU—Store Selected Bytes of Double Quadword</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>CLFLUSH—Flush Cache Line</td>
<td>mem 0000 1111:1010 1110: 11 101 000 0000 1111:1010 1110: mod 111 r/m</td>
</tr>
<tr>
<td>MOVTTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</td>
<td>xmmreg to mem 0110 0110:0000 1111:1010 0011:11 0010 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVTTPDQ—Store Double Quadword Using Non-Temporal Hint</td>
<td>xmmreg to mem 0110 0110:0000 1111:1010 0011:11 0010 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVTNDI—Store Doubleword Using Non-Temporal Hint</td>
<td>reg to mem 0000 1111:1100 0011: mod reg r/m</td>
</tr>
<tr>
<td>PAUSE—Spin Loop Hint</td>
<td>1111 0011:1001 0000</td>
</tr>
<tr>
<td>LFENCE—Load Fence</td>
<td>0000 1111:1010 1010: 11 101 000 0000 1111:1010 1010: 11 110 000</td>
</tr>
<tr>
<td>MFENCE—Memory Fence</td>
<td>0000 1111:1010 1010: 11 110 000</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

B.10 SSE3 FORMATS AND ENCODINGS TABLE

The tables in this section provide SSE3 formats and encodings. Some SSE3 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

When in IA-32e mode, use of the REX.R prefix permits instructions that use general purpose and XMM registers to access additional registers. Some instructions require the REX.W prefix to promote the instruction to 64-bit operation. Instructions that require the REX.W prefix are listed (with their opcodes) in Section B.13.

Table B-29. Formats and Encodings of SSE3 Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1</td>
<td>01100110:00001111:11010000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>01100110:00001111:11010000: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:11010000: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

Table B-30. Formats and Encodings for SSE3 Event Management Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR—Set up a linear address range to be monitored by hardware</td>
<td>0000 1111 : 0000 0001:11 001 000</td>
</tr>
<tr>
<td>eax, ecx, edx</td>
<td></td>
</tr>
<tr>
<td>MWAIT—Wait until write-back store performed within the range specified by the instruction MONITOR</td>
<td>0000 1111 : 0000 0001:11 001 001</td>
</tr>
<tr>
<td>eax, ecx</td>
<td></td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

### Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>FISTTP—Store ST in int16 (chop) and pop</td>
<td>11011 111 : modA 001 r/m</td>
</tr>
<tr>
<td>m16int</td>
<td></td>
</tr>
<tr>
<td>FISTTP—Store ST in int32 (chop) and pop</td>
<td>11011 011 : modA 001 r/m</td>
</tr>
<tr>
<td>m32int</td>
<td></td>
</tr>
<tr>
<td>FISTTP—Store ST in int64 (chop) and pop</td>
<td>11011 101 : modA 001 r/m</td>
</tr>
<tr>
<td>m64int</td>
<td></td>
</tr>
<tr>
<td>LDDQU—Load unaligned integer 128-bit</td>
<td>11110010:00001111:11110000: modA xmmreg r/m</td>
</tr>
<tr>
<td>xmm, m128</td>
<td></td>
</tr>
<tr>
<td>MOVDDUP—Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate</td>
<td>11110010:00001111:00100110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>MOVSHDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high</td>
<td>11110011:00001111:00110110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>MOVSLDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low</td>
<td>11110011:00001111:00010010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
</tbody>
</table>

### B.11 SSSE3 FORMATS AND ENCODING TABLE

The tables in this section provide SSSE3 formats and encodings. Some SSSE3 instructions require a mandatory prefix (66H) as part of the three-byte opcode. These prefixes are included in the table below.

### Table B-32. Formats and Encodings for SSSE3 Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PABSB—Packed Absolute Value Bytes</td>
<td>0000 1111:0011 1000: 0001 1100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0001 1100: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110 0110:0000 1111:0011 1000: 0001 1100: mod xmmreg r/m</td>
</tr>
<tr>
<td>PABSD—Packed Absolute Value Double Words</td>
<td>0000 1111:0011 1000: 0001 1100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0001 1110: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td>PABSw—Packed Absolute Value Words</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0001 1101:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0001 1101: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0001 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0001 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PALIGNR—Packed Align Right</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1, imm8</td>
<td>0000 1111:0011 1010:0000 1111:11 mmreg1 mmreg2: imm8</td>
</tr>
<tr>
<td>mem to mmreg, imm8</td>
<td>0000 1111:0011 1010:0000 1111: mod mmreg r/m: imm8</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1111:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1111: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PHADDD—Packed Horizontal Add Double Words</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0010: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0000 0010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0000 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PHADDSW—Packed Horizontal Add and Saturate</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0011:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0011: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0000 0011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0000 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PHADDW—Packed Horizontal Add Words</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0001:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0001: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0000 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0000 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PHSUBD—Packed Horizontal Subtract Double Words</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0110: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0000 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0000 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PHSUBSW—Packed Horizontal Subtract and Saturate</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0111:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0111: mod mmreg r/m</td>
</tr>
</tbody>
</table>

Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)
### Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0000 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0000 0111:11 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PHSUBW—Packed Horizontal Subtract Words**

| mmreg2 to mmreg1      | 0000 1111:0011 1000:0000 0101:11 mmreg1 mmreg2 |
| mem to mmreg          | 0000 1111:0011 1000:0000 0101:11 mod mmreg r/m |
| xmmreg2 to xmmreg1   | 0110 0110:0000 1111:0011 1000:0000 0101:11 xmmreg1 xmmreg2 |
| mem to xmmreg        | 0110 0110:0000 1111:0011 1000:0000 0101:11 mod xmmreg r/m |

**PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes**

| mmreg2 to mmreg1      | 0000 1111:0011 1000:0000 0100:11 mmreg1 mmreg2 |
| mem to mmreg          | 0000 1111:0011 1000:0000 0100:11 mod mmreg r/m |
| xmmreg2 to xmmreg1   | 0110 0110:0000 1111:0011 1000:0000 0100:11 xmmreg1 xmmreg2 |
| mem to xmmreg        | 0110 0110:0000 1111:0011 1000:0000 0100:11 mod xmmreg r/m |

**PMULHRSW—Packed Multiply HIgn with Round and Scale**

| mmreg2 to mmreg1      | 0000 1111:0011 1000:0000 1011:11 mmreg1 mmreg2 |
| mem to mmreg          | 0000 1111:0011 1000:0000 1011:11 mod mmreg r/m |
| xmmreg2 to xmmreg1   | 0110 0110:0000 1111:0011 1000:0000 1011:11 xmmreg1 xmmreg2 |
| mem to xmmreg        | 0110 0110:0000 1111:0011 1000:0000 1011:11 mod xmmreg r/m |

**PSHUFB—Packed Shuffle Bytes**

| mmreg2 to mmreg1      | 0000 1111:0011 1000:0000 0000:11 mmreg1 mmreg2 |
| mem to mmreg          | 0000 1111:0011 1000:0000 0000:11 mod mmreg r/m |
| xmmreg2 to xmmreg1   | 0110 0110:0000 1111:0011 1000:0000 0000:11 xmmreg1 xmmreg2 |
| mem to xmmreg        | 0110 0110:0000 1111:0011 1000:0000 0000:11 mod xmmreg r/m |

**PSIGNB—Packed Sign Bytes**

| mmreg2 to mmreg1      | 0000 1111:0011 1000:0000 0000:11 mmreg1 mmreg2 |
| mem to mmreg          | 0000 1111:0011 1000:0000 0000:11 mod mmreg r/m |
| xmmreg2 to xmmreg1   | 0110 0110:0000 1111:0011 1000:0000 0000:11 xmmreg1 xmmreg2 |
| mem to xmmreg        | 0110 0110:0000 1111:0011 1000:0000 0000:11 mod xmmreg r/m |

**PSIGND—Packed Sign Double Words**

| mmreg2 to mmreg1      | 0000 1111:0011 1000:0000 1010:11 mmreg1 mmreg2 |
| mem to mmreg          | 0000 1111:0011 1000:0000 1010:11 mod mmreg r/m |
| xmmreg2 to xmmreg1   | 0110 0110:0000 1111:0011 1000:0000 1010:11 xmmreg1 xmmreg2 |
| mem to xmmreg        | 0110 0110:0000 1111:0011 1000:0000 1010:11 mod xmmreg r/m |

**PSIGNw—Packed Sign Words**
### Table B-33. Formats and Encodings of AESNI and PCLMULQDQ Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:1101 1111:11: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

### B.12 AESNI AND PCLMULQDQ INSTRUCTION FORMATS AND ENCODINGS

Table B-33 shows the formats and encodings for AESNI and PCLMULQDQ instructions.
### B.13 SPECIAL ENCODINGS FOR 64-BIT MODE

The following Pentium, P6, MMX, SSE, SSE2, SSE3 instructions are promoted to 64-bit operation in IA-32e mode by using REX.W. However, these entries are special cases that do not follow the general rules (specified in Section B.4).

Table B-34. Special Case Instructions Promoted Using REX.W

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOVcc—Conditional Move</td>
<td></td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B 0000 1111:0100 tttn : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister2 to qwordregister1</td>
<td>0100 1R0B 0000 1111:0100 tttn : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB 0000 1111 : 0100 tttn : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB 0000 1111 : 0100 tttn : mod qwordreg r/m</td>
</tr>
<tr>
<td>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 1111 0010:0000 1111:0010 1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0010:0000 1111:0010 1101:11 r64 xmmreg</td>
</tr>
<tr>
<td>mem64 to r32</td>
<td>0100 0RXB 1111 0010:0000 1111:0010 1101:11: mod r32 r/m</td>
</tr>
<tr>
<td>mem64 to r64</td>
<td>0100 1RXB 1111 0010:0000 1111:0010 1101:11: mod r64 r/m</td>
</tr>
<tr>
<td>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>0100 0R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r32</td>
</tr>
<tr>
<td>r64 to xmmreg1</td>
<td>0100 1R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r64</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0100 0RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem64 to xmmreg</td>
<td>0100 1RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>0100 0R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r32</td>
</tr>
<tr>
<td>r64 to xmmreg1</td>
<td>0100 1R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r64</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0100 0RXB 1111 0010:0000 1111:0010 010: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem64 to xmmreg</td>
<td>0100 1RXB 1111 0010:0000 1111:0010 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 1111 0011:0000 1111:0010 1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0010:0000 1111:0010 1100:11 r64 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>0100 0RXB 1111 0010:0000 1111:0010 1100:11: mod r32 r/m</td>
</tr>
<tr>
<td>mem32 to r64</td>
<td>0100 1RXB 1111 0010:0000 1111:0010 1101: mod r64 r/m</td>
</tr>
<tr>
<td>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 1111 0010:0000 1111:0010 1100:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0010:0000 1111:0010 1100:11 r64 xmmreg</td>
</tr>
<tr>
<td>mem64 to r32</td>
<td>0100 0RXB 1111 0010:0000 1111:0010 1100:11: mod r32 r/m</td>
</tr>
<tr>
<td>mem64 to r64</td>
<td>0100 1RXB 1111 0010:0000 1111:0010 1100: mod r64 r/m</td>
</tr>
<tr>
<td>CVTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 1111 0011:0000 1111:0010 1100:11 r32 xmmreg1</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0011:0000 1111:0010 1100:11 r64 xmmreg1</td>
</tr>
<tr>
<td>mem to r32</td>
<td>0100 0RXB 1111 0011:0000 1111:0010 1100:11: mod r32 r/m</td>
</tr>
<tr>
<td>mem32 to r64</td>
<td>0100 1RXB 1111 0011:0000 1111:0010 1100: mod r64 r/m</td>
</tr>
<tr>
<td>MOV/D/MOVQ—Move doubleword</td>
<td></td>
</tr>
<tr>
<td>reg to mmxreg</td>
<td>0100 0R0B 0000 1111:0110 1110:11 mmxreg reg</td>
</tr>
<tr>
<td>qwordreg to mmxreg</td>
<td>0100 1R0B 0000 1111:0110 1110:11 mmxreg qwordreg</td>
</tr>
<tr>
<td>reg from mmxreg</td>
<td>0100 0R0B 0000 1111:0111 1111:10 mmxreg reg</td>
</tr>
<tr>
<td>qwordreg from mmxreg</td>
<td>0100 1R0B 0000 1111:0111 1111:10 mmxreg qwordreg</td>
</tr>
<tr>
<td>mem to mmxreg</td>
<td>0100 0RXB 0000 1111:0110 1110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem64 to mmxreg</td>
<td>0100 1RXB 0000 1111:0110 1110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem from mmxreg</td>
<td>0100 0RXB 0000 1111:0111 1111: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem64 from mmxreg</td>
<td>0100 1RXB 0000 1111:0111 1110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg with memory</td>
<td>0100 0RXB 0000 1111:0110 101g: mod mmxreg r/m</td>
</tr>
<tr>
<td>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 0000 1111:0101 0000:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 00001111:01010000:11 r64 xmmreg</td>
</tr>
<tr>
<td>PEXT/RW—Extract Word</td>
<td></td>
</tr>
<tr>
<td>mmreg to reg32, imm8</td>
<td>0100 0R0B 0000 1111:1100 0101:11 r32 mmreg: imm8</td>
</tr>
</tbody>
</table>
### B.14 SSE4.1 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.1 formats and encodings. Some SSE4.1 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables.

In 64-bit mode, some instructions require REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg to reg32, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r32 xmmreg: imm8</td>
</tr>
<tr>
<td>xmmreg to reg64, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r64 xmmreg: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>reg32 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 xmmreg r32: imm8</td>
</tr>
<tr>
<td>reg32 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 xmmreg r32: imm8</td>
</tr>
<tr>
<td>xmmreg to reg32</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to reg64</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r64 xmmreg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg to reg32, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r32 xmmreg: imm8</td>
</tr>
<tr>
<td>xmmreg to reg64, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r64 xmmreg: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>xmmreg to reg32</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to reg64</td>
<td>0110 0110:0000 1111:1100 0111:1100:11 r64 xmmreg</td>
</tr>
</tbody>
</table>

### Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg to reg64, imm8</td>
<td>0100 1ROB 0000 1111:1100 0101:11 r64 xmmreg: imm8</td>
</tr>
<tr>
<td>xmmreg to reg32, imm8</td>
<td>0100 0ROB 0110 0110 0000 1111:1100 0101:11 r32 xmmreg: imm8</td>
</tr>
<tr>
<td>xmmreg to reg64, imm8</td>
<td>0100 1ROB 0110 0110 0000 1111:1100 0101:11 r64 xmmreg: imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>0100 0ROB 0000 1111:1100 0100:11 mmreg r32: imm8</td>
</tr>
<tr>
<td>reg64 to mmreg, imm8</td>
<td>0100 1ROB 0000 1111:1100 0100:11 mmreg r64: imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>0100 0ROB 0000 1111:1100 0100:11 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>0100 1RXB 0000 1111:11000100 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td>reg32 to xmmreg, imm8</td>
<td>0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r32: imm8</td>
</tr>
<tr>
<td>reg64 to xmmreg, imm8</td>
<td>0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r64: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0100 0RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0100 1RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>xmmreg to reg32</td>
<td>0110 0110:0000 1111:1101 0111:1111:1101:1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to reg64</td>
<td>0110 0110:0000 1111:1101 0111:1111:1101:1101:11 r64 xmmreg</td>
</tr>
</tbody>
</table>

### Instruction and Format | Encoding |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg to reg32</td>
<td>0110 0110:0000 1111:1101 0111:1111:1101:1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to reg64</td>
<td>0110 0110:0000 1111:1101 0111:1111:1101:1101:11 r64 xmmreg</td>
</tr>
</tbody>
</table>

### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLENDPD — Blend Packed Double-Precision Floats</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1010: 0000 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010: 0000 1100:11 mod xmmreg r/m</td>
</tr>
<tr>
<td>BLENDPS — Blend Packed Single-Precision Floats</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1010: 0000 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010: 0000 1100:11 mod xmmreg r/m</td>
</tr>
<tr>
<td>BLENDVPD — Variable Blend Packed Double-Precision Floats</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1 &lt;xmm0&gt;</td>
<td>0110 0110:0000 1111:0011 1000: 0001 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg &lt;xmm0&gt;</td>
<td>0110 0110:0000 1111:0011 1000: 0001 0101:11 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**BLENDVPS — Variable Blend Packed Single-Precision Floats**

| xmmreg2 to xmmreg1 <xmm0>     | 0110 0110:0000 1111:0011 1000: 0001 0100:11 xmmreg1 xmmreg2             |
| mem to xmmreg <xmm0>          | 0110 0110:0000 1111:0011 1000: 0001 0100:11 mod xmmreg r/m              |

**DPPD — Packed Double-Precision Dot Products**

| xmmreg2 to xmmreg1, imm8      | 0110 0110:0000 1111:0011 1010: 0100 0001:11 xmmreg1 xmmreg2: imm8        |
| mem to xmmreg, imm8           | 0110 0110:0000 1111:0011 1010: 0100 0000: mod xmmreg r/m: imm8           |

**DPPS — Packed Single-Precision Dot Products**

| xmmreg2 to xmmreg1, imm8      | 0110 0110:0000 1111:0011 1010: 0100 0000:11 xmmreg1 xmmreg2: imm8        |
| mem to xmmreg, imm8           | 0110 0110:0000 1111:0011 1010: 0100 0000: mod xmmreg r/m: imm8           |

**EXTRACTPS — Extract From Packed Single-Precision Floats**

| reg from xmmreg , imm8        | 0110 0110:0000 1111:0011 1010: 0001 0111:11 xmmreg reg: imm8             |
| mem from xmmreg , imm8        | 0110 0110:0000 1111:0011 1010: 0001 0111: mod xmmreg r/m: imm8            |

**INSERTPS — Insert Into Packed Single-Precision Floats**

| xmmreg2 to xmmreg1, imm8      | 0110 0110:0000 1111:0011 1010: 0010 0001:11 xmmreg1 xmmreg2: imm8        |
| mem to xmmreg, imm8           | 0110 0110:0000 1111:0011 1010: 0010 0001: mod xmmreg r/m: imm8            |

**MOVNTDQA — Load Double Quadword Non-temporal Aligned**

| m128 to xmmreg                | 0110 0110:0000 1111:0011 1000: 0010 1010:11 r/m xmmreg2                  |

**MPSADBW — Multiple Packed Sums of Absolute Difference**

| xmmreg2 to xmmreg1, imm8      | 0110 0110:0000 1111:0011 1010: 0100 0010:11 xmmreg1 xmmreg2: imm8        |
| mem to xmmreg, imm8           | 0110 0110:0000 1111:0011 1010: 0100 0000: mod xmmreg r/m: imm8            |

**PACKUSDW — Pack with Unsigned Saturation**

| xmmreg2 to xmmreg1            | 0110 0110:0000 1111:0011 1000: 0010 1011:11 xmmreg1 xmmreg2              |
| mem to xmmreg                 | 0110 0110:0000 1111:0011 1000: 0010 1011: mod xmmreg r/m                 |

**PBLENDVDB — Variable Blend Packed Bytes**

<p>| xmmreg2 to xmmreg1 &lt;xmm0&gt;     | 0110 0110:0000 1111:0011 1000: 0001 0000:11 xmmreg1 xmmreg2             |
| mem to xmmreg &lt;xmm0&gt;          | 0110 0110:0000 1111:0011 1000: 0001 0000: mod xmmreg r/m                 |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PBLENDW — Blend Packed Words</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 1110:11 xmmreg1</td>
</tr>
<tr>
<td></td>
<td>xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PCMPEQQ — Compare Packed Qword Data of Equal</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0010 1001:11 xmmreg1</td>
</tr>
<tr>
<td></td>
<td>xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0010 1001:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PEXTRB — Extract Byte</strong></td>
<td></td>
</tr>
<tr>
<td>reg from xmmreg , imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 0100:11 xmmreg reg: imm8</td>
</tr>
<tr>
<td>xmmreg to mem, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 0100:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PEXTRD — Extract DWord</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to mem, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 0110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PEXTRQ — Extract QWord</strong></td>
<td></td>
</tr>
<tr>
<td>r64 from xmmreg, imm8</td>
<td>0110 0110:REX.w:0000 1111:0011 1010:0001 0110:11 xmmreg reg: imm8</td>
</tr>
<tr>
<td>m64 from xmmreg, imm8</td>
<td>0110 0110:REX.w:0000 1111:0011 1010:0001 0110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PEXTRW — Extract Word</strong></td>
<td></td>
</tr>
<tr>
<td>mem from xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 0101:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PHMINPOSUW — Packed Horizontal Word Minimum</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0000 0001:11 xmmreg1</td>
</tr>
<tr>
<td></td>
<td>xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0000 1110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PINSRB — Extract Byte</strong></td>
<td></td>
</tr>
<tr>
<td>reg to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 0000:11 xmmreg reg: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 0000:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PINSRD — Extract Dword</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0010:0010: mod xmmreg r/m; imm8</td>
</tr>
<tr>
<td><strong>PINSRQ — Extract QWord</strong></td>
<td></td>
</tr>
<tr>
<td>r64 to xmmreg, imm8</td>
<td>0110 0110:REX.w:0000 1111:0011 1010:0010 0010:11 xmmreg reg: imm8</td>
</tr>
<tr>
<td>m64 to xmmreg, imm8</td>
<td>0110 0110:REX.w:0000 1111:0011 1010:0010 0010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMAXSB — Maximum of Packed Signed Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXSD — Maximum of Packed Signed Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXUD — Maximum of Packed Unsigned Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXUW — Maximum of Packed Unsigned Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMINSB — Minimum of Packed Signed Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMINSD — Minimum of Packed Signed Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMINUD — Minimum of Packed Unsigned Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMINUW — Minimum of Packed Unsigned Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMOVSXBD — Packed Move Sign Extend - Byte to Dword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0010 0010:11 xmmreg1 xmmreg2</td>
</tr>
</tbody>
</table>
### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0010 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVSBQB — Packed Move Sign Extend - Byte to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0010 0011:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0010 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVSBWB — Packed Move Sign Extend - Byte to Word</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0010 0000:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0010 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVSBWD — Packed Move Sign Extend - Word to Dword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0001:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVSBWQ — Packed Move Sign Extend - Word to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0010:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZBDQB — Packed Move Zero Extend - Byte to Dword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0001:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZBDQB — Packed Move Zero Extend - Byte to Word</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0010:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZBDQ — Packed Move Zero Extend - Dword to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0011:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0011: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0011 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMULDQ — Multiply Packed Signed Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0010 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0010 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMULLD — Multiply Packed Signed Dword Integers, Store low Result</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0001 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PTEST — Logical Compare</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0001 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>ROUNDPD — Round Packed Double-Precision Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1001:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1001: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>ROUNDP — Round Packed Single-Precision Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1000: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>ROUNDSD — Round Scalar Double-Precision Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1011:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1011: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>ROUNDS — Round Scalar Single-Precision Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1010: mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>

### B.15 SSE4.2 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.2 formats and encodings. Some SSE4.2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables. In 64-bit mode, some instructions requires REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CRC32 — Accumulate CRC32</strong></td>
<td></td>
</tr>
<tr>
<td>reg2 to reg1</td>
<td>1111 0010:0000 1111:0011 1000: 1111 000w :11 reg1 reg2</td>
</tr>
<tr>
<td>mem to reg</td>
<td>1111 0010:0000 1111:0011 1000: 1111 000w : mod reg r/m</td>
</tr>
<tr>
<td>bytereg2 to reg1</td>
<td>1111 0010:0100 WR0B:0000 1111:0011 1000: 1111 0000 :11 reg1 bytereg2</td>
</tr>
<tr>
<td>m8 to reg</td>
<td>1111 0010:0100 WR0B:0000 1111:0011 1000: 1111 0000 : mod reg r/m</td>
</tr>
<tr>
<td>qwreg2 to qwreg1</td>
<td>1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 :11 qwreg1 qwreg2</td>
</tr>
<tr>
<td>mem64 to qwreg</td>
<td>1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 : mod qwreg r/m</td>
</tr>
<tr>
<td><strong>PCMPESTRI— Packed Compare Explicit-Length Strings To Index</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0001:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPESTRM— Packed Compare Explicit-Length Strings To Mask</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPISTRI— Packed Compare Implicit-Length String To Index</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0011:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPISTRM— Packed Compare Implicit-Length Strings To Mask</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010: 0110 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPGTQ— Packed Compare Greater Than</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>POPCNT— Return Number of Bits Set to 1</strong></td>
<td></td>
</tr>
<tr>
<td>reg2 to reg1</td>
<td>1111 0011:0000 1111:1011 1000:11 reg1 reg2</td>
</tr>
<tr>
<td>mem to reg</td>
<td>1111 0011:0000 1111:1011 1000:mod reg1 r/m</td>
</tr>
<tr>
<td>qwreg2 to qwreg1</td>
<td>1111 0011:0100 1R0B:0000 1111:1011 1000:11 reg1 reg2</td>
</tr>
<tr>
<td>mem64 to qwreg</td>
<td>1111 0011:0100 1R0B:0000 1111:1011 1000:mod reg1 r/m</td>
</tr>
</tbody>
</table>
B.16 AVX FORMATS AND ENCODING TABLE

The tables in this section provide AVX formats and encodings. A mixed form of bit/hex/symbolic forms are used to express the various bytes:

The C4/C5 and opcode bytes are expressed in hex notation; the first and second payload byte of VEX, the modR/M byte is expressed in combination of bit/symbolic form. The first payload byte of C4 is expressed as combination of bits and hex form, with the hex value preceded by an underscore. The VEX bit field to encode upper register 8-15 uses 1’s complement form, each of those bit field is expressed as lower case notation rxb, instead of RXB.

The hybrid bit-nibble-byte form is depicted below:

Figure B-2. Hybrid Notation of VEX-Encoded Key Instruction Bytes

Table B-37. Encodings of AVX instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBLENDPD — Blend Packed Double-Precision Floats</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0D:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0D:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0D:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0D:mod ymmreg1 r/m: ymmreg4</td>
</tr>
<tr>
<td>VBLENDPS — Blend Packed Single-Precision Floats</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0C:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0C:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0C:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0C:mod ymmreg1 r/m: ymmreg4</td>
</tr>
<tr>
<td>VBLENDVPD — Variable Blend Packed Double-Precision Floats</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4B:11 xmmreg1 xmmreg3: xmmreg4</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4B:mod xmmreg1 r/m: xmmreg4</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4B:11 ymmreg1 ymmreg3: ymmreg4</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4B:mod ymmreg1 r/m: ymmreg4</td>
</tr>
<tr>
<td>VBLENDVPS — Variable Blend Packed Single-Precision Floats</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4A:11 xmmreg1 xmmreg3: xmmreg4</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4A:mod xmmreg1 r/m: xmmreg4</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4A:11 ymmreg1 ymmreg3: ymmreg4</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4A:mod ymmreg1 r/m: ymmreg4</td>
</tr>
</tbody>
</table>

**VDPPD — Packed Double-Precision Dot Products**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:41:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:41:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VDPPS — Packed Single-Precision Dot Products**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:40:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:40:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VEXTRACTPS — Extract From Packed Single-Precision Floats**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rxb0_3: w_F 001:17:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rxb0_3: w_F 001:17:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VINSERTPS — Insert Into Packed Single-Precision Floats**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>use imm to merge xmmreg3 with xmmreg2 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:21:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>use imm to merge mem with xmmreg2 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:21:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VMOVNTDQA — Load Double Quadword Non-temporal Aligned**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>m128 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:2A:11 xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VMPSADBW — Multiple Packed Sums of Absolute Difference**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg3 with xmmreg2 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:42:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>m128 with xmmreg2 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:42:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPACKUSDW — Pack with Unsigned Saturation**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg3 and xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:2B:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>m128 and xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:2B:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPBLENDVB — Variable Blend Packed Bytes**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: w xmmreg2 001:4C:11 xmmreg1 xmmreg3: xmmreg4</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: w xmmreg2 001:4C:mod xmmreg1 r/m: xmmreg4</td>
</tr>
</tbody>
</table>

**VPBLENDW — Blend Packed Words**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0E:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0E:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPCMPEQQ — Compare Packed Qword Data of Equal**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:29:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:29:mod xmmreg1 r/m:</td>
</tr>
</tbody>
</table>
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VPEXTRB — Extract Byte</strong></td>
<td></td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:14:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:14:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPEXTROD — Extract DWord</strong></td>
<td></td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:16:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:16:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPEXTRQ — Extract QWord</strong></td>
<td></td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 1_F 001:16:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 1_F 001:16:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPEXTRW — Extract Word</strong></td>
<td></td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:15:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:15:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPHMINPOSUW — Packed Horizontal Word Minimum</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: w_F 001:41:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: w_F 001:41:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPINSRB — Insert Byte</strong></td>
<td></td>
</tr>
<tr>
<td>reg with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0 xmmreg2 001:20:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0 xmmreg2 001:20:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPINSRD — Insert DWord</strong></td>
<td></td>
</tr>
<tr>
<td>reg with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0 xmmreg2 001:22:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0 xmmreg2 001:22:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPINSRQ — Insert QWord</strong></td>
<td></td>
</tr>
<tr>
<td>r64 with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 1 xmmreg2 001:22:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>m64 with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 1 xmmreg2 001:22:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPMAXSB — Maximum of Packed Signed Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXSD — Maximum of Packed Signed Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXUD — Maximum of Packed Unsigned Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3F:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXUW — Maximum of Packed Unsigned Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3E:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:3E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINSB — Minimum of Packed Signed Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:38:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:38:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td><strong>VPMINSD — Minimum of Packed Signed Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxbo_2: w xmmreg2 001:39:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxbo_2: w xmmreg2 001:39:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINUd — Minimum of Packed Unsigned Dword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxbo_2: w xmmreg2 001:3B:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxbo_2: w xmmreg2 001:3B:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINUw — Minimum of Packed Unsigned Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxbo_2: w xmmreg2 001:3A:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxbo_2: w xmmreg2 001:3A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBD — Packed Move Sign Extend - Byte to Dword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:21:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:21:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBQ — Packed Move Sign Extend - Byte to Qword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:22:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:22:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBw — Packed Move Sign Extend - Byte to Word</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:20:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:20:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXWD — Packed Move Sign Extend - Word to Dword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:23:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:23:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXWQ — Packed Move Sign Extend - Word to Qword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:24:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:24:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXDQ — Packed Move Sign Extend - Dword to Qword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:25:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:25:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBD — Packed Move Zero Extend - Byte to Dword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:31:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:31:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBQ — Packed Move Zero Extend - Byte to Qword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:32:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:32:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBw — Packed Move Zero Extend - Byte to Word</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:30:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxbo_2: w_F 001:30:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXWD — Packed Move Zero Extend - Word to Dword</strong></td>
<td></td>
</tr>
</tbody>
</table>

**INSTRUCTION FORMATS AND ENCODINGS**
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:33:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:33:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPMOVZXWQ — Packed Move Zero</td>
<td></td>
</tr>
<tr>
<td>Extend - Word to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:34:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:34:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPMOVZXDQ — Packed Move Zero</td>
<td></td>
</tr>
<tr>
<td>Extend - Dword to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:35:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:35:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPMULDOQ — Multiply Packed Signed Dword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:28:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPMULLDOQ — Multiply Packed Signed Dword Integers, Store low Result</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:40:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:40:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPTEST — Logical Compare</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:17:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>C4: rxb0_2: w_F 001:17:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_2: w_F 101:17:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg</td>
<td>C4: rxb0_2: w_F 101:17:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VROUNDPD — Round Packed Double-Precision Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>C4: rxb0_3: w_F 001:09:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1, imm8</td>
<td>C4: rxb0_3: w_F 001:09:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm8</td>
<td>C4: rxb0_3: w_F 101:09:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm8</td>
<td>C4: rxb0_3: w_F 101:09:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td>VROUNDPSS — Round Packed Single-Precision Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>C4: rxb0_3: w_F 001:08:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1, imm8</td>
<td>C4: rxb0_3: w_F 001:08:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm8</td>
<td>C4: rxb0_3: w_F 101:08:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm8</td>
<td>C4: rxb0_3: w_F 101:08:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td>VROUNDSD — Round Scalar Double-Precision Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 and xmmreg3 to xmmreg1, imm8</td>
<td>C4: rxb0_3: w xmmreg2 001:0B:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 and mem to xmmreg1, imm8</td>
<td>C4: rxb0_3: w xmmreg2 001:0B:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>VROUNDS — Round Scalar Single-Precision Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 and xmmreg3 to xmmreg1, imm8</td>
<td>C4: rxb0_3: w xmmreg2 001:0A:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 and mem to xmmreg1, imm8</td>
<td>C4: rxb0_3: w xmmreg2 001:0A:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**INSTRUCTION FORMATS AND ENCODINGS**
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPCMPESTRI — Packed Compare Explicit Length Strings, Return Index</td>
<td><code>xmmreg2 with xmmreg1, imm8 C4: rxb0_3: w_F 001:61:11 xmmreg1 xmmreg2: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>mem with xmmreg1, imm8 C4: rxb0_3: w_F 001:61:mod xmmreg1 r/m: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>VPCMPGTM — Packed Compare Explicit Length Strings, Return Mask</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with xmmreg1, imm8 C4: rxb0_3: w_F 001:60:11 xmmreg1 xmmreg2: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>mem with xmmreg1, imm8 C4: rxb0_3: w_F 001:60:mod xmmreg1 r/m: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>VPCMPGTQ — Compare Packed Data for Greater Than</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with xmmreg3 to xmmreg1  C4: rxb0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with mem to xmmreg1  C4: rxb0_2: w xmmreg2 001:28:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td></td>
<td><code>VPCMPESTRI — Packed Compare Implicit Length Strings, Return Index</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with xmmreg1, imm8 C4: rxb0_3: w_F 001:63:11 xmmreg1 xmmreg2: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>mem with xmmreg1, imm8 C4: rxb0_3: w_F 001:63:mod xmmreg1 r/m: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>VPCMPISTRM — Packed Compare Implicit Length Strings, Return Mask</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with xmmreg1, imm8 C4: rxb0_3: w_F 001:62:11 xmmreg1 xmmreg2: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>mem with xmmreg, imm8 C4: rxb0_3: w_F 001:62:mod xmmreg1 r/m: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>VAESENC — Perform One Round of an AES Encryption Flow</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with xmmreg3 to xmmreg1  C4: rxb0_2: w xmmreg2 001:DC:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with mem to xmmreg1  C4: rxb0_2: w xmmreg2 001:DC:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td></td>
<td><code>VAESENCLAST — Perform Last Round of an AES Encryption Flow</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with xmmreg3 to xmmreg1  C4: rxb0_2: w xmmreg2 001:DD:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with mem to xmmreg1  C4: rxb0_2: w xmmreg2 001:DD:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td></td>
<td><code>VAESENC — Perform One Round of an AES Encryption Flow</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 to xmmreg1</code></td>
</tr>
<tr>
<td></td>
<td><code>mem to xmmreg1 C4: rxb0_2: w_F 001:DB:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td></td>
<td><code>VAESENCLAST — Perform Last Round of an AES Encryption Flow</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with xmmreg3 to xmmreg1  C4: rxb0_2: w xmmreg2 001:DF:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 with mem to xmmreg1  C4: rxb0_2: w xmmreg2 001:DF:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td></td>
<td><code>VAESIMC — Perform the AES InvMixColumn Transformation</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 to xmmreg1</code></td>
</tr>
<tr>
<td></td>
<td><code>mem to xmmreg1 C4: rxb0_2: w_F 001:DB:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td></td>
<td><code>VAESKEYGENASSIST — AES Round Key Generation Assist</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 to xmmreg1, imm8 C4: rxb0_3: w_F 001:DF:11 xmmreg1 xmmreg2: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>mem to xmmreg, imm8 C4: rxb0_3: w_F 001:DF:mod xmmreg1 r/m: imm</code></td>
</tr>
<tr>
<td></td>
<td><code>VPABSB — Packed Absolute Value</code></td>
</tr>
<tr>
<td></td>
<td><code>xmmreg2 to xmmreg1</code></td>
</tr>
<tr>
<td></td>
<td><code>C4: rxb0_2: w_F 001:1C:11 xmmreg1 xmmreg2</code></td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w_F 001:1Cmod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPABSD — Packed Absolute Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: <code>rxb0_2: w_F 001:1E:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w_F 001:1E:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPABSW — Packed Absolute Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: <code>rxb0_2: w_F 001:1D:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w_F 001:1D:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPALIGNR — Packed Align Right</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1, imm8</td>
<td>C4: <code>rxb0_3: w xmmreg2 001:DD:11 xmmreg1 xmmreg3:imm</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1, imm8</td>
<td>C4: <code>rxb0_3: w xmmreg2 001:DD:mod xmmreg1 r/m:imm</code></td>
</tr>
<tr>
<td><strong>VPHADDD — Packed Horizontal Add</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:02:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:02:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPHADDW — Packed Horizontal Add</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:01:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:01:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPHADDSW — Packed Horizontal Add and Saturate</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:03:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:03:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPHSUBD — Packed Horizontal Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:06:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:06:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPHSUBW — Packed Horizontal Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:05:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:05:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPHSUBSW — Packed Horizontal Subtract and Saturate</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:07:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:07:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:04:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:04:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPMULHRSW — Packed Multiply High with Round and Scale</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:08:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:08:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPSHUFB — Packed Shuffle Bytes</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:00:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:00:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPSIGNB — Packed SIGN</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: <code>rxb0_2: w xmmreg2 001:08:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_2: w xmmreg2 001:08:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPSIGND — Packed SIGN</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_2: w xmmreg2 001:0A:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_2: w xmmreg2 001:0A:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VPSIGNW — Packed SIGN</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_2: w xmmreg2 001:09:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_2: w xmmreg2 001:09:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VADDSUBPD — Packed Double-FP Add/Subtract</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:D0:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:D0:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 001:D0:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 001:D0:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VADDSUBPS — Packed Single-FP Add/Subtract</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 011:D0:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 011:D0:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 011:D0:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 011:D0:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VHADDPD — Packed Double-FP Horizontal Add</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:7C:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:7C:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 001:7C:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 001:7C:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VHADDPS — Packed Single-FP Horizontal Add</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 011:7C:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 011:7C:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 011:7C:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r_xxmmreglo2 011:7C:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7C:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7C:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 111:7C:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 111:7C:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VHSUBPD — Packed Double-FP Horizontal Subtract

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:7D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:7D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:7D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:7D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:7D:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:7D:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:7D:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:7D:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VHSUBPS — Packed Single-FP Horizontal Subtract

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:7D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:7D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:7D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:7D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7D:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7D:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 111:7D:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 111:7D:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VLDDQU — Load Unaligned Integer 128 Bits

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:F0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 011:F0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:F0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 111:F0:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VMOVDDUP — Move One Double-FP and Duplicate

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:12:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 011:12:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 011:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:12:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:12:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 111:12:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 111:12:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VMOVHLPS — Move Packed Single-Precision Floating-Point Values High to Low

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 and xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:12:11 xmmreg1 xmmreg3</td>
</tr>
</tbody>
</table>
### Instruction and Format Encoding

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 and xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:12:11 xmmreg1 xmmreglo3</td>
</tr>
</tbody>
</table>

#### VMOVSHDUP — Move Packed Single-FP High and Duplicate

<table>
<thead>
<tr>
<th>xmmreg2 to xmmreg1</th>
<th>C4: rxb0_1: w_F 010:16:11 xmmreg1 xmmreg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:16:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 to xmmreg1</td>
<td>C5: r_F 010:16:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:16:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:16:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:16:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 to ymmreg1</td>
<td>C5: r_F 110:16:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:16:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VMOVSLDUP — Move Packed Single-FP Low and Duplicate

<table>
<thead>
<tr>
<th>xmmreg2 to xmmreg1</th>
<th>C4: rxb0_1: w_F 010:12:11 xmmreg1 xmmreg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 to xmmreg1</td>
<td>C5: r_F 010:12:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:12:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:12:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 to ymmreg1</td>
<td>C5: r_F 110:12:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:12:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VADDPD — Add Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 001:58:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:58:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:58:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:58:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:58:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:58:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VADDSD — Add Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 011:58:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:58:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:58:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VANDPD — Bitwise Logical AND of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 001:54:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:54:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:54:11 xmmreg1 xmmreglo3</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r_xmmreglo2 001:54:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreg2 with ymmreg3 to ymmreg1</code></td>
<td>C4: <code>rb0_1: w ymmreg2 101:54:11 ymmreg1 ymmreg3</code></td>
</tr>
<tr>
<td><code>ymmreg2 with mem to ymmreg1</code></td>
<td>C4: <code>rb0_1: w ymmreg2 101:54:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreglo2 with ymmreglo3 to ymmreg1</code></td>
<td>C5: <code>r_ymmreglo2 101:54:11 ymmreg1 ymmreglo3</code></td>
</tr>
<tr>
<td><code>ymmreglo2 with mem to ymmreg1</code></td>
<td>C5: <code>r_ymmreglo2 101:54:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VANDNPD — Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 001:55:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 001:55:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td>C5: <code>r_xmmreglo2 001:55:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r_xmmreglo2 001:55:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VCMPPD — Compare Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 011:C2:11 xmmreg1 xmmreg3: imm</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 011:C2:mod xmmreg1 r/m: imm</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td>C5: <code>r_xmmreglo2 011:C2:11 xmmreg1 xmmreglo3: imm</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r_xmmreglo2 011:C2:mod xmmreg1 r/m: imm</code></td>
</tr>
<tr>
<td><strong>VCMPSD — Compare Scalar Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 001:2F:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 001:2F:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo to xmmreg1</code></td>
<td>C5: <code>r_F 001:2F:11 xmmreg1 xmmreglo</code></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C5: <code>r_F 001:2F:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VCOMISD — Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</strong></td>
<td></td>
</tr>
<tr>
<td><code>VCVTDQ2PD— Convert Packed Dword Integers to Packed Double-Precision FP Values</code></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 010:E6:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 010:E6:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 010:E6:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:E6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:E6:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:E6:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 110:E6:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:E6:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VCVTDQ2PS— Convert Packed Dword Integers to Packed Single-Precision FP Values

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:5B:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:5B:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:5B:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:5B:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:5B:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:5B:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 100:5B:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:5B:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VCVTPD2DQ— Convert Packed Double-Precision FP Values to Packed Dword Integers

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:E6:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:E6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 011:E6:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 011:E6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:E6:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:E6:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 111:E6:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 111:E6:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VCVTPD2PS— Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:5A:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:5A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:5A:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:5A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:5A:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:5A:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 101:5A:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:5A:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VCVTPS2DQ— Convert Packed Single-Precision FP Values to Packed Dword Integers

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:5B:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:5B:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>
## INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:5B:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:5B:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:5B:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:5B:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 101:5B:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:5B:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VCVTPS2PD— Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:5A:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:5A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:5A:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:5A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:5A:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:5A:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 100:5A:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:5A:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VCVTS2SI— Convert Scalar Double-Precision FP Value to Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to reg32</td>
<td>C4: rxb0_1: 0_F 011:2D:11 reg xmmreg1</td>
</tr>
<tr>
<td>mem to reg32</td>
<td>C4: rxb0_1: 0_F 011:2D:mod reg r/m</td>
</tr>
<tr>
<td>xmmreglo to reg32</td>
<td>C5: r_F 011:2D:11 reg xmmreglo</td>
</tr>
<tr>
<td>mem to reg32</td>
<td>C5: r_F 011:2D:mod reg r/m</td>
</tr>
<tr>
<td>ymmreg1 to reg64</td>
<td>C4: rxb0_1: 1_F 111:2D:11 reg ymmreg1</td>
</tr>
<tr>
<td>mem to reg64</td>
<td>C4: rxb0_1: 1_F 111:2D:mod reg r/m</td>
</tr>
<tr>
<td><strong>VCVTS2SS — Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3</td>
<td>C4: rxb0_1: w xmmreg2 011:5A:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3</td>
<td>C5: r_xmmreglo2 011:5A:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VCVTSS2SD— Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3</td>
<td>C4: rxb0_1: w xmmreg2 010:5A:11 xmmreg1 xmmreg3</td>
</tr>
</tbody>
</table>
### Instruction and Format Encoding

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5A:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5A:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VCVTTPD2DQ— Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers**

| xmmreg2 to xmmreg1 | C4: rxb0_1: w_F 001:E6:11 xmmreg1 xmmreg2 |
| mem to xmmreg1 | C4: rxb0_1: w_F 001:E6:mod xmmreg1 r/m |
| xmmreglo to xmmreg1 | C5: r_F 001:E6:11 xmmreg1 xmmreglo |
| mem to xmmreg1 | C5: r_F 001:E6:mod xmmreg1 r/m |
| ymmreg2 to ymmreg1 | C4: rxb0_1: w_F 101:E6:11 ymmreg1 ymmreg2 |
| mem to ymmreg1 | C4: rxb0_1: w_F 101:E6:mod ymmreg1 r/m |
| ymmreglo to ymmreg1 | C5: r_F 101:E6:11 ymmreg1 ymmreglo |
| mem to ymmreg1 | C5: r_F 101:E6:mod ymmreg1 r/m |

**VCVTTPS2DQ— Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers**

| xmmreg2 to xmmreg1 | C4: rxb0_1: w_F 010:5B:11 xmmreg1 xmmreg2 |
| mem to xmmreg1 | C4: rxb0_1: w_F 010:5B:mod xmmreg1 r/m |
| xmmreglo to xmmreg1 | C5: r_F 010:5B:11 xmmreg1 xmmreglo |
| mem to xmmreg1 | C5: r_F 010:5B:mod xmmreg1 r/m |
| ymmreg2 to ymmreg1 | C4: rxb0_1: w_F 110:5B:11 ymmreg1 ymmreg2 |
| mem to ymmreg1 | C4: rxb0_1: w_F 110:5B:mod ymmreg1 r/m |
| ymmreglo to ymmreg1 | C5: r_F 110:5B:11 ymmreg1 ymmreglo |
| mem to ymmreg1 | C5: r_F 110:5B:mod ymmreg1 r/m |

**VCVTTSD2SI— Convert with Truncation Scalar Double-Precision FP Value to Signed Integer**

| xmmreg1 to reg32 | C4: rxb0_1: 0_F 011:2C:11 reg xmmreg1 |
| mem to reg32 | C4: rxb0_1: 0_F 011:2C:mod reg r/m |
| xmmreglo to reg32 | C5: r_F 011:2C:11 reg xmmreglo |
| mem to reg32 | C5: r_F 011:2C:mod reg r/m |
| xmmreg1 to reg64 | C4: rxb0_1: 1_F 011:2C:11 reg xmmreg1 |
| mem to reg64 | C4: rxb0_1: 1_F 011:2C:mod reg r/m |

**VDIVPD — Divide Packed Double-Precision Floating-Point Values**

<p>| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:5E:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:5E:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:5E:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:5E:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 101:5E:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 101:5E:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r_ymmreglo2 101:5E:11 ymmreg1 ymmreglo3 |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:5E:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VDIVSD — Divide Scalar Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5E:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5E:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VMASKMOVQDU — Store Selected Bytes of Double Quadword</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem; xmmreg2 as mask</td>
<td>C4: rxb0_1: w_F 001:F7:11 r/m xmmreg1: xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem; xmmreg2 as mask</td>
<td>C5: r_F 001:F7:11 r/m xmmreg1: xmmreg2</td>
</tr>
<tr>
<td>VMAXPD — Return Maximum Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5F:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5F:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:5F:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:5F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:5F:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:5F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VMAXSD — Return Maximum Scalar Double-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VMINPD — Return Minimum Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VMINSD — Return Minimum Scalar Double-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5D:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VMOVAPD — Move Aligned Packed Double-Precision Floating-Point Values**

| xmmreg2 to xmmreg1 | C4: rxb0_1: w_F 001:28:11 xmmreg1 xmmreg2 |
| mem to xmmreg1 | C4: rxb0_1: w_F 001:28:mod xmmreg1 r/m |
| xmmreglo to xmmreg1 | C5: r_F 001:28:11 xmmreg1 xmmreglo |
| mem to xmmreg1 | C5: r_F 001:28:mod xmmreg1 r/m |
| xmmreg1 to xmmreg2 | C4: rxb0_1: w_F 001:29:11 xmmreg2 xmmreg1 |
| xmmreg1 to mem | C4: rxb0_1: w_F 001:29:mod r/m xmmreg1 |
| xmmreg1 to xmmreglo | C5: r_F 001:29:11 xmmreglo xmmreg1 |
| xmmreg1 to mem | C5: r_F 001:29:mod r/m xmmreg1 |
| ymmreg2 to ymmreg1 | C4: rxb0_1: w_F 101:28:11 ymmreg1 ymmreg2 |
| mem to ymmreg1 | C4: rxb0_1: w_F 101:28:mod ymmreg1 r/m |
| ymmreglo to ymmreg1 | C5: r_F 101:28:11 ymmreg1 ymmreglo |
| mem to ymmreg1 | C5: r_F 101:28:mod ymmreg1 r/m |
| ymmreg1 to ymmreg2 | C4: rxb0_1: w_F 101:29:11 ymmreg2 ymmreg1 |
| ymmreg1 to mem | C4: rxb0_1: w_F 101:29:mod r/m ymmreg1 |
| ymmreg1 to xmmreglo | C5: r_F 101:29:11 xmmreglo ymmreg1 |
| ymmreg1 to mem | C5: r_F 101:29:mod r/m ymmreg1 |

**VMOVD — Move Doubleword**

| reg32 to xmmreg1 | C4: rxb0_1: 0_F 001:6E:11 xmmreg1 reg32 |
| mem32 to xmmreg1 | C4: rxb0_1: 0_F 001:6E:mod xmmreg1 r/m |
| reg32 to xmmreg1 | C5: r_F 001:6E:11 xmmreg1 reg32 |
| mem32 to xmmreg1 | C5: r_F 001:6E:mod xmmreg1 r/m |
| xmmreg1 to reg32 | C4: rxb0_1: 0_F 001:7E:11 reg32 xmmreg1 |
| xmmreg1 to mem32 | C4: rxb0_1: 0_F 001:7E:mod mem32 xmmreg1 |
| xmmreglo to reg32 | C5: r_F 001:7E:11 reg32 xmmreglo |
| xmmreglo to mem32 | C5: r_F 001:7E:mod mem32 xmmreglo |

**VMOVQ — Move Quadword**

| reg64 to xmmreg1 | C4: rxb0_1: 1_F 001:6E:11 xmmreg1 reg64 |
| mem64 to xmmreg1 | C4: rxb0_1: 1_F 001:6E:mod xmmreg1 r/m |
| xmmreg1 to reg64 | C4: rxb0_1: 1_F 001:7E:11 reg64 xmmreg1 |
| xmmreg1 to mem64 | C4: rxb0_1: 1_F 001:7E:mod r/m xmmreg1 |

**VMOVDQA — Move Aligned Double Quadword**

| xmmreg2 to xmmreg1 | C4: rxb0_1: w_F 001:6F:11 xmmreg1 xmmreg2 |
| mem to xmmreg1 | C4: rxb0_1: w_F 001:6F:mod xmmreg1 r/m |
| xmmreglo to xmmreg1 | C5: r_F 001:6F:11 xmmreg1 xmmreglo |
| mem to xmmreg1 | C5: r_F 001:6F:mod xmmreg1 r/m |
## INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 001:7F:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 001:7F:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 001:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:6F:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:6F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 101:6F:mod ymmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:6F:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 101:7F:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 101:7F:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 101:7F:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 101:7F:mod r/m ymmreg1</td>
</tr>
</tbody>
</table>

### VMOVDQU — Move Unaligned Double Quadword

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:6F:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:6F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 010:6F:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:6F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 010:7F:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 010:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 010:7F:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:6F:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:6F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 110:6F:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:6F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 110:7F:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 110:7F:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 110:7F:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 110:7F:mod r/m ymmreg1</td>
</tr>
</tbody>
</table>

### VMOVHPD — Move High Packed Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg1 and mem to xmmreg2</td>
<td>C4: rxb0_1: w xmmreg1 001:16:11 xmmreg2 r/m</td>
</tr>
<tr>
<td>xmmreg1 and mem to xmmreglo2</td>
<td>C5: r_xmmreg1 001:16:11 xmmreglo2 r/m</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:17:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 001:17:mod r/m xmmreglo</td>
</tr>
</tbody>
</table>

### VMOVLPD — Move Low Packed Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg1 and mem to xmmreg2</td>
<td>C4: rxb0_1: w xmmreg1 001:12:11 xmmreg2 r/m</td>
</tr>
<tr>
<td>xmmreg1 and mem to xmmreglo2</td>
<td>C5: r_xmmreg1 001:12:11 xmmreglo2 r/m</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:13:mod r/m xmmreg1</td>
</tr>
</tbody>
</table>
### Instruction and Format Encoding

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 001:13:mod r/m xmmreglo</td>
</tr>
<tr>
<td><strong>VMOVMSKPD — Extract Packed Double-Precision Floating-Point Sign Mask</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to reg</td>
<td>C4: rxb0_1: w_F 001:50:11 reg xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to reg</td>
<td>C5: r_F 001:50:11 reg xmmreglo</td>
</tr>
<tr>
<td>ymmreg2 to reg</td>
<td>C4: rxb0_1: w_F 101:50:11 reg ymmreg1</td>
</tr>
<tr>
<td>ymmreglo to reg</td>
<td>C5: r_F 101:50:11 reg ymmreglo</td>
</tr>
<tr>
<td><strong>VMOVNTDQ — Store Double Quadword Using Non-Temporal Hint</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:E7:11 r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 001:E7:11 r/m xmmreglo</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 101:E7:11 r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreglo to mem</td>
<td>C5: r_F 101:E7:11 r/m ymmreglo</td>
</tr>
<tr>
<td><strong>VMOVNTPD — Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:2B:11 r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 001:2B:11 r/m xmmreglo</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 101:2B:11r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreglo to mem</td>
<td>C5: r_F 101:2B:11r/m ymmreglo</td>
</tr>
<tr>
<td><strong>VMOVSD — Move Scalar Double-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:10:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:10:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 011:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:11:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 011:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:11:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 011:11:mod r/m xmmreglo</td>
</tr>
<tr>
<td><strong>VMOVUPD — Move Unaligned Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:10:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:10:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:10:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg1</td>
<td>C5: r_F 101:10:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 001:11:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:11:mod r/m xmmreg1</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 001:11:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 001:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 101:11:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 101:11:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to xmmreglo</td>
<td>C5: r_F 101:11:11 xmmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 101:11:mod r/m ymmreg1</td>
</tr>
</tbody>
</table>

**VMULPD — Multiply Packed Double-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:59:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 001:59:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:59:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 001:59:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 101:59:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1    | C4: rxb0_1: w ymmreg2 101:59:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r ymmreglo2 101:59:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r ymmreglo2 101:59:mod ymmreg1 r/m |

**VMULSD — Multiply Scalar Double-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 011:59:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 011:59:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 011:59:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 011:59:mod xmmreg1 r/m |

**VORPD — Bitwise Logical OR of Double-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:56:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 001:56:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:56:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 001:56:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 101:56:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1    | C4: rxb0_1: w ymmreg2 101:56:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r ymmreglo2 101:56:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r ymmreglo2 101:56:mod ymmreg1 r/m |

**VPACKSSWB— Pack with Signed Saturation**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:63:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 001:63:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:63:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 001:63:mod xmmreg1 r/m |

**VPACKSSDW— Pack with Signed Saturation**

<p>| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:6B:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 001:6B:mod xmmreg1 r/m |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:6B:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:6B:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPACKUSWB— Pack with Unsigned Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:67:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:67:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:67:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:67:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDB — Add Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FC:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:FC:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:FC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDW — Add Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FD:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FD:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:FD:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:FD:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDD — Add Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FE:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:FE:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:FE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDQ — Add Packed Quadword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D4:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D4:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDSB — Add Packed Signed Integers with Signed Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EC:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:EC:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:EC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDSW — Add Packed Signed Integers with Signed Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:ED:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:ED:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:ED:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:ED:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>
### Instruction and Format Encoding

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VPADDUSB — Add Packed Unsigned Integers with Unsigned Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DC:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DC:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDUSW — Add Packed Unsigned Integers with Unsigned Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DD:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DD:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DD:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DD:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPAND — Logical AND</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DB:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DB:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPANDN — Logical AND NOT</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DF:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DF:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DF:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DF:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPAVGB — Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E0:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E0:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPAVGW — Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E3:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E3:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E3:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E3:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPCMPEQB — Compare Packed Data for Equal</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:74:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:74:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:74:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:74:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPCMPEQW — Compare Packed Data for Equal</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:75:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:75:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:75:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:75:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPCMPEQD — Compare Packed Data for Equal</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:76:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:76:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:76:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:76:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPCMPGTB — Compare Packed Signed Integers for Greater Than</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:64:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:64:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:64:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:64:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPCMPGTW — Compare Packed Signed Integers for Greater Than</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:65:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:65:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:65:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:65:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPMADDWD — Multiply and Add Packed Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F5:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F5:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPMAXSW — Maximum of Packed Signed Word Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EE:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:EE:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:EE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXUB — Maximum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DE:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DE:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINSW — Minimum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EA:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:EA:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:EA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINUB — Minimum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DA:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DA:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVMSKB — Move Byte Mask</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C4: rxb0_1: w_F 001:D7:11 reg xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C5: r_F 001:D7:11 reg xmmreg1</td>
</tr>
<tr>
<td><strong>VPMULHUW — Multiply Packed Unsigned Integers and Store High Result</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E4:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E4:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMULHW — Multiply Packed Signed Integers and Store High Result</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E5:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E5:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMULLW — Multiply Packed Signed Integers and Store Low Result</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D5:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D5:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>VPMULUDQ — Multiply Packed Unsigned Doubleword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F4:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F4:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPOR — Bitwise Logical OR</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EB:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EB:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSADBW — Compute Sum of Absolute Differences</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F6:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F6:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSHUFDF — Shuffle Packed Doublewords</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 001:70:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 001:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm</td>
<td>C5: r_F 001:70:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C5: r_F 001:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPSHUFHF — Shuffle Packed High Words</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 010:70:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 010:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm</td>
<td>C5: r_F 010:70:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C5: r_F 010:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPSHUFIL — Shuffle Packed Low Words</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 011:70:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 011:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm</td>
<td>C5: r_F 011:70:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C5: r_F 011:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPSLUDQ — Shift Double Quadword Left Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm</td>
<td>C5: r_F 001:73:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td><strong>VPSLLL — Shift Packed Data Left Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F1:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F1:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F1:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F1:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm8</td>
<td>C5: r_F 001:71:11 xmmreg1 xmmreglo: imm</td>
</tr>
</tbody>
</table>

**VPSLLD — Shift Packed Data Left Logical**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F2:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1     | C4: rxb0_1: w xmmreg2 001:F2:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:F2:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r xmmreglo2 001:F2:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8  | C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:72:11 xmmreg1 xmmreglo: imm |

**VPSLLQ — Shift Packed Data Left Logical**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F3:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1     | C4: rxb0_1: w xmmreg2 001:F3:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:F3:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r xmmreglo2 001:F3:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8  | C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:73:11 xmmreg1 xmmreglo: imm |

**VPSRAW — Shift Packed Data Right Arithmetic**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:E1:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1     | C4: rxb0_1: w xmmreg2 001:E1:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:E1:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r xmmreglo2 001:E1:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8  | C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:71:11 xmmreg1 xmmreglo: imm |

**VPSRAD — Shift Packed Data Right Arithmetic**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:E2:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1     | C4: rxb0_1: w xmmreg2 001:E2:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:E2:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r xmmreglo2 001:E2:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8  | C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:72:11 xmmreg1 xmmreglo: imm |

**VPSRLDQ — Shift Double Quadword Right Logical**

| xmmreg2 to xmmreg1 using imm8  | C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:73:11 xmmreg1 xmmreglo: imm |

**VPSRLW — Shift Packed Data Right Logical**

<p>| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:D1:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1     | C4: rxb0_1: w xmmreg2 001:D1:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:D1:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r xmmreglo2 001:D1:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8  | C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo to xmmreg1 using imm8</td>
<td>C5: r_F 001:71:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td><strong>VPSRLD — Shift Packed Data Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D2:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D2:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:D2:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:D2:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreglo2: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm8</td>
<td>C5: r_F 001:72:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td><strong>VPSRLQ — Shift Packed Data Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D3:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D3:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:D3:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:D3:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreglo2: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm8</td>
<td>C5: r_F 001:73:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td><strong>VPSUBB — Subtract Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F8:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F8:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBW — Subtract Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F9:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F9:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmrelog2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBD — Subtract Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FA:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:FA:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmrelog2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:FA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBQ — Subtract Packed Quadword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FB:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:FB:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmrelog2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:FB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBSB — Subtract Packed Signed Integers with Signed Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EB:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EB:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>
### Instruction and Format Encoding

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E8:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBSW — Subtract Packed Signed Integers with Signed Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E9:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E9:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBUSB — Subtract Packed Unsigned Integers with Unsigned Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D8:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D8:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKHBW — Unpack High Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:68:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:68:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:68:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:68:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKHWD — Unpack High Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:69:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:69:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:69:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:69:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKHDQ — Unpack High Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6A:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:6A:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:6A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKHDQ — Unpack High Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:6D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:6D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>VPUNPCKLBW — Unpack Low Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:60:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:60:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:60:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:60:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKLWD — Unpack Low Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:61:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:61:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:61:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:61:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKLDQ — Unpack Low Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:62:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:62:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:62:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:62:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKLQDQ — Unpack Low Data</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:6C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:6C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPXOR — Logical Exclusive OR</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EF:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EF:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:EF:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:EF:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VSHUFPD — Shuffle Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w xmmreg2 001:C6:11 xmmreg1 xmmreg3:imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w xmmreg2 001:C6:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1 using imm8</td>
<td>C5: r xmmreglo2 001:C6:11 xmmreg1 xmmreglo3:imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1 using imm8</td>
<td>C4: rxb0_1: w ymmreg2 101:C6:11 ymmreg1 ymmreg3:imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1 using imm8</td>
<td>C4: rxb0_1: w ymmreg2 101:C6:mod ymmreg1 r/m:imm</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1 using imm8</td>
<td>C5: r ymmreglo2 101:C6:11 ymmreg1 ymmreglo3:imm</td>
</tr>
<tr>
<td><strong>VSQRTPD — Compute Square Roots of Packed Double-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:51:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:51:11 xmmreg1 xmmreglo</td>
</tr>
</tbody>
</table>
### Instruction and Format

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:51:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:51:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 to ymmreg1</td>
<td>C5: r_F 101:51:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:51:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VSQRTSD — Compute Square Root of Scalar Double-Precision Floating-Point Value**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:51:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:51:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:51:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VSUBPD — Subtract Packed Double-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5C:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VSUBSD — Subtract Scalar Double-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5C:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VUCOMISD — Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg1, set EFLAGS</td>
<td>C4: rxb0_1: w_F xmmreg1 001:2E:11 xmmreg2</td>
</tr>
<tr>
<td>mem with xmmreg1, set EFLAGS</td>
<td>C4: rxb0_1: w_F xmmreg1 001:2E:mod r/m</td>
</tr>
<tr>
<td>xmmreglo with xmmreg1, set EFLAGS</td>
<td>C5: r_F xmmreg1 001:2E:11 xmmreglo</td>
</tr>
<tr>
<td>mem with xmmreg1, set EFLAGS</td>
<td>C5: r_F xmmreg1 001:2E:mod r/m</td>
</tr>
</tbody>
</table>

**VUNPCKHPD — Unpack and Interleave High Packed Double-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:15:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:15:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:15:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:15:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:15:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:15:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>
### Instruction and Format Encoding

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:15:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:15:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VUNPCKHPS — Unpack and Interleave High Packed Single-Precision Floating-Point Values

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 000:15:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 000:15:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 000:15:mod xmmreg1 r/m |
| ymmreg2 with ymmreglo3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:15:11 ymmreg1 ymmreglo3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 100:15:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r ymmreglo2 100:15:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r ymmreglo2 100:15:mod ymmreg1 r/m |

#### VUNPCKLPS — Unpack and Interleave Low Packed Single-Precision Floating-Point Values

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 000:14:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 000:14:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 000:14:mod xmmreg1 r/m |
| ymmreg2 with ymmreglo3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:14:11 ymmreg1 ymmreglo3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 100:14:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r ymmreglo2 100:14:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r ymmreglo2 100:14:mod ymmreg1 r/m |

#### VXORPD — Bitwise Logical XOR for Double-Precision Floating-Point Values

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:57:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:57:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 001:57:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 001:57:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 101:57:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 101:57:mod ymmreg1 r/m |
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:57:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:57:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VADDPS — Add Packed Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:58:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 000:58:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 000:58:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 000:58:mod xmmreg1 r/m |
| ymmreg2 with ymmreglo3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:58:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 100:58:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r ymmreglo2 100:58:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r ymmreglo2 100:58:mod ymmreg1 r/m |

**VADDSS — Add Scalar Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 010:58:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 010:58:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 010:58:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 010:58:mod xmmreg1 r/m |

**VANDPS — Bitwise Logical AND of Packed Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:54:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 000:54:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 000:54:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 000:54:mod xmmreg1 r/m |
| ymmreg2 with ymmreglo3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:54:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 100:54:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r ymmreglo2 100:54:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r ymmreglo2 100:54:mod ymmreg1 r/m |

**VANDNPS — Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:55:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 000:55:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r xmmreglo2 000:55:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r xmmreglo2 000:55:mod xmmreg1 r/m |
| ymmreg2 with ymmreglo3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:55:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 100:55:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r ymmreglo2 100:55:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r ymmreglo2 100:55:mod ymmreg1 r/m |

**VCMPPS — Compare Packed Single-Precision Floating-Point Values**

<p>| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:C2:11 xmmreg1 xmmreg3:imm |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:C2:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:C2:11 xmmreg1 xmmreglo3: imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:C2:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:C2:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:C2:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 100:C2:11 ymmreg1 ymmreglo3: imm</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 100:C2:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VCMPSS — Compare Scalar Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w 010:C2:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w 010:C2:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 010:C2:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 010:C2:mod xmmreg1 r/m |

**VCOMISS — Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS**

| xmmreg2 with xmmreg1 | C4: rxb0_1: w_F 000:2F:11 xmmreg1 xmmreg2 |
| mem with xmmreg1 | C4: rxb0_1: w_F 000:2F:mod xmmreg1 r/m |
| xmmreglo with xmmreg1 | C5: r_F 000:2F:11 xmmreg1 xmmreglo |
| mem with xmmreg1 | C5: r_F 000:2F:mod xmmreg1 r/m |

**VCVTSI2SS — Convert Dword Integer to Scalar Single-Precision FP Value**

| xmmreg2 with reg to xmmreg1 | C4: rxb0_1: 0_F 010:2A:11 xmmreg1 reg |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: 0_F 010:2A:mod xmmreg1 r/m |
| xmmreglo2 with reglo to xmmreg1 | C5: r_xmmreglo2 010:2A:11 xmmreg1 reglo |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 010:2A:mod xmmreg1 r/m |

**VCVTSS2SI — Convert Scalar Single-Precision FP Value to Dword Integer**

| xmmreg1 to reg | C4: rxb0_1: 0_F 010:2D:11 reg xmmreg1 |
| mem to reg | C4: rxb0_1: 0_F 010:2D:mod reg r/m |
| xmmreglo to reg | C5: r_F 010:2D:11 reg xmmreglo |
| mem to reg | C5: r_F 010:2D:mod reg r/m |

**VCVTTS2SI — Convert with Truncation Scalar Single-Precision FP Value to Dword Integer**

| xmmreg1 to reg | C4: rxb0_1: 0_F 010:2C:11 reg xmmreg1 |
| mem to reg | C4: rxb0_1: 0_F 010:2C:mod reg r/m |
| xmmreglo to reg | C5: r_F 010:2C:11 reg xmmreglo |
| mem to reg | C5: r_F 010:2C:mod reg r/m |
## INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg1 to reg</td>
<td>C4: rxb0_1: 1_F 010:2C:11 reg xmmreg1</td>
</tr>
<tr>
<td>mem to reg</td>
<td>C4: rxb0_1: 1_F 010:2C:mod reg r/m</td>
</tr>
<tr>
<td><strong>VDIVPS — Divide Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5E:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 000:5E:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 000:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:5E:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:5E:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 100:5E:11 ymmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to ymmreg1</td>
<td>C5: r xmmreglo2 100:5E:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VDIVSS — Divide Scalar Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5E:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 010:5E:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 010:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VLDMXCSR — Load MXCSR Register</strong></td>
<td></td>
</tr>
<tr>
<td>mem to MXCSR reg</td>
<td>C4: rxb0_1: w_F 000:AEmod 011 r/m</td>
</tr>
<tr>
<td>mem to MXCSR reg</td>
<td>C5: r_F 000:AEmod 011 r/m</td>
</tr>
<tr>
<td><strong>VMAXPS — Return Maximum Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5F:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 000:5F:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 000:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VMAXSS — Return Maximum Scalar Single-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5F:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 010:5F:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 010:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VMINPS — Return Minimum Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5D:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>
### Instruction and Format

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:5D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:5D:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:5D:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 100:5D:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 100:5D:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VMINSS</strong> — Return Minimum Scalar Single-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VMOVAPS</strong>— Move Aligned Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:28:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:28:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:28:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:28:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 000:29:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 000:29:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 000:29:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 000:29:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:28:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:28:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to ymmreg1</td>
<td>C5: r_F 100:28:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:28:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 100:29:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 100:29:mod r/m ymmreg1</td>
</tr>
<tr>
<td><strong>VMOVHPS</strong> — Move High Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 with mem to xmmreg2</td>
<td>C4: rxb0_1: w xmmreg1 000:16:mod xmmreg2 r/m</td>
</tr>
<tr>
<td>xmmreg1 with mem to xmmreglo2</td>
<td>C5: r_xmmreg1 000:16:mod xmmreglo2 r/m</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 000:17:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 000:17:mod r/m xmmreglo</td>
</tr>
<tr>
<td><strong>VMOVLHPS</strong> — Move Packed Single-Precision Floating-Point Values Low to High</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:16:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:16:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>VMOVLPS — Move Low Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 with mem to xmmreg2</td>
<td>C4: rxb0_1: w xmmreg1 000:12:mod xmmreg2 r/m</td>
</tr>
<tr>
<td>xmmreg1 with mem to xmmreglo2</td>
<td>C5: r_xmmreg1 000:12:mod xmmreglo2 r/m</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 000:13:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 000:13:mod r/m xmmreglo</td>
</tr>
<tr>
<td>VMOVMSKPS — Extract Packed Single-Precision Floating-Point Sign Mask</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to reg</td>
<td>C4: rxb0_1: w_F 000:50:11 reg xmmreg2</td>
</tr>
<tr>
<td>xmmreglo to reg</td>
<td>C5: r_F 000:50:11 reg xmmreglo</td>
</tr>
<tr>
<td>ymmreg2 to reg</td>
<td>C4: rxb0_1: w_F 100:50:11 reg ymmreg2</td>
</tr>
<tr>
<td>ymmreglo to reg</td>
<td>C5: r_F 100:50:11 reg ymmreglo</td>
</tr>
<tr>
<td>VMOVNTPS — Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 000:2B:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 000:2B:mod r/m xmmreglo</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 100:2B:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreglo to mem</td>
<td>C5: r_F 100:2B:mod r/m ymmreglo</td>
</tr>
<tr>
<td>VMOVSS — Move Scalar Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:10:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C5: r_xmmreg2 010:10:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 010:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:11:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 010:11:mod r/m xmmreglo</td>
</tr>
<tr>
<td>VMOVUPS — Move Unaligned Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:10:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:10:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:10:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 100:10:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 000:11:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 000:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 000:11:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 000:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 100:11:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 100:11:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 100:11:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 100:11:mod r/m ymmreg1</td>
</tr>
</tbody>
</table>

**VMULPS — Multiply Packed Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:59:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 000:59:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 000:59:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r_xmmreglo2 000:59:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:59:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1    | C4: rxb0_1: w ymmreg2 100:59:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r_ymmreglo2 100:59:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1  | C5: r_ymmreglo2 100:59:mod ymmreg1 r/m |

**VMULSS — Multiply Scalar Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 010:59:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 010:59:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 010:59:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r_xmmreglo2 010:59:mod xmmreg1 r/m |

**VORPS — Bitwise Logical OR of Single-Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:56:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1    | C4: rxb0_1: w xmmreg2 000:56:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 000:56:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1  | C5: r_xmmreglo2 000:56:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:56:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1    | C4: rxb0_1: w ymmreg2 100:56:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r_ymmreglo2 100:56:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1  | C5: r_ymmreglo2 100:56:mod ymmreg1 r/m |

**VRCPPS — Compute Reciprocals of Packed Single-Precision Floating-Point Values**

| xmmreg2 to xmmreg1        | C4: rxb0_1: w_F 000:53:11 xmmreg1 xmmreg2 |
| mem to xmmreg1            | C4: rxb0_1: w_F 000:53:mod xmmreg1 r/m |
| xmmreglo to xmmreg1       | C5: r_F 000:53:11 xmmreg1 xmmreglo |
| mem to xmmreg1            | C5: r_F 000:53:mod xmmreg1 r/m |
| ymmreg2 to ymmreg1         | C4: rxb0_1: w_F 100:53:11 ymmreg1 ymmreg2 |
| mem to ymmreg1             | C4: rxb0_1: w_F 100:53:mod ymmreg1 r/m |
| ymmreglo to ymmreg1        | C5: r_F 100:53:11 ymmreg1 ymmreglo |
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:53:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VRCPPSS — Compute Reciprocal of Scalar Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:53:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:53:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:53:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:53:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VRSQRTPS — Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:52:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:52:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:52:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:52:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:52:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:52:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 100:52:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:52:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VRSQRTSS — Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:52:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:52:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:52:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:52:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VSHUFPS — Shuffle Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1, imm8</td>
<td>C4: rxb0_1: w xmmreg2 000:C6:11 xmmreg1 xmmreg3:imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1, imm8</td>
<td>C4: rxb0_1: w xmmreg2 000:C6:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1, imm8</td>
<td>C5: r_xmmreglo2 000:C6:11 xmmreg1 xmmreglo3:imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1, imm8</td>
<td>C5: r_xmmreglo2 000:C6:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1, imm8</td>
<td>C4: rxb0_1: w ymmreg2 100:C6:11 ymmreg1 ymmreg3:imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1, imm8</td>
<td>C4: rxb0_1: w ymmreg2 100:C6:mod ymmreg1 r/m:imm</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1, imm8</td>
<td>C5: r_ymreglo2 100:C6:11 ymmreg1 ymmreglo3:imm</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1, imm8</td>
<td>C5: r_ymreglo2 100:C6:mod ymmreg1 r/m:imm</td>
</tr>
<tr>
<td><strong>VSVQRTPS — Compute Square Roots of Packed Single-Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:51:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:51:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:51:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:51:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 100:51:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:51:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VSQRTSS — Compute Square Root of Scalar Single-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:51:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:51:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VSTMXCSR — Store MXCSR Register State</td>
<td>C4: rxb0_1: w_F 000:AE:mod 011 r/m</td>
</tr>
<tr>
<td>MXCSR to mem</td>
<td>C4: r_F 000:AE:mod 011 r/m</td>
</tr>
<tr>
<td>VSUBPS — Subtract Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:5C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VSUBSS — Subtract Scalar Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VUCOMISS — Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg1</td>
<td>C4: rxb0_1: w_F 000:2E:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem with xmmreg1</td>
<td>C4: rxb0_1: w_F 000:2E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo with xmmreg1</td>
<td>C5: r_F 000:2E:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem with xmmreg1</td>
<td>C5: r_F 000:2E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>UNPCKHPS — Unpack and Interleave High Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:15:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 000:15:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 000:15:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>UNPCLKLPS — Unpack and Interleave Low Packed Single-Precision Floating-Point Value</td>
<td></td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:14:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:14:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:14:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VXORPS — Bitwise Logical XOR for Single-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:57:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:57:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:57:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:57:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:57:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 100:57:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 100:57:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VBROADCAST — Load with Broadcast**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: 0_F 001:18:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_2: 0_F 101:18:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_2: 0_F 101:19:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_2: 0_F 101:1A:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VEXTRACTF128 — Extract Packed Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreg2 to xmmreg1, imm8</td>
<td>C4: rxb0_3: 0_F 001:19:11 xmmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>ymmreg2 to mem, imm8</td>
<td>C4: rxb0_3: 0_F 001:19:mod r/m ymmreg2: imm</td>
</tr>
</tbody>
</table>

**VINSETF128 — Insert Packed Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg3 and merge with ymmreg2 to ymmreg1, imm8</td>
<td>C4: rxb0_3: 0 ymmreg2 201:18:11 ymmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>mem and merge with ymmreg2 to ymmreg1, imm8</td>
<td>C4: rxb0_3: 0 ymmreg2 101:18:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPERMILPD — Permute Double-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: 0 xmmreg2 001:0D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: 0 xmmreg2 001:0D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_2: 0 ymmreg2 101:0D:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_2: 0 ymmreg2 101:0D:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm</td>
<td>C4: rxb0_3: 0_F 001:05:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1, imm</td>
<td>C4: rxb0_3: 0_F 001:05:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm</td>
<td>C4: rxb0_3: 0_F 101:05:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm</td>
<td>C4: rxb0_3: 0_F 101:05:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPERMILPS — Permute Single-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: 0 xmmreg2 001:0C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: 0 xmmreg2 001:0C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm</td>
<td>C4: rxb0_3: 0_F 001:04:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>mem to xmmreg1, imm</td>
<td>C4: rxb0_3:0_F 001:04:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_2:0 ymmreg2 101:0C:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_2:0 ymmreg2 101:0C:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm</td>
<td>C4: rxb0_3:0_F 101:04:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm</td>
<td>C4: rxb0_3:0_F 101:04:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPERM2F128 — Permute Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_3:0 ymmreg2 101:06:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_3:0 ymmreg2 101:06:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VTESTPD/VTESTPS — Packed Bit Test**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2:0_F 001:0E:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2:0_F 001:0E:mod xmmreg2 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_2:0_F 101:0E:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_2:0_F 101:0E:mod ymmreg2 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2:0_F 001:0F:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2:0_F 001:0F:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_2:0_F 101:0F:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_2:0_F 101:0F:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The term "lo" refers to the lower eight registers, 0-7
B.17 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-38 shows the five different formats used for floating-point instructions. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011.

Table B-38. General Floating-Point Instruction Formats

<table>
<thead>
<tr>
<th>Instruction</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>Optional Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11011</td>
<td>OPA</td>
<td>mod 1 OPB r/m s-i-b disp</td>
</tr>
<tr>
<td>2</td>
<td>11011</td>
<td>MF OPA</td>
<td>mod OPB r/m s-i-b disp</td>
</tr>
<tr>
<td>3</td>
<td>11011</td>
<td>d P OPA</td>
<td>1 1 OPB R ST(i)</td>
</tr>
<tr>
<td>4</td>
<td>11011</td>
<td>0 0 1 1 1 1 OP</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11011</td>
<td>0 1 1 1 1 1 OP</td>
<td></td>
</tr>
<tr>
<td>15–11</td>
<td>10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MF** = Memory Format
- 00 — 32-bit real
- 01 — 32-bit integer
- 10 — 64-bit real
- 11 — 16-bit integer

**P** = Pop
- 0 — Do not pop stack
- 1 — Pop stack after operation

**d** = Destination
- 0 — Destination is ST(0)
- 1 — Destination is ST(i)

**R XOR d**:
- 0 — Destination OP Source
- 1 — Source OP Destination

**ST(i)** = Register stack element i
- 000 = Stack Top
- 001 = Second stack element
- 111 = Eighth stack element

The Mod and R/M fields of the ModR/M byte have the same interpretation as the corresponding fields of the integer instructions. The SIB byte and disp (displacement) are optionally present in instructions that have Mod and R/M fields. Their presence depends on the values of Mod and R/M, as for integer instructions.

Table B-39 shows the formats and encodings of the floating-point instructions.

Table B-39. Floating-Point Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2XM1 - Compute 2^{ST(0)} - 1</td>
<td>11011 001 : 1111 0000</td>
</tr>
<tr>
<td>FABS - Absolute Value</td>
<td>11011 001 : 1110 0001</td>
</tr>
<tr>
<td>FADD - Add</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 32-bit memory</td>
<td>11011 000 : mod 000 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 64-bit memory</td>
<td>11011 100 : mod 000 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(0) + ST(i)</td>
<td>11011 d00 : 11 000 ST(i)</td>
</tr>
<tr>
<td>FADDP - Add and Pop</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + ST(i)</td>
<td>11011 110 : 11 000 ST(i)</td>
</tr>
<tr>
<td>FBLD - Load Binary Coded Decimal</td>
<td>11011 111 : mod 100 r/m</td>
</tr>
<tr>
<td>FBSTP - Store Binary Coded Decimal and Pop</td>
<td>11011 111 : mod 110 r/m</td>
</tr>
<tr>
<td>FCHS - Change Sign</td>
<td>11011 001 : 1110 0000</td>
</tr>
<tr>
<td>FCLEX - Clear Exceptions</td>
<td>11011 011 : 1110 0010</td>
</tr>
<tr>
<td>FCOM - Compare Real</td>
<td></td>
</tr>
</tbody>
</table>

B-114 Vol. 2D
### Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit memory</td>
<td>11011 000 : mod 010 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 100 : mod 010 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 000 : 11 010 ST(i)</td>
</tr>
</tbody>
</table>

**FCOMP - Compare Real and Pop**
- 32-bit memory: 11011 000 : mod 011 r/m
- 64-bit memory: 11011 100 : mod 011 r/m
- ST(i): 11011 000 : 11 011 ST(i)

**FCOMPP - Compare Real and Pop Twice**
11011 110 : 11 011 001

**FCOMP - Compare Real, Set EFLAGS, and Pop**
11011 111 : 11 110 ST(i)

**FCOS - Cosine of ST(0)**
11011 101 : 1100 0 ST(i)

**FDECSTP - Decrement Stack-Top Pointer**
11011 001 : 1111 0110

**FDIV - Divide**
- ST(0) ← ST(0) ÷ 32-bit memory: 11011 000 : mod 110 r/m
- ST(0) ← ST(0) ÷ 64-bit memory: 11011 100 : mod 110 r/m
- ST(d) ← ST(0) ÷ ST(i): 11011 d00 : 1111 R ST(i)

**FDIVP - Divide and Pop**
ST(0) ← ST(0) ÷ ST(i): 11011 110 : 1111 ST(i)

**FDIVR - Reverse Divide**
- ST(0) ← 32-bit memory ÷ ST(0): 11011 000 : mod 111 r/m
- ST(0) ← 64-bit memory ÷ ST(0): 11011 100 : mod 111 r/m
- ST(d) ← ST(i) ÷ ST(0): 11011 d00 : 1111 R ST(i)

**FDIVRP - Reverse Divide and Pop**
ST(0) ← ST(i) ÷ ST(0): 11011 110 : 1111 ST(i)

**FFREE - Free ST(i) Register**
11011 101 : 1100 0 ST(i)

**FIADD - Add Integer**
- ST(0) ← ST(0) + 16-bit memory: 11011 110 : mod 000 r/m
- ST(0) ← ST(0) + 32-bit memory: 11011 010 : mod 000 r/m

**FICOM - Compare Integer**
- 16-bit memory: 11011 110 : mod 010 r/m
- 32-bit memory: 11011 010 : mod 010 r/m

**FICOMP - Compare Integer and Pop**
- 16-bit memory: 11011 110 : mod 011 r/m
- 32-bit memory: 11011 010 : mod 011 r/m

**FIDIV - Divide**
- ST(0) ← ST(0) ÷ 16-bit memory: 11011 110 : mod 110 r/m
- ST(0) ← ST(0) ÷ 32-bit memory: 11011 010 : mod 110 r/m

**FIDIVR - Reverse Divide**
ST(0) ← 16-bit memory ÷ ST(0): 11011 110 : mod 111 r/m
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) ← 32-bit memory + ST(0)</td>
<td>11011 010 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FILD</strong> - Load Integer</td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 111 : mod 000 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 011 : mod 000 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 111 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>FIMUL</strong> - Multiply</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) × 16-bit memory</td>
<td>11011 110 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) × 32-bit memory</td>
<td>11011 010 : mod 001 r/m</td>
</tr>
<tr>
<td><strong>FINCSTP</strong> - Increment Stack Pointer</td>
<td>11011 001 : 1111 0111</td>
</tr>
<tr>
<td><strong>FINIT</strong> - Initialize Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td><strong>FIST</strong> - Store Integer</td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 111 : mod 010 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 011 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>FISTP</strong> - Store Integer and Pop</td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 111 : mod 011 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 011 : mod 011 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 111 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FISUB</strong> - Subtract</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) - 16-bit memory</td>
<td>11011 110 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) - 32-bit memory</td>
<td>11011 010 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>FISUBR</strong> - Reverse Subtract</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← 16-bit memory − ST(0)</td>
<td>11011 110 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(0) ← 32-bit memory − ST(0)</td>
<td>11011 010 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>FLD</strong> - Load Real</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 000 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 000 r/m</td>
</tr>
<tr>
<td>80-bit memory</td>
<td>11011 011 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 001 : 11 000 ST(i)</td>
</tr>
<tr>
<td><strong>FLD1</strong> - Load +1.0 into ST(0)</td>
<td>11011 001 : 1110 1000</td>
</tr>
<tr>
<td><strong>FLDCW</strong> - Load Control Word</td>
<td>11011 001 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>FLDENV</strong> - Load FPU Environment</td>
<td>11011 001 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>FLDL2E</strong> - Load $\log_2(e)$ into ST(0)</td>
<td>11011 001 : 1110 1010</td>
</tr>
<tr>
<td><strong>FLDL2T</strong> - Load $\log_2(10)$ into ST(0)</td>
<td>11011 001 : 1110 1001</td>
</tr>
<tr>
<td><strong>FLDLOG2</strong> - Load $\log_2(10)$ into ST(0)</td>
<td>11011 001 : 1110 1100</td>
</tr>
<tr>
<td><strong>FLDLNZ</strong> - Load $\log_2(2)$ into ST(0)</td>
<td>11011 001 : 1110 1101</td>
</tr>
<tr>
<td><strong>FLDPI</strong> - Load $\pi$ into ST(0)</td>
<td>11011 001 : 1110 1011</td>
</tr>
<tr>
<td><strong>FLDZ</strong> - Load +0.0 into ST(0)</td>
<td>11011 001 : 1110 1110</td>
</tr>
<tr>
<td><strong>FMUL</strong> - Multiply</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) ← ST(0) × 32-bit memory</td>
<td>11011 000 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) × 64-bit memory</td>
<td>11011 100 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(0) × ST(i)</td>
<td>11011 d00 : 1100 1 ST(i)</td>
</tr>
<tr>
<td><strong>FMULP - Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>ST(i) ← ST(0) × ST(i)</td>
<td>11011 110 : 1100 1 ST(i)</td>
</tr>
<tr>
<td><strong>FNOP - No Operation</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1101 0000</td>
</tr>
<tr>
<td><strong>FPATAN - Partial Arctangent</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 0011</td>
</tr>
<tr>
<td><strong>FPREM - Partial Remainder</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 1000</td>
</tr>
<tr>
<td><strong>FPREM1 - Partial Remainder (IEEE)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 0101</td>
</tr>
<tr>
<td><strong>FPTAN - Partial Tangent</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 0010</td>
</tr>
<tr>
<td><strong>FRNDINT - Round to Integer</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 1100</td>
</tr>
<tr>
<td><strong>FRSTOR - Restore FPU State</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 101 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>FSAVE - Store FPU State</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 101 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>FScale - Scale</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 1101</td>
</tr>
<tr>
<td><strong>FSIN - Sine</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 1110</td>
</tr>
<tr>
<td><strong>FSINCOS - Sine and Cosine</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 1011</td>
</tr>
<tr>
<td><strong>FSQRT - Square Root</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 1010</td>
</tr>
<tr>
<td><strong>FST - Store Real</strong></td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 010 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 010 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 101 : 11 010 ST(i)</td>
</tr>
<tr>
<td><strong>FSTCW - Store Control Word</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FSTENV - Store FPU Environment</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>FSTP - Store Real and Pop</strong></td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 011 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 011 r/m</td>
</tr>
<tr>
<td>80-bit memory</td>
<td>11011 011 : mod 111 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 101 : 11 011 ST(i)</td>
</tr>
<tr>
<td><strong>FSTSW - Store Status Word into AX</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 111 : 1110 0000</td>
</tr>
<tr>
<td><strong>FSTSW - Store Status Word into Memory</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 101 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FSUB - Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) – 32-bit memory</td>
<td>11011 000 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) – 64-bit memory</td>
<td>11011 100 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(0) – ST(i)</td>
<td>11011 d00 : 1110 R ST(i)</td>
</tr>
<tr>
<td><strong>FSUBP - Subtract and Pop</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) – ST(i)</td>
<td>11011 110 : 1110 1 ST(i)</td>
</tr>
<tr>
<td><strong>FSUBR - Reverse Subtract</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 000 : mod 101 r/m</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) ← 64-bit memory – ST(0)</td>
<td>11011 100 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(i) – ST(0)</td>
<td>11011 d00 : 1110 R ST(i)</td>
</tr>
<tr>
<td>FSUBRP – Reverse Subtract and Pop</td>
<td></td>
</tr>
<tr>
<td>ST(i) ← ST(i) – ST(0)</td>
<td>11011 110 : 1110 0 ST(i)</td>
</tr>
<tr>
<td>FTST – Test</td>
<td>11011 001 : 1110 0100</td>
</tr>
<tr>
<td>FUCOM – Unordered Compare Real</td>
<td>11011 101 : 1110 0 ST(i)</td>
</tr>
<tr>
<td>FUCOMP – Unordered Compare Real and Pop</td>
<td>11011 101 : 1110 1 ST(i)</td>
</tr>
<tr>
<td>FUCOMPP – Unordered Compare Real and Pop Twice</td>
<td>11011 010 : 1110 1001</td>
</tr>
<tr>
<td>FUCOMI – Unordered Compare Real and Set EFLAGS</td>
<td>11011 011 : 11 101 ST(i)</td>
</tr>
<tr>
<td>FUCOMIP – Unordered Compare Real, Set EFLAGS, and Pop</td>
<td>11011 111 : 11 101 ST(i)</td>
</tr>
<tr>
<td>FXAM – Examine</td>
<td>11011 001 : 1110 0101</td>
</tr>
<tr>
<td>FXCH – Exchange ST(0) and ST(i)</td>
<td>11011 001 : 1100 1 ST(i)</td>
</tr>
<tr>
<td>FXTRACT – Extract Exponent and Significand</td>
<td>11011 001 : 1111 0100</td>
</tr>
<tr>
<td>FYL2X – ST(1) × log2(ST(0))</td>
<td>11011 001 : 1111 0001</td>
</tr>
<tr>
<td>FYL2XP1 – ST(1) × log2(ST(0) + 1.0)</td>
<td>11011 001 : 1111 1001</td>
</tr>
<tr>
<td>FWAIT – Wait until FPU Ready</td>
<td>1001 1011 (same instruction as WAIT)</td>
</tr>
</tbody>
</table>

B.18 VMX INSTRUCTIONS

Table B-40 describes virtual-machine extensions (VMX).

Table B-40. Encodings for VMX Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVEPT—Invalidate Cached EPT Mappings</td>
<td></td>
</tr>
<tr>
<td>Descriptor m128 according to reg</td>
<td>01100110 00001111 00111000 10000000: mod reg r/m</td>
</tr>
<tr>
<td>INVPID—Invalidate Cached VPID Mappings</td>
<td></td>
</tr>
<tr>
<td>Descriptor m128 according to reg</td>
<td>01100110 00001111 00111000 10000001: mod reg r/m</td>
</tr>
<tr>
<td>VMCALL—Call to VM Monitor</td>
<td></td>
</tr>
<tr>
<td>Call VMM: causes VM exit.</td>
<td>00001111 00000001 11000001</td>
</tr>
<tr>
<td>VMCLEAR—Clear Virtual-Machine Control Structure</td>
<td></td>
</tr>
<tr>
<td>mem32:VMCS_data_ptr</td>
<td>01100110 00001111 11000111: mod 110 r/m</td>
</tr>
<tr>
<td>mem64:VMCS_data_ptr</td>
<td>01100110 00001111 11000111: mod 110 r/m</td>
</tr>
<tr>
<td>VMFUNC—Invoke VM Function</td>
<td></td>
</tr>
<tr>
<td>Invoke VM function specified in EAX</td>
<td>00001111 00000001 11010100</td>
</tr>
<tr>
<td>VMLAUNCH—Launch Virtual Machine</td>
<td></td>
</tr>
<tr>
<td>Launch VM managed by Current_VMCS</td>
<td>00001111 00000001 11000010</td>
</tr>
<tr>
<td>VMRESUME—Resume Virtual Machine</td>
<td></td>
</tr>
<tr>
<td>Resume VM managed by Current_VMCS</td>
<td>00001111 00000001 11000011</td>
</tr>
<tr>
<td>VMPTRLD—Load Pointer to Virtual-Machine Control Structure</td>
<td></td>
</tr>
</tbody>
</table>
Table B-40. Encodings for VMX Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem32 to Current_VMCS_ptr</td>
<td>00001111 11000111: mod 110 r/m</td>
</tr>
<tr>
<td>mem64 to Current_VMCS_ptr</td>
<td>00001111 11000111: mod 110 r/m</td>
</tr>
<tr>
<td>VMMPTRST—Store Pointer to Virtual-Machine Control Structure</td>
<td></td>
</tr>
<tr>
<td>Current_VMCS_ptr to mem32</td>
<td>00001111 11000111: mod 111 r/m</td>
</tr>
<tr>
<td>Current_VMCS_ptr to mem64</td>
<td>00001111 11000111: mod 111 r/m</td>
</tr>
<tr>
<td>VMREAD—Read Field from Virtual-Machine Control Structure</td>
<td></td>
</tr>
<tr>
<td>r32 ((VMCS_fieldn)) to r32</td>
<td>00001111 01111000: 11 reg2 reg1</td>
</tr>
<tr>
<td>r32 ((VMCS_fieldn)) to mem32</td>
<td>00001111 01111000: mod r32 r/m</td>
</tr>
<tr>
<td>r64 ((VMCS_fieldn)) to r64</td>
<td>00001111 01111000: 11 reg2 reg1</td>
</tr>
<tr>
<td>r64 ((VMCS_fieldn)) to mem64</td>
<td>00001111 01111000: mod r64 r/m</td>
</tr>
<tr>
<td>VMWRITE—Write Field to Virtual-Machine Control Structure</td>
<td></td>
</tr>
<tr>
<td>r32 to r32 ((VMCS_fieldn))</td>
<td>00001111 01111001: 11 reg1 reg2</td>
</tr>
<tr>
<td>mem32 to r32 ((VMCS_fieldn))</td>
<td>00001111 01111001: mod r32 r/m</td>
</tr>
<tr>
<td>r64 to r64 ((VMCS_fieldn))</td>
<td>00001111 01111001: 11 reg1 reg2</td>
</tr>
<tr>
<td>mem64 to r64 ((VMCS_fieldn))</td>
<td>00001111 01111001: mod r64 r/m</td>
</tr>
<tr>
<td>VMXOFF—Leave VMX Operation</td>
<td>Leave VMX.</td>
</tr>
<tr>
<td>VMXON—Enter VMX Operation</td>
<td>00001111 00000001 11000100</td>
</tr>
</tbody>
</table>

B.19 SMX INSTRUCTIONS

Table B-38 describes Safer Mode extensions (VMX). **GETSEC leaf functions are selected by a valid value in EAX on input.**

Table B-41. Encodings for SMX Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETSEC—GETSEC leaf functions are selected by the value in EAX on input</td>
<td></td>
</tr>
<tr>
<td>GETSEC[CAPABILITIES]</td>
<td>00001111 00110111 () (EAX= 0)</td>
</tr>
<tr>
<td>GETSEC[ENTERACCS]</td>
<td>00001111 00110111 () (EAX= 2)</td>
</tr>
<tr>
<td>GETSEC[EXITAC]</td>
<td>00001111 00110111 () (EAX= 3)</td>
</tr>
<tr>
<td>GETSEC[SENTER]</td>
<td>00001111 00110111 () (EAX= 4)</td>
</tr>
<tr>
<td>GETSEC[SEXIT]</td>
<td>00001111 00110111 () (EAX= 5)</td>
</tr>
<tr>
<td>GETSEC[PARAMETERS]</td>
<td>00001111 00110111 () (EAX= 6)</td>
</tr>
<tr>
<td>GETSEC[SMCTRL]</td>
<td>00001111 00110111 () (EAX= 7)</td>
</tr>
<tr>
<td>GETSEC[WAKEUP]</td>
<td>00001111 00110111 () (EAX= 8)</td>
</tr>
</tbody>
</table>
The two tables in this appendix itemize the Intel C/C++ compiler intrinsics and functional equivalents for the Intel MMX technology, SSE, SSE2, SSE3, and SSSE3 instructions.

There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics. Please refer to http://www.intel.com/support/performancetools/.

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are “composites” because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

```
_mm_<intrin_op>_<suffix>
```

where:

- `<intrin_op>` Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction.
- `<suffix>` Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s).

The remaining letters denote the type:

- `s` single-precision floating point
- `d` double-precision floating point
- `i128` signed 128-bit integer
- `i64` signed 64-bit integer
- `u64` unsigned 64-bit integer
- `i32` signed 32-bit integer
- `u32` unsigned 32-bit integer
- `i16` signed 16-bit integer
- `u16` unsigned 16-bit integer
- `i8` signed 8-bit integer
- `u8` unsigned 8-bit integer

The variable `r` is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest word of `r`.

The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```c
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```c
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the XMM register that holds the value `t` will look as follows:

```
| 127 | 64 | 63 | 0 |
```

```
  2.0
  1.0
```
The "scalar" element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

```
data_type intrinsic_name (parameters)
```

Where:
- `data_type` is the return data type, which can be either void, int, __m64, __m128, __m128d, or __m128i. Only the _mm_empty intrinsic returns void.
- `intrinsic_name` is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
- `parameters` represents the parameters required by each intrinsic.

## C.1 SIMPLE INTRINSICS

**NOTE**

### Table C.1. Simple Intrinsics

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD</td>
<td>__m128d __m_add_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ADDPS</td>
<td>__m128 __m_add_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>ADDSD</td>
<td>__m128d __m_add_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ADDSS</td>
<td>__m128 __m_add_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td>__m128d __m_addsub_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td>__m128 __m_addsub_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>AESDEC</td>
<td>__m128i __m_aesdec (__m128i, __m128i)</td>
</tr>
<tr>
<td>AESDECLAST</td>
<td>__m128i __m_aesdeclast (__m128i, __m128i)</td>
</tr>
<tr>
<td>AESENCE</td>
<td>__m128i __m_aesenc (__m128i, __m128i)</td>
</tr>
<tr>
<td>AESENCLAST</td>
<td>__m128i __m_aesenclast (__m128i, __m128i)</td>
</tr>
<tr>
<td>AESIMC</td>
<td>__m128i __m_aesimc (__m128i)</td>
</tr>
<tr>
<td>AESKEYGENASSIST</td>
<td>__m128i __m_aesimc (__m128i, const int)</td>
</tr>
<tr>
<td>ANDNPD</td>
<td>__m128d __m_andnot_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ANDNPS</td>
<td>__m128 __m_andnot_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>ANDPD</td>
<td>__m128d __m_and_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ANDPS</td>
<td>__m128 __m_and_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>BLENDPD</td>
<td>__m128d __m_blend_pd(__m128d v1, __m128d v2, const int mask)</td>
</tr>
<tr>
<td>BLENDPS</td>
<td>__m128 __m_blend_ps(__m128 v1, __m128 v2, const int mask)</td>
</tr>
<tr>
<td>BLENDVPD</td>
<td>__m128d __m_blendv_pd(__m128d v1, __m128d v2, __m128d v3)</td>
</tr>
<tr>
<td>BLENDVPS</td>
<td>__m128 __m_blendv_ps(__m128 v1, __m128 v2, __m128 v3)</td>
</tr>
<tr>
<td>CLFLUSH</td>
<td>void __m_clflush(void const *)</td>
</tr>
<tr>
<td>CMPPD</td>
<td>__m128d __m_cmpreq_pd(__m128d a, __m128d b)</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPPS</td>
<td>__m128d _mm_cmpeq_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmplt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmple_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpgt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpge_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpneq_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpngt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpnge_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpord_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpunord_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpnle_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>CMPSD</td>
<td>__m128d _mm_cmpeq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmplt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmple_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpgt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpge_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpneq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpngt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpnge_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpord_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpunord_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>CMPSS</td>
<td>__m128d _mm_cmpeq_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmplt_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmple_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpgt_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpge_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpneq_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpnlt_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpngt_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpnge_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpord_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cmpunord_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__m128_mm_cmpnlt_ss(__m128 a, __m128 b)</td>
<td>__m128_mm_cmpnle_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>__m128_mm_cmpngt_ss(__m128 a, __m128 b)</td>
<td>__m128_mm_cmpnge_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>__m128_mm_cmpord_ss(__m128 a, __m128 b)</td>
<td>__m128_mm_cmpunord_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>COMISD</td>
<td>int __mm_comieq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comilt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comile_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comigt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comige_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comineq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>COMISS</td>
<td>int __mm_comieq_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comilt_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comile_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comigt_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comige_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>int __mm_comineq_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>CRC32</td>
<td>unsigned int __mm_crc32_u8(unsigned int crc, unsigned char data)</td>
</tr>
<tr>
<td></td>
<td>unsigned int __mm_crc32_u16(unsigned int crc, unsigned short data)</td>
</tr>
<tr>
<td></td>
<td>unsigned int __mm_crc32_u32(unsigned int crc, unsigned int data)</td>
</tr>
<tr>
<td></td>
<td>unsigned __int64 __int64_mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)</td>
</tr>
<tr>
<td>CVTDQ2PD</td>
<td>__m128d_mm_cvtepi32_pd(__m128i a)</td>
</tr>
<tr>
<td>CVTDQ2PS</td>
<td>__m128d_mm_cvtepi32_ps(__m128i a)</td>
</tr>
<tr>
<td>CVTPD2DQ</td>
<td>__m128i_mm_cvtpd_epi32(__m128d a)</td>
</tr>
<tr>
<td>CVTPD2PI</td>
<td>__m64d_mm_cvtpd_epi32(__m128d a)</td>
</tr>
<tr>
<td>CVTPD2PS</td>
<td>__m128d_mm_cvtpd_ps(__m128d a)</td>
</tr>
<tr>
<td>CVTPPI2PD</td>
<td>__m128d_mm_cvtpi32_pd(__m64 a)</td>
</tr>
<tr>
<td>CVTPPI2PS</td>
<td>__m128d_mm_cvtpi32_ps(__m128d a, __m64 b)</td>
</tr>
<tr>
<td>CVTPS2DQ</td>
<td>__m128i_mm_cvtpsi32(__m128 a)</td>
</tr>
<tr>
<td>CVTPS2PD</td>
<td>__m128d_mm_cvtpsi32(__m128 a)</td>
</tr>
<tr>
<td>CVTPS2PI</td>
<td>__m64d_mm_cvtpsi32(__m128 a)</td>
</tr>
<tr>
<td>CVTSD2SI</td>
<td>int __mm_cvtsd_si32(__m128d a)</td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td>__m128d_mm_cvtsd_ss(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>CVTSI2SD</td>
<td>__m128d_mm_cvtsi32_sd(__m128d a, int b)</td>
</tr>
<tr>
<td>CVTSI2SS</td>
<td>__m128d_mm_cvtsi32_ss(__m128d a, int b)</td>
</tr>
<tr>
<td>CVTSS2SD</td>
<td>__m128d_mm_cvtsi32_sd64(__m128d a, int b)</td>
</tr>
<tr>
<td>CVTSS2SI</td>
<td>int __mm_cvtsi32_si32(__m128 a)</td>
</tr>
<tr>
<td></td>
<td>int __mm_cvtsi32_si32(__m128 a)</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTTPD2DQ</td>
<td>_m128i_mm_cvttpd_epi32(_m128d a)</td>
</tr>
<tr>
<td>CVTTPD2PI</td>
<td>_m64_mm_cvttppd_pi32(_m128d a)</td>
</tr>
<tr>
<td>CVTTPS2DQ</td>
<td>_m128i_mm_cvttpps_epi32(_m128 a)</td>
</tr>
<tr>
<td>CVTTPS2PI</td>
<td>_m64_mm_cvtttps_pi32(_m128 a)</td>
</tr>
<tr>
<td>CVTTSD2SI</td>
<td>_mm_cvtttsd_si32(_m128d a)</td>
</tr>
<tr>
<td>CVTTSS2SI</td>
<td>_mm_cvttts2si(_m128 a)</td>
</tr>
<tr>
<td>DIVPD</td>
<td>_m128d_mm_div_pd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>DIVPS</td>
<td>_m128_mm_div_ps(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>DIVSD</td>
<td>_m128d_mm_div_sd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>DIVSS</td>
<td>_m128mm_div_ss(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>DPPD</td>
<td>_m128d_mm_dp_pd(_m128d a, __m128d b, const int mask)</td>
</tr>
<tr>
<td>DPPS</td>
<td>_m128_mm_dp_ps(_m128 a, __m128 b, const int mask)</td>
</tr>
<tr>
<td>EMMS</td>
<td>_mm_empty()</td>
</tr>
<tr>
<td>EXTRACTPS</td>
<td>int_mm_extract_ps(_m128 src, const int ndx)</td>
</tr>
<tr>
<td>HADDPD</td>
<td>_m128d_mm_hadd_pd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>HADDPS</td>
<td>_m128_mm_hadd_ps(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>HSUBPD</td>
<td>_m128d_mm_hsub_pd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>HSUBPS</td>
<td>_m128_mm_hsub_ps(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>INSERTPS</td>
<td>_m128_mm_insert_ps(_m128 dst, __m128 src, const int ndx)</td>
</tr>
<tr>
<td>LDDQU</td>
<td>_m128i_mm_lddqu_si128(_m128i const *p)</td>
</tr>
<tr>
<td>LDMXCSR</td>
<td>_mm_setcsr(unsigned int i)</td>
</tr>
<tr>
<td>LFENCE</td>
<td>_mm lfence(void)</td>
</tr>
<tr>
<td>MASKMOVDQU</td>
<td>_mm_maskmoveu_si128(_m128i d, __m128i n, char *p)</td>
</tr>
<tr>
<td>MASKMOVQ</td>
<td>_mm_maskmove_si64(_m64 d, __m64 n, char *p)</td>
</tr>
<tr>
<td>MAXPD</td>
<td>_m128d_mm_max_pd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>MAXPS</td>
<td>_m128_mm_max_ps(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>MAXSD</td>
<td>_m128d_mm_max_sd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>MAXSS</td>
<td>_m128_mm_max_ss(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>MFENCE</td>
<td>_mm mfence(void)</td>
</tr>
<tr>
<td>MINPD</td>
<td>_m128d_mm_min_pd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>MINPS</td>
<td>_m128_mm_min_ps(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>MINSD</td>
<td>_m128d_mm_min_sd(_m128d a, __m128d b)</td>
</tr>
<tr>
<td>MINSS</td>
<td>_m128_mm_min_ss(_m128 a, __m128 b)</td>
</tr>
<tr>
<td>MONITOR</td>
<td>_mm_monitor(void const *p, unsigned extensions, unsigned hints)</td>
</tr>
<tr>
<td>MOVAPD</td>
<td>_m128d_mm_load_pd(double * p)</td>
</tr>
<tr>
<td>MOVAPS</td>
<td>_m128_mm_load_ps(float * p)</td>
</tr>
</tbody>
</table>
Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
</table>
| MOVD     | __m128i_mm_cvts32_si128(int a)  
          | int_mm_cvtsi128_si32(__m128i a)  
          | __m64_mm_cvts32_si64(int a)  
          | int_mm_cvtsi64_si32(__m64 a) |
| MOVDDUP  | __m128d_mm_movedup_pd(__m128d a)  
          | __m128d_mm_loaddup_pd(double const * dp) |
| MOVDQA   | __m128i_mm_loads128(__m128i a)  
          | void_mm_store_si128(__m128i * p, __m128i a) |
| MOVDFQ   | __m128i_mm_loadu128(__m128i a)  
          | void_mm_storeu128(__m128i * p, __m128i a) |
| MOVDFQ2Q | __m64_mm_movepi64_pi64(__m128i a) |
| MOVHLPS  | __m128_mm_movehls128(__m128 a, __m128 b) |
| MOVHPD   | __m128d_mm_loadhpd(__m128d a, double * p)  
          | void_mm_storehpd(double * p, __m128d a) |
| MOVHPS   | __m128_mm_loadhp(__m128 a, __m64 * p)  
          | void_mm_storehp(__m64 * p, __m128 a) |
| MOVLPD   | __m128d_mm_loadlp(__m128d a, double * p)  
          | void_mm_storelp(double * p, __m128d a) |
| MOVLPD   | __m128d_mm_loadlp(__m128d a, double * p)  
          | void_mm_storelp(double * p, __m128d a) |
| MOVNTDQA | __m128i_mm_stream_loads128(__m128i a)  
          | void_mm_store_si128(__m128i * p, __m128i a) |
| MOVNTDQ  | void_mm_stream_s128(__m128i a)  
          | void_mm_storehpd(double * p, __m128d a) |
| MOVNTPD  | void_mm_stream_s128(__m128i a)  
          | void_mm_storehpd(double * p, __m128d a) |
| MOVNTPS  | void_mm_stream_ps(float * p, __m128 a)  
          | void_mm_stream_s128(int * p, int a) |
| MOVNTQ   | void_mm_stream_s128(int * p, int a)  
          | void_mm_stream_s128(int * p, int a) |
| MOVQ     | __m128i_mm_loadl_epi64(__m128i * p)  
          | void_mm_storel_epi64(__m128i * p, __m128i a)  
          | __m128i_mm_move_epi64(__m128i a) |
| MOVQ2DQ  | __m128i_mm_movepi64_epi64(__m64 a) |
| MOVSD    | __m128d_mm_loadsd(double * p)  
          | void_mm_store_sd(double * p, __m128d a)  
          | __m128d_mm_move_sd(__m128d a, __m128d b) |
| MOVSDUP  | __m128_mm_movehdup_ps(__m128 a) |
| MOVSLDUP | __m128_mm_movedup_ps(__m128 a) |
| MOVSS    | __m128_mm_load_ss(float * p)  
          | void_mm_store_ss(float * p, __m128 a)  
          | __m128_mm_move_ss(__m128 a, __m128 b) |
Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
</table>
| MOVUPD   | __m128d_mm_loadu_pd(double * p)  
               void_mm_storeu_pd(double *p, __m128d a) |
| MOVUPS   | __m128_mm_loadu_ps(float * p)    
               void_mm_storeu_ps(float *p, __m128 a) |
| MPSADBW  | __m128i_mm_mpsadbw_epi8(__m128i s1, __m128i s2, const int mask) |
| MULPD    | __m128d_mm_mul_pd(__m128d a, __m128d b) |
| MULPS    | __m128_mm_mul_ss(__m128 a, __m128 b) |
| MULSD    | __m128d_mm_mul_sd(__m128d a, __m128d b) |
| MULSS    | __m128_mm_mul_ss(__m128 a, __m128 b) |
| MWAIT    | void_mm_mwait(unsigned extensions, unsigned hints) |
| ORPD     | __m128d_mm_or_pd(__m128d a, __m128d b) |
| ORPS     | __m128_mm_or_ps(__m128 a, __m128 b) |
| PABSB    | __m64_mm_abs_pi8 (__m64 a)  
               __m128i_mm_abs_epi8 (__m128i a) |
| PABSD    | __m64_mm_abs_pi32 (__m64 a)  
               __m128i_mm_abs_epi32 (__m128i a) |
| PABSW    | __m64_mm_abs_pl6 (__m64 a)  
               __m128i_mm_abs_epi16 (__m128i a) |
<p>| PACKSSWB | __m128i_mm_packs_epi16(__m128i m1, __m128i m2) |
| PACKSSWB | __m64_mm_packs_pi16(__m64 m1, __m64 m2) |
| PACKSSDW | __m128i_mm_packs_epi32 (__m128i m1, __m128i m2) |
| PACKSSDW | __m64_mm_packs_pi32 (__m64 m1, __m64 m2) |
| PACKUSDW | __m128i_mm_packus_epi32(__m128i m1, __m128i m2) |
| PACKUSDW | __m64_mm_packus_epi16(__m128i m1, __m128i m2) |
| PACKUSWB | __m64_mm_packs_pu16(__m64 m1, __m64 m2) |
| PADD     | __m128i_mm_add_epi8(__m128i m1, __m128i m2) |
| PADD     | __m64_mm_add_pi8(__m64 m1, __m64 m2) |
| PADDW    | __m128i_mm_add_epi16(__m128i m1, __m128i m2) |
| PADDW    | __m64_mm_add_pl6(__m64 m1, __m64 m2) |
| PADDQ    | __m128i_mm_add_epi64(__m128i m1, __m128i m2) |
| PADDQ    | __m64_mm_add_si64(__m64 m1, __m64 m2) |
| PADDSB   | __m128i_mm_adds_epi8(__m128i m1, __m128i m2) |
| PADDSB   | __m64_mm_adds_pi8(__m64 m1, __m64 m2) |
| PADDWB   | __m128i_mm_adds_epi16(__m128i m1, __m128i m2) |
| PADDWB   | __m64_mm_adds_pi16(__m64 m1, __m64 m2) |
| PADDUSB  | __m128i_mm_adds_epi8(__m128i m1, __m128i m2) |
| PADDUSB  | __m64_mm_adds_pi8(__m64 m1, __m64 m2) |
| PADDUSW  | __m128i_mm_adds_epi16(__m128i m1, __m128i m2) |
| PADDUSW  | __m64_mm_adds_pi16(__m64 m1, __m64 m2) |</p>
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PALIGNR</td>
<td>__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)</td>
</tr>
<tr>
<td></td>
<td>__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)</td>
</tr>
<tr>
<td>PAND</td>
<td>__m128i _mm_and_si128(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_and_si64(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PANDN</td>
<td>__m128i _mm_andnot_si128(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_andnot_si64(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PAUSE</td>
<td>void _mm_pause(void)</td>
</tr>
<tr>
<td>PAVGB</td>
<td>__m128i _mm_avg_epu8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_avg_epu8(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PAVGW</td>
<td>__m128i _mm_avg_epu16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_avg_epu16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PBLENDVB</td>
<td>__m128i _mm_blendv_epi (__m128i v1, __m128i v2, __m128i mask)</td>
</tr>
<tr>
<td>PBLENDW</td>
<td>__m128i _mm_blendv_epi16(__m128i v1, __m128i v2, const int mask)</td>
</tr>
<tr>
<td>PCLMULQDQ</td>
<td>__m128i _mm_clmulepi64_si128 (__m128i m1, __m128i m2, const int)</td>
</tr>
<tr>
<td>PCMPEQB</td>
<td>__m128i _mm_cmpeqb_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_cmpeqb_epi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPEQBQ</td>
<td>__m128i _mm_cmpeq_epi64(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPEQW</td>
<td>__m128i _mm_cmpeq_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPEQWQ</td>
<td>__m64 _mm_cmpeq_epi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPEQD</td>
<td>__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPEQDQ</td>
<td>__m64 _mm_cmpeq_epi32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPGTD</td>
<td>__m128i _mm_cmpgt_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPGTDQ</td>
<td>__m64 _mm_cmpgt_epi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPGTW</td>
<td>__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPGTWQ</td>
<td>__m64 _mm_cmpgt_epi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPGTMD</td>
<td>__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPGTMDQ</td>
<td>__m64 _mm_cmpgt_epi32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPISTRI</td>
<td>int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
</tbody>
</table>

Table C-1. Simple Intrinsics (Contd.)
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int _mm_cmpestrc</strong></td>
<td>(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpestro</strong></td>
<td>(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpestrs</strong></td>
<td>(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpistrz</strong></td>
<td>(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td><strong>PCMPISTRM</strong></td>
<td>__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpistra</strong></td>
<td>(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpistrc</strong></td>
<td>(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpistro</strong></td>
<td>(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpistrs</strong></td>
<td>(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td><strong>int _mm_cmpistrz</strong></td>
<td>(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td><strong>PCMPGTQ</strong></td>
<td>__m128i _mm_cmpgt_epi64(__m128i a, __m128i b)</td>
</tr>
<tr>
<td><strong>PEXTRB</strong></td>
<td>int _mm_extract_epi8 (__m128i src, const int ndx)</td>
</tr>
<tr>
<td><strong>PEXTRD</strong></td>
<td>int _mm_extract_epi32 (__m128i src, const int ndx)</td>
</tr>
<tr>
<td><strong>PEXTRQ</strong></td>
<td>__int64 _mm_extract_epi64 (__m128i src, const int ndx)</td>
</tr>
<tr>
<td><strong>PEXTRW</strong></td>
<td>int _mm_extract_epi16(__m128i a, int n)</td>
</tr>
<tr>
<td><strong>PEXTRW</strong></td>
<td>int _mm_extract_pi16(__m128i a, int n)</td>
</tr>
<tr>
<td><strong>__m128i _mm_hadd_pi32</strong></td>
<td>(__m64 a, __m64 b)</td>
</tr>
<tr>
<td><strong>PHADD</strong></td>
<td>__m128i _mm_hadd_epi32 (__m128i a, __m128i b)</td>
</tr>
<tr>
<td><strong>PHADDSW</strong></td>
<td>__m128i _mm_hadds_epi16 (__m128i a, __m128i b)</td>
</tr>
<tr>
<td><strong>PHADDD</strong></td>
<td>__m128i _mm_hadd_epi32 (__m128i a, __m128i b)</td>
</tr>
<tr>
<td><strong>__m64 _mm_max_epi8</strong></td>
<td>(__m128i a, __m128i b)</td>
</tr>
<tr>
<td><strong>PMAXSB</strong></td>
<td>__m128i _mm_max_epi8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td><strong>PMAXSD</strong></td>
<td>__m128i _mm_max_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PMAXSW</td>
<td>__m128i_mm_max_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXSW</td>
<td>__m64_mm_max_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMAXUB</td>
<td>__m128i_mm_max_epu8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXUB</td>
<td>__m64_mm_max_pu8(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMAXUD</td>
<td>__m128i_mm_max_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXUD</td>
<td>__m64_mm_max_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMINSB</td>
<td>__m128i_mm_min_epi8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINSB</td>
<td>__m128i_mm_min_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINSW</td>
<td>__m128i_mm_min_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINSW</td>
<td>__m64_mm_min_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMINUB</td>
<td>__m128i_mm_min_epi8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINUB</td>
<td>__m64_mm_min_pu8(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMINUD</td>
<td>__m128i_mm_min_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINUD</td>
<td>__m128i_mm_min_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMOVMSKB</td>
<td>int_mm_movemask_epi8(__m128i a)</td>
</tr>
<tr>
<td>PMOVMSKB</td>
<td>int_mm_movemask_pi8(__m64 a)</td>
</tr>
<tr>
<td>PMOVSBW</td>
<td>__m128i_mm_cvtepi8_epi16(__m128i a)</td>
</tr>
<tr>
<td>PMOVSBXBD</td>
<td>__m128i_mm_cvtepi8_epi32(__m128i a)</td>
</tr>
<tr>
<td>PMOVSBXQ</td>
<td>__m128i_mm_cvtepi8_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVSXWD</td>
<td>__m128i_mm_cvtepi16_epi32(__m128i a)</td>
</tr>
<tr>
<td>PMOVSXWQ</td>
<td>__m128i_mm_cvtepi16_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVSXDO</td>
<td>__m128i_mm_cvtepi32_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVBXW</td>
<td>__m128i_mm_cvtepu8_epi16(__m128i a)</td>
</tr>
<tr>
<td>PMOVBXQ</td>
<td>__m128i_mm_cvtepu8_epi32(__m128i a)</td>
</tr>
<tr>
<td>PMOVBXQ</td>
<td>__m128i_mm_cvtepu8_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVBXQ</td>
<td>__m128i_mm_cvtepu16_epi32(__m128i a)</td>
</tr>
<tr>
<td>PMOVBXQ</td>
<td>__m128i_mm_cvtepu16_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMULRHSW</td>
<td>__m64_mm_mullhs_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMULRHSW</td>
<td>__m128i_mm_mullhs_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMULHUW</td>
<td>__m128i_mm_mulhi_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMULHUW</td>
<td>__m64_mm_mulhi_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMULH</td>
<td>__m128i_mm_mulhi_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PMULH</td>
<td>__m64_mm_mulhi_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PMULLUD</td>
<td>__m128i_mm_mullo_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMULLW</td>
<td>__m128i_mm_mullo_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PMULLW</td>
<td>__m64_mm_mullo_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PMULDUQ</td>
<td>__m64_mm_mullsu32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PMULDUQ</td>
<td>__m128i_mm_mull_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>POPCNT</td>
<td>int _mm_popcnt_u32(unsigned int a)</td>
</tr>
<tr>
<td></td>
<td>int64_t _mm_popcnt_u64(unsigned __int64 a)</td>
</tr>
<tr>
<td>POR</td>
<td>_m64 _mm_or_si64(_m64 m1, _m64 m2)</td>
</tr>
<tr>
<td>POR</td>
<td>_m128i _mm_or_si128(_m128i m1, _m128i m2)</td>
</tr>
<tr>
<td>PREFETCHh</td>
<td>void _mm_prefetch(char *a, int sel)</td>
</tr>
<tr>
<td>PSADBW</td>
<td>_m128i _mm_sad_epu8(_m128i a, _m128i b)</td>
</tr>
<tr>
<td>PSADBW</td>
<td>_m64 _mm_sad_epu8(_m64 a, _m64 b)</td>
</tr>
<tr>
<td>PSHUFB</td>
<td>_m64 _mm_shuffle_epi8(_m64 a, _m64 b)</td>
</tr>
<tr>
<td></td>
<td>_m128i _mm_shuffle_epi8(_m128i a, _m128i b)</td>
</tr>
<tr>
<td>PSHUFD</td>
<td>_m128i _mm_shuffle_epi32(_m128i a, int n)</td>
</tr>
<tr>
<td>PSHUFHW</td>
<td>_m128i _mm_shuffle_epi64(_m128i a)</td>
</tr>
<tr>
<td>PSHUFW</td>
<td>_m64 _mm_shuffle_epi16(_m64 a, int n)</td>
</tr>
<tr>
<td>PSIGNB</td>
<td>_m64 _mm_sign_pi8(_m64 a, _m64 b)</td>
</tr>
<tr>
<td></td>
<td>_m128i _mm_sign_epi8(_m128i a, _m128i b)</td>
</tr>
<tr>
<td>PSIGND</td>
<td>_m64 _mm_sign_epi32(_m64 a, _m64 b)</td>
</tr>
<tr>
<td></td>
<td>_m128i _mm_sign_epi32(_m128i a, _m128i b)</td>
</tr>
<tr>
<td>PSIGNW</td>
<td>_m128i _mm_sign_epi64(_m128i a)</td>
</tr>
<tr>
<td>PSLLW</td>
<td>_m128i _mm_sll_epi16(_m128i m, _m128i count)</td>
</tr>
<tr>
<td>PSLLW</td>
<td>_m128i _mm_slli_epi16(_m128i m, int count)</td>
</tr>
<tr>
<td>PSLLW</td>
<td>_m64 _mm_sll_epi16(_m64 m, _m64 count)</td>
</tr>
<tr>
<td>PSLLW</td>
<td>_m64 _mm_slli_epi16(_m64 m, int count)</td>
</tr>
<tr>
<td>PSLLD</td>
<td>_m128i _mm_slli_epi32(_m128i m, int count)</td>
</tr>
<tr>
<td>PSLLD</td>
<td>_m128i _mm_slli_epi32(_m128i m, _m128i count)</td>
</tr>
<tr>
<td>PSLLQ</td>
<td>_m64 _mm_slli_epi64(_m64 m, int count)</td>
</tr>
<tr>
<td>PSLLQ</td>
<td>_m128i _mm_slli_epi64(_m128i m, _m128i count)</td>
</tr>
<tr>
<td>PSLLDQ</td>
<td>_m128i _mm_slli_epi128(_m128i m, int imm)</td>
</tr>
<tr>
<td>PSRAW</td>
<td>_m128i _mm_sra_epi16(_m128i m, _m128i count)</td>
</tr>
<tr>
<td>PSRAW</td>
<td>_m128i _mm_sra_epi16(_m128i m, int count)</td>
</tr>
<tr>
<td>PSRAD</td>
<td>_m128i _mm_sra_epi32(_m128i m, _m128i count)</td>
</tr>
<tr>
<td>PSRAD</td>
<td>_m128i _mm_sra_epi32(_m128i m, int count)</td>
</tr>
<tr>
<td>PSRLW</td>
<td>_m128i _mmSrL_epi16(_m128i m, _m128i count)</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m128i_mm_srl_epi16</td>
<td>__m128i_mm_srl_epi16(__m128i m, int count)</td>
</tr>
<tr>
<td>__m64_mm_srl_pi16</td>
<td>__m64 mm_srl_pi16(__m64 m, __m64 count)</td>
</tr>
<tr>
<td>__m64_mm_srl_pi16</td>
<td>__m64_mm_srl_pi16(__m64 m, int count)</td>
</tr>
<tr>
<td>PSRLD</td>
<td>__m128i_mm_srl_epi32(__m128i m, __m128i count)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_srl_epi32(__m128i m, __m128i count)</td>
</tr>
<tr>
<td>PSRLD</td>
<td>__m64_mm_srl_pi32(__m64 m, __m64 count)</td>
</tr>
<tr>
<td></td>
<td>__m64_mm_srl_pi32(__m64 m, int count)</td>
</tr>
<tr>
<td>PSRLQ</td>
<td>__m128i_mm_srl_epi64(__m128i m, __m128i count)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_srl_epi64(__m128i m, int count)</td>
</tr>
<tr>
<td>PSRLQ</td>
<td>__m64_mm_srl_si64(__m64 m, __m64 count)</td>
</tr>
<tr>
<td></td>
<td>__m64_mm_srl_si64(__m64 m, int count)</td>
</tr>
<tr>
<td>PSRLDQ</td>
<td>__m128i_mm_srl_si128(__m128i m, int imm)</td>
</tr>
<tr>
<td>PSUBB</td>
<td>__m128i_mm_sub_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBB</td>
<td>__m64_mm_sub_pi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBW</td>
<td>__m128i_mm_sub_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBW</td>
<td>__m64_mm_sub_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBD</td>
<td>__m128i_mm_sub_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBD</td>
<td>__m64_mm_sub_pi32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBQ</td>
<td>__m128i_mm_sub_epi64(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBQ</td>
<td>__m64_mm_sub_pi64(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBSB</td>
<td>__m128i_mm_subs_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBSB</td>
<td>__m64_mm_subs_pi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBSW</td>
<td>__m128i_mm_subs_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBSW</td>
<td>__m64_mm_subs_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBUSB</td>
<td>__m128i_mm_subs_epu8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBUSB</td>
<td>__m64_mm_subs_pu8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBUSW</td>
<td>__m128i_mm_subs_epu16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBUSW</td>
<td>__m64_mm_subs_pu16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PTEST</td>
<td>int _mm_testz_si128(__m128i s1, __m128i s2)</td>
</tr>
<tr>
<td></td>
<td>int _mm_testc_si128(__m128i s1, __m128i s2)</td>
</tr>
<tr>
<td></td>
<td>int _mm_testnzc_si128(__m128i s1, __m128i s2)</td>
</tr>
<tr>
<td>PUNPCKHbw</td>
<td>__m64_mm_unpackhi_pi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKHbw</td>
<td>__m128i_mm_unpackhi_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKHwd</td>
<td>__m64_mm_unpackhi_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKHwd</td>
<td>__m128i_mm_unpackhi_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKHDQ</td>
<td>__m64_mm_unpackhi_pi32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKHDQ</td>
<td>__m128i_mm_unpackhi_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKHQDQ</td>
<td>__m128i_mm_unpackhi_epi64(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLbw</td>
<td>__m64_mm_unpacklo_pi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKLbw</td>
<td>__m128i_mm_unpacklo_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLwd</td>
<td>__m64_mm_unpacklo_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>PUNPCKLWD</td>
<td>__m128i_mm_unpacklo_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>__m64_mm_unpacklo_pi32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>__m128i_mm_unpacklo_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>__m128i_mm_unpacklo_epi64(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PXOR</td>
<td>__m64_mm_xor_si64(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PXOR</td>
<td>__m128i_mm_xor_si128(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>RCPPS</td>
<td>__m128_mm_rcp_ps(__m128 a)</td>
</tr>
<tr>
<td>RCPSS</td>
<td>__m128_mm_rcp_ss(__m128 a)</td>
</tr>
<tr>
<td>ROUNDPD</td>
<td>__m128_mm_round_pd(__m128d s1, int iRoundMode)</td>
</tr>
<tr>
<td>ROUNDPS</td>
<td>__m128_mm_round_ps(__m128 s1, int iRoundMode)</td>
</tr>
<tr>
<td>ROUNDSD</td>
<td>__m128d_mm_round_sd(__m128d dst, __m128d s1, int iRoundMode)</td>
</tr>
<tr>
<td>ROUNDSS</td>
<td>__m128_mm_round_ss(__m128 dst, __m128 s1, int iRoundMode)</td>
</tr>
<tr>
<td>RSQRTPS</td>
<td>__m128_mm_rsqrt_ps(__m128 a)</td>
</tr>
<tr>
<td>RSQRTPS</td>
<td>__m128_mm_rsqrt_ps(__m128 a)</td>
</tr>
<tr>
<td>SFENCE</td>
<td>void_mm_sfence(void)</td>
</tr>
<tr>
<td>SHUFPD</td>
<td>__m128d_mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)</td>
</tr>
<tr>
<td>SHUFPS</td>
<td>__m128_mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)</td>
</tr>
<tr>
<td>SQRTPD</td>
<td>__m128d_mm_sqrt_pd(__m128d a)</td>
</tr>
<tr>
<td>SQRTPS</td>
<td>__m128_mm_sqrt_ps(__m128 a)</td>
</tr>
<tr>
<td>SQRSD</td>
<td>__m128d_mm_sqrt_sd(__m128d a)</td>
</tr>
<tr>
<td>SQRSS</td>
<td>__m128_mm_sqrt_ss(__m128 a)</td>
</tr>
<tr>
<td>STMXCSR</td>
<td>__mm_getcsr(void)</td>
</tr>
<tr>
<td>SUBPD</td>
<td>__m128d_mm_sub_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>SUBPS</td>
<td>__m128_mm_sub_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>SUBSD</td>
<td>__m128d_mm_sub_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>SUBSS</td>
<td>__m128_mm_sub_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>UCOMISD</td>
<td>int_mm_ucomieq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>UCOMIDT</td>
<td>int_mm_ucomileq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>UCOMIL</td>
<td>int_mm_ucomileq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>UCOMIGT</td>
<td>int_mm_ucomigte_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>UCOMINEQ</td>
<td>int_mm_ucomineq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>UCOMIEQ</td>
<td>int_mm_ucomieq_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>UCOMIMT</td>
<td>int_mm_ucomimt_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>UCOMIL</td>
<td>int_mm_ucomileq_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>UCOMIGT</td>
<td>int_mm_ucomigte_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>UCOMINEQ</td>
<td>int_mm_ucomineq_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>UCOMIEQ</td>
<td>int_mm_ucomieq_ss(__m128 a, __m128 b)</td>
</tr>
</tbody>
</table>
C.2 COMPOSITE INTRINSICS

Table C-2. Composite Intrinsics

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set_epi64(__m64 q1, __m64 q0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set_epi32(int i3, int i2, int i1, int i0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w1, short w0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9, char w8, char w7, char w6, char w5, char w4, char w3, char w2, char w1, char w0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set1_epi64(__m64 q)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set1_epi32(int a)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set1_epi16(short a)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_set1_epi8(char a)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_setr_epi64(__m64 q1, __m64 q0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_setr_epi32(int i3, int i2, int i1, int i0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_setr_epi16(short w7, short w6, short w5, short w4, short w3, short w2, short w1, short w0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_setr_epi8(char w15, char w14, char w13, char w12, char w11, char w10, char w9, char w8, char w7, char w6, char w5, char w4, char w3, char w2, char w1, char w0)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128i_mm_setzero_si128()</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128_mm_set_ps1(float w)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128_mm_set1_ps(float w)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128cmm_set1_pd(double w)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128d_mm_set_sd(double w)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128d_mm_set_pd(double z, double y)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128_mm_set_ps(float z, float y, float x, float w)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128d_mm_setr_pd(double z, double y)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128d_mm_setr_ps(float z, float y, float x, float w)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128d_mm_setzero_pd(void)</td>
</tr>
<tr>
<td>(composite)</td>
<td>_m128_mm_setzero_ps(void)</td>
</tr>
</tbody>
</table>
### Table C-2. Composite Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVSD + shuffle</td>
<td>__m128d_mm_load_pd(double * p)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_load1_pd(double *p)</td>
</tr>
<tr>
<td>MOVSS + shuffle</td>
<td>__m128_mm_load_ps1(float * p)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_load1_ps(float *p)</td>
</tr>
<tr>
<td>MOVAPD + shuffle</td>
<td>__m128d_mm_loadr_pd(double * p)</td>
</tr>
<tr>
<td>MOVAPS + shuffle</td>
<td>__m128_mm_loadr_ps(float * p)</td>
</tr>
<tr>
<td>MOVSD + shuffle</td>
<td>void__mm_store1_pd(double *p, __m128d a)</td>
</tr>
<tr>
<td>MOVSS + shuffle</td>
<td>void__mm_store1_ps(float *p, __m128 a)</td>
</tr>
<tr>
<td>MOVAPD + shuffle</td>
<td>__mm_storer_pd(double * p, __m128d a)</td>
</tr>
<tr>
<td>MOVAPS + shuffle</td>
<td>__mm_storer_ps(float * p, __m128 a)</td>
</tr>
</tbody>
</table>
INDEX

Numerics
0000 B-42
64-bit mode
  control and debug registers 2-12
default operand size 2-12
direct memory-offset MOVs 2-11
general purpose encodings B-18
immediates 2-11
introduction 2-7
machine instructions B-1
reg (reg) field B-4
REX prefixes 2-8, B-2
RIP-relative addressing 2-12
SMD encodings B-38
special instruction encodings B-65
summary table notation 3-8

A
AAA instruction 3-19, 3-21
AAD instruction 3-21
AAM instruction 3-23
AAS instruction 3-25
ADC instruction 3-27, 3-541
ADD instruction 3-19, 3-32, 3-282, 3-541
ADDPD instruction 3-34
ADDPSC- Add Packed Single-Precision Floating-Point Values 3-37
Addressing methods
  RIP-relative 2-12
Addressing, segments 1-6
ADDS- Add Scalar Double-Precision Floating-Point Values 3-40
ADDS instruction 3-40
ADDS- Add Scalar Single-Precision Floating-Point Values 3-42
ADDSUBPD instruction 3-44
ADDSUBPS instruction 3-46
AESDEC/AESDECLAST- Perform One Round of an AES Decryption
Flow 3-57
AESIMC- Perform the AES InvMixColumn Transformation 3-53
AESKEYGENASSIST - AES Round Key Generation Assist 3-60
AND instruction 3-62, 3-541
AND instruction 3-62, 3-541
ANDNPS- Bitwise Logical AND NOT of Packed Single Precision
Floating-Point Values 3-74
ANDPD- Bitwise Logical AND of Packed Double Precision Float-
ing-Point Values 3-65
ANDPD instruction 3-64
ANDPS- Bitwise Logical AND of Packed Single Precision Float-
ing-Point Values 3-68
Arctangent, x87 FPU operation 3-368
ARPL instruction 3-77
authenticated code execution mode 6-3

B
Base (operand addressing) 2-3
BCD integers
  packed 3-282, 3-284, 3-318, 3-320
  unpacked 3-19, 3-21, 3-23, 3-25
BEXTR - Bit Field Extract 3-81
Binary numbers 1-6
Bit order 1-4
BLSMSK - Get Mask Up to Lowest Set Bit 3-90
bootstrap processor 6-16, 6-21, 6-29, 6-30
BOUND instruction 3-107
BOUND range exceeded exception (#BR) 3-107
Branch hints 2-2
Brand information 3-219
  processor brand index 3-222
  processor brand string 3-219
BSF instruction 3-109
BSR instruction 3-111
BSWAP instruction 3-113
BT instruction 3-114
BTC instruction 3-116, 3-541
BTR instruction 3-118, 3-541
BTS instruction 3-120, 3-541
Byte order 1-4
C
C/C++ compiler intrinsics
  compiler functional equivalents C-1
  composite C-14
description of 3-13
  lists of C-1
  simple C-2
Cache and TLB information 3-213
Cache Inclusiveness 3-193
Caches, invalidating (flushing) 3-473, 5-552
CALL instruction 3-123
GETSEC 6-2
CBW instruction 3-136
CDQ instruction 3-281
CDQF instruction 3-136
CF (carry) flag, EFLAGS register 3-32, 3-114, 3-116, 3-118, 3-120,
  3-138, 3-149, 3-286, 3-447, 3-452, 4-144, 4-519, 4-586, 4-607,
  4-610, 4-651
CLC instruction 3-138
CLD instruction 3-139
CLEFLUSH instruction 3-140, 3-142
CPUID flag 3-212
CLI instruction 3-144
CLTS instruction 3-146
CMC instruction 3-149
CMOVcc flag 3-212
CMOVcc instructions 3-150
CPUID flag 3-212
CMP instruction 3-154
CMPPD- Compare Packed Double-Precision Floating-Point Values
3-156
CMPPS- Compare Packed Single-Precision Floating-Point Values
3-163
CMPS instruction 3-170, 4-546
CMPSB instruction 3-170
CMPSD- Compare Scalar Double-Precision Floating-Point Values
3-174
CMPSD instruction 3-170
CMPSQ instruction 3-170
CMPSQ instruction 3-170
CMSS- Compare Scalar Single-Precision Floating-Point Values
3-178
CMPSW instruction 3-170
CMPXCHG instruction 3-182, 3-541
CMXPCHG16B instruction 3-184
CPUID bit 3-210
CMXPCHG8B instruction 3-184
INDEX

CPUID flag 3-212
COMISD- Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS 3-187
COMISS- Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS 3-189
Compatibility mode
  introduction 2-7
  see 64-bit mode
  summary table notation 3-10
Compatibility, software 1-5
Condition code flags, EFLAGS register 3-150
Condition code flags, x87 FPU status word
  flags affected by instructions 3-15
  setting 3-404, 3-406, 3-409
Conditional jump 3-487
Conforming code segment 3-519
Constants (floating point), loading 3-358
Control registers, moving values to and from 4-40
Cosine, x87 FPU operation 3-334, 3-386
CPL 3-144, 5-92
CPUID instruction 3-191, 3-212
  36-bit page size extension 3-212
  APIC on-chip 3-212
  basic CPUID information 3-192
  cache and TLB characteristics 3-192
  CLFLUSH flag 3-212
  CLFLUSH instruction cache line size 3-208
  CMPXCHG16B flag 3-210
  CMPXCHG8B flag 3-212
  CPL qualified debug store 3-209
  debug extensions, CR4.DE 3-212
  debug store supported 3-213
  deterministic cache parameters leaf 3-192, 3-195, 3-196, 3-197, 3-198, 3-199, 3-200, 3-201
  extended function information 3-204, 3-205
  feature information 3-211
  FPU on-chip 3-212
  FSAVE flag 3-213
  FXRSTOR flag 3-213
  IA-32e mode available 3-205
  input limits for EAX 3-206
  L1 Context ID 3-210
  local APIC physical ID 3-208
  machine check architecture 3-212
  machine check exception 3-212
  memory type range registers 3-212
  MONITOR feature information 3-217
  MONITOR/MWAIT flag 3-209
  MONITOR/MWAIT leaf 3-193, 3-194, 3-196, 3-202
  MWAIT feature information 3-217
  page attribute table 3-212
  page size extension 3-212
  performance monitoring features 3-218
  physical address bits 3-206
  physical address extension 3-212
  power management 3-217, 3-218, 3-219
  processor brand index 3-208, 3-219
  processor brand string 3-205, 3-219
  processor serial number 3-192, 3-212
  processor type field 3-207
  RDMSR flag 3-212
  returned in EBX 3-208
  returned in ECX & EDX 3-208
  self snoop 3-213
  SpeedStep technology 3-209
  SS2 extensions flag 3-213
  SSE extensions flag 3-213
  SSE3 extensions flag 3-209
  SSSE3 extensions flag 3-209
  SYSENTER flag 3-212
  SYSEXIT flag 3-212
  thermal management 3-217, 3-218, 3-219
  thermal monitor 3-209, 3-213
  time stamp counter 3-212
  using CPUID 3-191
  vendor ID string 3-206
  version information 3-192, 3-217
  virtual 8086 Mode flag 3-212
  virtual address bits 3-206
  WRMSR flag 3-212
CQO instruction 3-281
CR0 control register 4-626
CS register 3-124, 3-460, 3-480, 3-493, 4-36, 4-385
CVTQ2PD- Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values 3-231, 5-24, 5-30, 5-49, 5-51, 5-56, 5-61, 5-76, 5-78
CVTQ2PD instruction 3-228
CVTQ2PS- Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values 3-235
CVTPD2DQ- Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 3-238
CVTPD2PI instruction 3-242
CVTPD2PS- Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values 3-243
CVTP2PD instruction 3-247
CVTP2PS instruction 3-248
CVTSD2Q- Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values 3-273
CVTSS2SI- Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer 3-206
CVTSS2SD- Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value 5-24, 5-51, 5-56, 5-61, 5-78
CVTSS2SS- Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value 3-262
CVTTPD2PD- Convert Scalar Double Precision Floating-Point Value to Doubleword Integer 3-256
CVTTPD2Q- Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 3-268
CVTTP2PI instruction 3-272
CVTTP2Q- Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values 3-273
CVTTP2PI instruction 3-276
CVTTSI2SI- Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer 3-277
CVTTS2SI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer 3-279
CWD instruction 3-281
CWDE instruction 3-136
D
D (default operation size) flag, segment descriptor 4-389
DAA instruction 3-282
DAS instruction 3-284
Debug registers, moving value to and from 4-43
DEC instruction 3-286, 3-541
Denormalized finite number 3-409
Detecting and Enabling SMX
  level 2 6-1
DF (direction) flag, EFLAGS register 3-139, 3-171, 3-454, 3-543, 4-107, 4-176, 4-588, 4-640
Displacement (operand addressing) 2-3
DIV instruction 3-288
Divide error exception (#DE) 3-288
DIVPD- Divide Packed Double-Precision Floating-Point Values 3-291, 5-296
DIVPS- Divide Packed Single-Precision Floating-Point Values 3-294
DIVSD- Divide Scalar Double-Precision Floating-Point Values 3-297
DIVSS- Divide Scalar Single-Precision Floating-Point Values 3-299
DS register 3-170, 3-525, 3-543, 4-107, 4-176
E
EDI register 4-588, 4-640, 4-644
Effective address 3-529
EFLAGS register
  condition codes 3-152, 3-326, 3-331
  flags affected by instructions 3-15
  popping 4-393
  popping on return from interrupt 3-480
  pushing 4-512
  pushing on interrupts 3-460
  saving 4-576
  status flags 3-154, 3-490, 4-593, 4-675
EIP register 3-124, 3-460, 3-480, 3-493
EMMS instruction 3-306
Encodings
  See machine instructions, opcodes
ENTER instruction 3-307
GETSEC 6-3, 6-10
ES register 3-525, 4-176, 4-588, 4-644
ESI register 3-170, 3-543, 4-107, 4-176, 4-640
ESP register 3-124
EVEX.R 3-5
Exceptions
  BOUND range exceeded (#BR) 3-107
  notation 1-6
  overflow exception (#OF) 3-460
  returning from 3-480
GETSEC 6-3, 6-5
Exponent, extracting from floating-point number 3-424
Extract exponent and significand, x87 FPU operation 3-424
EXTRACTPS- Extract packed floating-point values 3-310
F
F2XM1 instruction 3-312, 3-424
FABS instruction 3-314
FADD instruction 3-315
FADDP instruction 3-315
Far pointer, loading 3-525
Far return, RET instruction 4-549
FBLD instruction 3-318
FBSTP instruction 3-320
FCHS instruction 3-322
FCLEX instruction 3-324
FCMOVcc instructions 3-326
FCOM instruction 3-328
FCOMI instruction 3-331
FCOMIP instruction 3-331
FCOMP instruction 3-328
FCOMPP instruction 3-328
FCOS instruction 3-334
FDECTSP instruction 3-336
FDIV instruction 3-337
FDIVP instruction 3-337
FDIVR instruction 3-340
FDIVRP instruction 3-340
Feature information, processor 3-191
FFREE instruction 3-343
FIADD instruction 3-315
FICOM instruction 3-344
FICOMP instruction 3-344
FIDIV instruction 3-337
FIDIVR instruction 3-340
FILD instruction 3-346
FIMUL instruction 3-364
FINCSTP instruction 3-348
FINIT instruction 3-349
FINIT/FNINIT instructions 3-379
FIST instruction 3-351
FISTP instruction 3-351
FISTTP instruction 3-354
FISUB instruction 3-398
FISUBR instruction 3-401
FLD instruction 3-356
FLD1 instruction 3-358
FLDCW instruction 3-360
FLDENV instruction 3-362
FLDL2E instruction 3-358
FLDL2T instruction 3-358
FLDLG2 instruction 3-358
FLDLN2 instruction 3-358
FLDPI instruction 3-358
FLDZ instruction 3-358
Floating point instructions
  machine encodings B-65
Floating-point exceptions
  SSE and SSE2 SIMD 3-17
  x87 FPU 3-17
Flushing
  caches 3-473, 5-552
  TLB entry 3-475
FMUL instruction 3-364
INDEX

FMULP instruction 3-364
FNCLX instruction 3-324
FNINIT instruction 3-349
FNOP instruction 3-367
FNSAVE instruction 3-379
FNSTCW instruction 3-392
FNSTENV instruction 3-362, 3-394
FNSTSW instruction 3-396
FPATAN instruction 3-368
FPREM instruction 3-370
FPREM1 instruction 3-372
FPTAN instruction 3-374
FRNDINT instruction 3-376
FRSTOR instruction 3-377
FS register 3-525
FSAVE instruction 3-379
FSAVE/FNSAVE instructions 3-377
FSCALE instruction 3-382
FSIN instruction 3-384
FSINCOS instruction 3-386
FSQRT instruction 3-388
FST instruction 3-390
FSTCW instruction 3-392
FNSTENV instruction 3-394
FSTP instruction 3-390
FNSTSW instruction 3-396
FSUB instruction 3-398
FSUBP instruction 3-398
FSUBR instruction 3-401
FSUBRP instruction 3-401
FTST instruction 3-404
FUCOM instruction 3-406
FUCOMI instruction 3-331
FUCOMIP instruction 3-331
FUCOMP instruction 3-406
FUCOMPP instruction 3-406
FXAM instruction 3-409
FXCH instruction 3-411
FXRSTOR instruction 3-413
CPUID flag 3-213
FXSAVE instruction 3-416, 5-549, 5-550, 5-574, 5-586, 5-591, 5-595, 5-598, 5-601, 5-604, 5-608
CPUID flag 3-213
FXTRACT instruction 3-382, 3-424
FYL2X instruction 3-426
FYL2XP1 instruction 3-428

G
GDT (global descriptor table) 3-534, 3-537
GDTR (global descriptor table register) 3-534, 4-596

General-purpose instructions
64-bit encodings B-18
non-64-bit encodings B-7

General-purpose registers
moving value to and from 4-36
popping all 4-389
pushing all 4-510
GETSEC 6-1, 6-2, 6-5
GS register 3-525

H
HADDPD instruction 3-430, 3-431
HADDPDS instruction 3-433
Hexadecimal numbers 1-6
HLT instruction 3-436
HSUBPD instruction 3-437
HSUBPS instruction 3-440

I
IA-32e mode
CPUID flag 3-205
introduction 2-7, 2-13, 2-35
see 64-bit mode
see compatibility mode
IDIV instruction 3-443
IDT (interrupt descriptor table) 3-461, 3-534
IDTR (interrupt descriptor table register) 3-534, 4-622
IF (interrupt enable) flag, EFLAGS register 3-144, 4-641
Immediate operands 2-3
IMUL instruction 3-446
IN instruction 3-450
INC instruction 3-452, 3-541
Index (operand addressing) 2-3
Initialization x87 FPU 3-349
initiating logical processor 6-3, 6-4, 6-5, 6-10, 6-21
INS instruction 3-454, 4-546
INSB instruction 3-454
INSD instruction 3-454
INSERTPS- Insert Scalar Single-Precision Floating-Point Value 3-457
instruction encodings B-61, B-67, B-74

Instruction format
base field 2-3
description of reference information 3-1
displacement 2-3
immediate 2-3
index field 2-3
Mod field 2-3
ModR/M byte 2-3
opcode 2-3
operands 1-5
prefixes 2-1
r/m field 2-3
reg/opcode field 2-3
scale field 2-3
SIB byte 2-3
See also: machine instructions, opcodes

Instruction reference, nomenclature 3-1
Instruction set, reference 3-1
INSW instruction 3-454
INT 3 instruction 3-460
Integer, storing, x87 FPU data type 3-351
Intel 64 architecture
definition of 1-4
instruction format 2-1
relation to IA-32 1-4
Intel NetBurst microarchitecture 1-3
Intel software network link 1-8
Intel VTune Performance Analyzer
related information 1-7
INDEX

Intel Xeon processor 1-1
Intel® Trusted Execution Technology 6-3
Inter-privilege level
call, CALL instruction 3-123
return, RET instruction 4-549
Interrupts
returning from 3-480
software 3-460
INTn instruction 3-460
INTO instruction 3-460
Intrinsics
compiler functional equivalents C-1
description of 3-13
list of C-1
simple C-2
INVD instruction 3-473
INVLPG instruction 3-475
IOPL (I/O privilege level) field, EFLAGS register 3-144
IRET instruction 3-480
IRETD instruction 3-480
J
Jcc instructions 3-487
JMP instruction 3-492
Jump operation 3-492
L
L1 Context ID 3-210
LAHF instruction 3-518
LAR instruction 3-519
Last branch
interrupt & exception recording
description of 4-561
LDDQU instruction 3-522
LDMXCSR instruction 3-524, 4-526, 5-554
LDS instruction 3-525
LDT (local descriptor table) 3-537
LDTR (local descriptor table register) 3-537, 4-624
LEA instruction 3-529
LEAVE instruction 3-531
LES instruction 3-525
LFENCE instruction 3-533
LFS instruction 3-525
LGDT instruction 3-534
LGS instruction 3-525
LIDT instruction 3-534
LLDT instruction 3-537
LMSW instruction 3-539
Load effective address operation 3-529
LOCK prefix 3-28, 3-33, 3-62, 3-116, 3-118, 3-120, 3-182, 3-286,
3-452, 3-541, 4-161, 4-164, 4-166, 4-586, 4-651, 5-565, 5-570,
5-578
Locking operation 3-541
LODS instruction 3-543, 4-546
LODSB instruction 3-543
LODSD instruction 3-543
LODSQ instruction 3-543
LODSW instruction 3-543
Log (base 2), x87 FPU operation 3-428
Log epsilon, x87 FPU operation 3-426
LOOP instructions 3-546
LOOPcc instructions 3-546
LSL instruction 3-548
LSS instruction 3-525
LTR instruction 3-551
LZCNT - Count the Number of Leading Zero Bits 3-553
M
Machine check architecture
CPUID flag 3-212
description 3-212
Machine instructions
64-bit mode B-1
condition test (tttn) field B-6
direction bit (d) field B-6
floating-point instruction encodings B-65
general description B-1
general-purpose encodings B-7–B-38
legacy prefixes B-1
MMX encodings B-39–B-42
opcode fields B-2
operand size (w) bit B-4
P6 family encodings B-42
Pentium processor family encodings B-38
reg (reg) field B-3, B-4
REX prefixes B-2
segment register (sreg) field B-5
sign-extend (s) bit B-5
SIMD 64-bit encodings B-38
special 64-bit encodings B-65
special fields B-2
special-purpose register (eee) field B-5
SSE encodings B-43–B-49
SSE2 encodings B-49–B-59
SSE3 encodings B-60–B-61
SSSE3 encodings B-61–B-64
VMX encodings B-118, B-119
See also: opcodes
Machine status word, CR0 register 3-539, 4-626
MASKMOVQ instruction 4-43
MASKMOVQ instruction 5-291
MAXPD- Maximum of Packed Double-Precision Floating-Point Values
4-12
MAXPS- Maximum of Packed Single-Precision Floating-Point Values
4-15
MAXSD- Return Maximum Scalar Double-Precision Floating-Point Value
4-18
MAXSS- Return Maximum Scalar Single-Precision Floating-Point Value
4-20
measured environment 6-1
Measured Launch Environment 6-1, 6-25
MFENCE instruction 4-22
MINPD- Minimum of Packed Double-Precision Floating-Point Values
4-23
MINPS- Minimum of Packed Single-Precision Floating-Point Values
4-26
MINSD- Return Minimum Scalar Double-Precision Floating-Point Value
4-29
MINSS- Return Minimum Scalar Single-Precision Floating-Point Value
4-31

Vol. 2D INDEX-5
INDEX

MLE 6-1
MMX instructions
  CPUID flag for technology 3-213
  encodings B-39
Mod field, instruction format 2-3
Model & family information 3-217
ModR/M byte 2-3
  16-bit addressing forms 2-5
  32-bit addressing forms of 2-6
  description of 2-3
MONITOR instruction 4-33
  CPUID flag 3-209
  feature data 3-217
MOV instruction 4-35
MOV instruction (control registers) 4-40
MOV instruction (debug registers) 4-43, 4-53
MOVAPD- Move Aligned Packed Double-Precision Floating-Point Values 4-45
MOVAPS- Move Aligned Packed Single-Precision Floating-Point Values 4-49
MOV instruction 4-53
MOVDDUP- Replicate Double FP Values 4-59
MOVQ2Q instruction 4-75
MOVQQA- Move Aligned Packed Integer Values 4-62
MOVQQU- Move Unaligned Packed Integer Values 4-67
MOVLHPS - Move Packed Single-Precision Floating-Point Values
  High to Low 4-76
MOVHPD- Move High Packed Double-Precision Floating-Point Values 4-78
MOVHPS- Move High Packed Single-Precision Floating-Point Values 4-80
MOVLPD- Move Low Packed Double-Precision Floating-Point Values 4-84
MOVLPS- Move Low Packed Single-Precision Floating-Point Values 4-86
MOVMSKPD instruction 4-88
MOVMSKPS instruction 4-90
MOVNTDQ instruction 4-106
MOVNTDQ- Store Packed Integers Using Non-Temporal Hint 4-94
MOVNTI instruction 4-106
MOVNTPD- Store Packed Double-Precision Floating-Point Values
  Using Non-Temporal Hint 4-98
MOVNTPS- Store Packed Single-Precision Floating-Point Values
  Using Non-Temporal Hint 4-100
MOVQ instruction 4-102
MOVQ instruction 4-53, 4-103
MOVQ2DQ instruction 4-106
MOVQ instruction 4-107, 4-546
MOVSB instruction 4-107
MOVSD instruction 4-107
MOVSD- Move or Merge Scalar Double-Precision Floating-Point Value 4-111
MOVSHDUP- Replicate Single FP Values 4-114
MOVSLDUP- Replicate Single FP Values 4-117
MOVQ instruction 4-107
MOVSS- Move or Merge Scalar Single-Precision Floating-Point Value
  4-120
MOVSW instruction 4-107
MOVX instruction 4-124
MOVSXD instruction 4-124
MOVUPS- Move Unaligned Packed Single-Precision Floating-Point Values 4-126
MOVUPD- Move Unaligned Packed Double-Precision Floating-Point Values 4-126
MVZX instruction 4-134
MSRs (model specific registers)
  reading 4-528
MUL instruction 3-23, 4-144
MULPD- Multiply Packed Double-Precision Floating-Point Values 4-146
MULPS- Multiply Packed Single-Precision Floating-Point Values 4-149
MULSD- Multiply Scalar Double-Precision Floating-Point Values 4-152
MULSS- Multiply Scalar Single-Precision Floating-Point Values 4-154
Multi-byte no operation 4-161, 4-163, B-13
MULX - Unsigned Multiply Without Affecting Flags 4-156
MVMM 6-1, 6-4, 6-5, 6-37
MWAIT instruction 4-158
  CPUID flag 3-209
  feature data 3-217
N
NaN. testing for 3-404
Near
  return, RET instruction 4-549
NEG instruction 3-541, 4-161
NetBurst microarchitecture (see Intel NetBurst microarchitecture)
No operation 4-161, 4-163, B-12
Nomenclature, used in instruction reference pages 3-1
NOP instruction 4-163
NOT instruction 3-541, 4-164
Notation
  bit and byte order 1-4
  exceptions 1-6
  hexadecimal and binary numbers 1-6
  instruction operands 1-5
  reserved bits 1-5
  segmented addressing 1-6
Notational conventions 1-4
NT (nested task) flag, EFLAGS register 3-480
O
OF (carry) flag, EFLAGS register 3-447
OF (overflow) flag, EFLAGS register 3-32, 3-460, 4-144, 4-586,
  4-607, 4-610, 4-651
Opcode format 2-3
Opcodes
  addressing method codes for A-1
  extensions A-17
  extensions tables A-18
  group numbers A-17
  integers
    one-byte opcodes A-7
    two-byte opcodes A-7
  key to abbreviations A-1
  look-up examples A-3, A-17, A-20
  ModR/M byte A-17
  one-byte opcodes A-3, A-7
opcode maps A-1
operand type codes for A-2
register codes for A-3
superscripts in tables A-6
two-byte opcodes A-4, A-5, A-7
VMX instructions B-118, B-119
x87 ESC instruction opcodes A-20

Operands 1-5
OR instruction 3-541, 4-166
ORPS- Bitwise Logical OR of Packed Single Precision Floating-Point Values 4-171
OUT instruction 4-174
OUTS instruction 4-176, 4-546
OUTSB instruction 4-176
OUTSD instruction 4-176
OUTSW instruction 4-176
Overflow exception (#OF) 3-460

P
P6 family processors
description of 1-1
machine encodings B-42
PABSB instruction 4-180, 4-194, 5-84, 5-405, 5-416, 5-431
PABSD instruction 4-180, 4-194, 5-84, 5-405, 5-416, 5-431
PACKSSDw instruction 4-186
PACKSSWB instruction 4-186
PACKUSWB instruction 4-199
PADD/PADDW/PADDD/PADDQ - Add Packed Integers 4-204
PADDQ instruction 4-211
PADDW instruction 4-211
PADDUSB instruction 4-215
PADDUSW instruction 4-215
PALIGNR instruction 4-219
PAND instruction 4-223
PANDN instruction 4-226
GETSEC 6-4
PAUSE instruction 4-229
PAVGB instruction 4-230
PAVGW instruction 4-230
PCE flag, CR4 register 4-534
PCLMULQDQ - Carry-Less Multiplication Quadword 5-315, 5-324
PCMPQ instruction 4-244
PCMPQD instruction 4-244
PCMPQW instruction 4-244
PCMPGTB instruction 4-257
PCMPGTQ instruction 4-257
PCMPGTW instruction 4-257
PDEP - Parallel Bits Deposit 4-270
PE (protection enable) flag, CR0 register 3-539
Pending break enable 3-213
Pentium 4 processor 1-1
Pentium II processor 1-2
Pentium III processor 1-2
Pentium Pro processor 1-2
Pentium processor 1-1
Pentium processor family processors
machine encodings B-38
Performance-monitoring counters
CPUID inquiry for 3-218

PEXTRW instruction 4-277
PHADDD instruction 4-280
PHADD instruction 4-284
PHADDW instruction 4-280
PHSUBD instruction 4-288
PHSUBSW instruction 4-291
PHSUBW instruction 4-288
Pi 3-358
PMADDUBW instruction 4-298
PMADDUDW instruction 4-298
PMULHRSW instruction 4-361
PMULHUW instruction 4-365
PMULHW instruction 4-369
PMULLW instruction 4-377
PMULUDQ instruction 4-381
POP instruction 4-384
POP instruction 4-389
POP instruction 4-393
POPFD instruction 4-393
POP instruction 4-393
POR instruction 4-397
PREFETCHh instruction 4-400
PREFETCHwT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint 4-404
Prefixes
Address-size override prefix 2-2
Branch hints 2-2
branch hints 2-2
instruction, description of 2-1
legacy prefix encodings B-1
LOCK 2-1, 3-541
Operand-size override prefix 2-2
REP or REPE/REPZ 2-1
REP/REPE/REPZ/REPNZ 4-545
REPNZ 2-1
REX prefix encodings B-2
Segment override prefixes 2-2
PSADBW instruction 4-404
PSHUFB instruction 4-408
PSHUFBD instruction 4-412
PSHUFHW instruction 4-416
PSHUFHW instruction 4-419
PSHUFHW instruction 4-422
PSIGNB instruction 4-423
PSIGND instruction 4-423
PSIGNV instruction 4-423
PSLLD instruction 4-429
PSLLDQ instruction 4-427
PSLLQ instruction 4-429
PSLLQ instruction 4-429
PSRAW instruction 4-441
PSRLD instruction 4-453
PSRLQ instruction 4-451
PSRLQ instruction 4-453
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSRLw instruction</td>
<td>4-453</td>
</tr>
<tr>
<td>PSUBBB instruction</td>
<td>4-465</td>
</tr>
<tr>
<td>PSUBD instruction</td>
<td>4-465</td>
</tr>
<tr>
<td>PSUBQ instruction</td>
<td>4-472</td>
</tr>
<tr>
<td>PSUBSB instruction</td>
<td>4-475</td>
</tr>
<tr>
<td>PSUBSW instruction</td>
<td>4-479</td>
</tr>
<tr>
<td>PSUBUSB instruction</td>
<td>4-479</td>
</tr>
<tr>
<td>PSUBUSW instruction</td>
<td>4-479</td>
</tr>
<tr>
<td>PSUBW instruction</td>
<td>4-465</td>
</tr>
<tr>
<td>PTEST - Packed Bit Test</td>
<td>3-513</td>
</tr>
<tr>
<td>PUNPCKHBW instruction</td>
<td>4-487</td>
</tr>
<tr>
<td>PUNPCKHDQ instruction</td>
<td>4-487</td>
</tr>
<tr>
<td>PUNPCKHQDQ instruction</td>
<td>4-487</td>
</tr>
<tr>
<td>PUNPCKHWD instruction</td>
<td>4-487</td>
</tr>
<tr>
<td>PUNPCKLBW instruction</td>
<td>4-497</td>
</tr>
<tr>
<td>PUNPCKLDQ instruction</td>
<td>4-497</td>
</tr>
<tr>
<td>PUNPCKLQDQ instruction</td>
<td>4-497</td>
</tr>
<tr>
<td>PUNPCKLWD instruction</td>
<td>4-497</td>
</tr>
<tr>
<td>PUSH instruction</td>
<td>4-507</td>
</tr>
<tr>
<td>PUSHA instruction</td>
<td>4-510</td>
</tr>
<tr>
<td>PUSHAD instruction</td>
<td>4-512</td>
</tr>
<tr>
<td>PUSHFD instruction</td>
<td>4-512</td>
</tr>
<tr>
<td>LXOR instruction</td>
<td>4-514</td>
</tr>
<tr>
<td>R/m field, instruction format</td>
<td>2-3</td>
</tr>
<tr>
<td>RC (rounding control) field</td>
<td>3-351, 3-358, 3-390</td>
</tr>
<tr>
<td>RCL instruction</td>
<td>4-517</td>
</tr>
<tr>
<td>RCPPS instruction</td>
<td>4-522</td>
</tr>
<tr>
<td>RCPS instruction</td>
<td>4-524</td>
</tr>
<tr>
<td>RCR instruction</td>
<td>4-517</td>
</tr>
<tr>
<td>RDMSR instruction</td>
<td>4-528, 4-534, 4-541</td>
</tr>
<tr>
<td>CPUID flag</td>
<td>3-212</td>
</tr>
<tr>
<td>RDPMC instruction</td>
<td>4-531, 4-533, 5-558</td>
</tr>
<tr>
<td>RDTSC instruction</td>
<td>4-537, 4-541, 4-543</td>
</tr>
<tr>
<td>Reg(opcode) field, instruction format</td>
<td>2-3</td>
</tr>
<tr>
<td>Related literature</td>
<td>1-7</td>
</tr>
<tr>
<td>Remainder, x87 FPU operation</td>
<td>3-372</td>
</tr>
<tr>
<td>REP/REPE/REPZ/REPNE/REPNZ prefixes</td>
<td>3-171, 3-455, 4-177, 4-545</td>
</tr>
<tr>
<td>Reserved</td>
<td>3-390</td>
</tr>
<tr>
<td>use of reserved bits</td>
<td>1-5</td>
</tr>
<tr>
<td>Responding logical processor</td>
<td>6-4</td>
</tr>
<tr>
<td>responding logical processor</td>
<td>6-4, 6-5</td>
</tr>
<tr>
<td>RET instruction</td>
<td>4-549</td>
</tr>
<tr>
<td>REX prefixes</td>
<td></td>
</tr>
<tr>
<td>addressing modes</td>
<td>2-9</td>
</tr>
<tr>
<td>and INC/DEC</td>
<td>2-8</td>
</tr>
<tr>
<td>encodings</td>
<td>2-8, B-2</td>
</tr>
<tr>
<td>field names</td>
<td>2-9</td>
</tr>
<tr>
<td>Mod/R/M byte</td>
<td>2-8</td>
</tr>
<tr>
<td>overview</td>
<td>2-8</td>
</tr>
<tr>
<td>REX.B</td>
<td>2-8</td>
</tr>
<tr>
<td>REX.R</td>
<td>2-8</td>
</tr>
<tr>
<td>REX.W</td>
<td>2-8</td>
</tr>
<tr>
<td>special encodings</td>
<td>2-11</td>
</tr>
<tr>
<td>RIP-relative addressing</td>
<td>2-12</td>
</tr>
<tr>
<td>ROL instruction</td>
<td>4-517</td>
</tr>
<tr>
<td>ROR instruction</td>
<td>4-517</td>
</tr>
<tr>
<td>RORX - Rotate Right Logical Without Affecting Flags</td>
<td>4-559</td>
</tr>
<tr>
<td>Rounding</td>
<td></td>
</tr>
<tr>
<td>modes, floating-point operations</td>
<td>4-561</td>
</tr>
<tr>
<td>Rounding control (RC) field</td>
<td></td>
</tr>
<tr>
<td>MXCSR register</td>
<td>4-561</td>
</tr>
<tr>
<td>x87 FPU control word</td>
<td>4-561</td>
</tr>
<tr>
<td>Rounding, round to integer, x87 FPU operation</td>
<td>3-376</td>
</tr>
<tr>
<td>ROUNDPD - Round Packed Double-Precision Floating-Point Values</td>
<td>4-652</td>
</tr>
<tr>
<td>RPL field</td>
<td>3-77</td>
</tr>
<tr>
<td>RSM instruction</td>
<td>4-570</td>
</tr>
<tr>
<td>RSQRTPS instruction</td>
<td>4-572</td>
</tr>
<tr>
<td>RSQRTSS instruction</td>
<td>4-574</td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Safer Mode Extensions</td>
<td>6-1</td>
</tr>
<tr>
<td>SAHF instruction</td>
<td>4-576</td>
</tr>
<tr>
<td>SAL instruction</td>
<td>4-578</td>
</tr>
<tr>
<td>SAR instruction</td>
<td>4-578</td>
</tr>
<tr>
<td>SBB instruction</td>
<td>3-541, 4-585</td>
</tr>
<tr>
<td>Scale (operand addressing)</td>
<td>2-3</td>
</tr>
<tr>
<td>Scale, x87 FPU operation</td>
<td>3-382</td>
</tr>
<tr>
<td>Scan string instructions</td>
<td>4-588</td>
</tr>
<tr>
<td>SCAS instruction</td>
<td>4-546, 4-588</td>
</tr>
<tr>
<td>SCASB instruction</td>
<td>4-588</td>
</tr>
<tr>
<td>SCASD instruction</td>
<td>4-588</td>
</tr>
<tr>
<td>SCASW instruction</td>
<td>4-588</td>
</tr>
<tr>
<td>Segment</td>
<td></td>
</tr>
<tr>
<td>descriptor, segment limit</td>
<td>3-548</td>
</tr>
<tr>
<td>limit</td>
<td>3-548</td>
</tr>
<tr>
<td>registers, moving values to and from</td>
<td>4-36</td>
</tr>
<tr>
<td>selector, RPL field</td>
<td>3-77</td>
</tr>
<tr>
<td>Segmented addressing</td>
<td>1-6</td>
</tr>
<tr>
<td>Self Snoop</td>
<td>3-213</td>
</tr>
<tr>
<td>GETSEC</td>
<td>6-2, 6-3, 6-5</td>
</tr>
<tr>
<td>SENTER sleep state</td>
<td>6-10</td>
</tr>
<tr>
<td>SETcc instructions</td>
<td>4-592</td>
</tr>
<tr>
<td>GETSEC</td>
<td>6-4</td>
</tr>
<tr>
<td>SF (sign) flag, EFLAGS register</td>
<td>3-32</td>
</tr>
<tr>
<td>SFENCE instruction</td>
<td>4-595</td>
</tr>
<tr>
<td>SGDT instruction</td>
<td>4-596</td>
</tr>
<tr>
<td>SHAF instruction</td>
<td>4-576</td>
</tr>
<tr>
<td>Shift instructions</td>
<td>4-578</td>
</tr>
<tr>
<td>SHL instruction</td>
<td>4-578</td>
</tr>
<tr>
<td>SHLD instruction</td>
<td>4-607</td>
</tr>
<tr>
<td>SHR instruction</td>
<td>4-578</td>
</tr>
<tr>
<td>SHRD instruction</td>
<td>4-610</td>
</tr>
<tr>
<td>SHUFPD - Shuffle Packed Double Precision Floating-Point Values</td>
<td>4-613, 4-652</td>
</tr>
<tr>
<td>SHUFPSP - Shuffle Packed Single Precision Floating-Point Values</td>
<td>4-618</td>
</tr>
<tr>
<td>SIB byte</td>
<td>2-3</td>
</tr>
<tr>
<td>32-bit addressing forms</td>
<td>2-7, 2-20</td>
</tr>
<tr>
<td>description</td>
<td>2-3</td>
</tr>
<tr>
<td>SIDT instruction</td>
<td>4-596, 4-622</td>
</tr>
<tr>
<td>Significand, extracting from floating-point number</td>
<td>3-424</td>
</tr>
<tr>
<td>SIMD floating-point exceptions, unmasking, effects of</td>
<td>3-524, 4-526, 5-554</td>
</tr>
<tr>
<td>Sine, x87 FPU operation</td>
<td>3-384, 3-386</td>
</tr>
</tbody>
</table>
INDEX

SINIT 6-4
SLDT instruction 4-624
GETSEC 6-4
SMSW instruction 4-626
SpeedStep technology 3-209
SQRTPD—Square Root of Double-Precision Floating-Point Values 4-652
SQRTPD—Square Root of Double-Precision Floating-Point Values 4-628
SQRTPS—Square Root of Single-Precision Floating-Point Values 4-631
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value 4-634, 4-652
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value 4-636
Square root, x87 FPU operation 3-374
Task register loading 3-551
storing 4-648
Task switch
CALL instruction 3-123
return from nested task, IRET instruction 3-480
TEST instruction 4-675, 5-546
Thermal Monitor
CPUID flag 3-213
Thermal Monitor 2 3-209
CPUID flag 3-209
Time Stamp Counter 3-212
Time-stamp counter, reading 4-541, 4-543
TLB entry, invalidating (flushing) 3-475
Trusted Platform Module 6-4, 6-5
TS (task switched) flag, CR0 register 3-146
TSS, relationship to task register 4-648
TZCNT - Count the Number of Trailing Zero Bits 4-677

UCOMISD - Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS 4-679
UCOMISD instruction 4-677
UCOMISS - Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS 4-681
UD2 instruction 4-683
Undefined, format opcodes 3-404
Unordered values 3-328, 3-404, 3-406
UNPCKHDPD- Unpack and Interleave High Packed Double-Precision Floating-Point Values 4-684
UNPCKHPS- Unpack and Interleave High Packed Single-Precision Floating-Point Values 4-688
UNPCKLPD- Unpack and Interleave Low Packed Double-Precision Floating-Point Values 4-692
UNPCKLPS- Unpack and Interleave Low Packed Single-Precision Floating-Point Values 4-696

V
VALIGND/VALIGNQ- Align Doubleword/Quadword Vectors 4-699, 5-5
VBLENDMPPD- Blend Float64 Vectors Using an OpMask Control 5-9
VCVTLPD2UDQ- Convert Packed Double-Precision Floating-Point Values to Packed Signed Doubleword Integers 5-27
VCVTLPD2UDQ- Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values 5-40
VCVTSD2USI- Convert Scalar Double-Precision Floating-Point Val-
ue to Unsigned Doubleword Integer 5-53
VCVTSS2USI- Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer 5-54
VCVTTPD2UDQ- Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers 5-58
VCVTTPS2UDQ- Convert with Truncation Packed Single-Precision Floating-Point Value to Unsigned Integer 5-63
VCVTSS2USI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer 5-69
VCVTSS2USI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer 5-70
VCVTUDQ2PD- Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values 4-699, 5-72
VCVTUDQ2PS- Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values 5-74
VCVTUSI2SD- Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value 5-80
VCVTUSI2SS- Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value 5-82
VERR instruction 5-92
Version information, processor 3-191
VERW instruction 5-92
VEX - 3-3
VEX.B - 3-3
VEXL - 3-3, 3-4
VEX.mmmm - 3-3
VEX.pp - 3-4, 3-5
VEX.R - 3-4
VEX.vvvv - 3-3
VEXW - 3-3
VEX.X - 3-3
VEXP2PD—Approximation to the Exponential 2^x of Packed Double-Precision Floating-Point Values with Less Than 2^-23 Relative Error 5-94
VEXP2PS—Approximation to the Exponential 2^x of Packed Single-Precision Floating-Point Values with Less Than 2^-23 Relative Error 6-10
VEXTRACTF128- Extract Packed Floating-Point Values 5-94
VFMADD132SS/VFMADD213SS/VFMADD231SS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values 5-137
VFMADD132SD/VFMADD213SD/VFMADD231SD - Fused Multiply-Alternating Add/Subtract of Scalar Double-Precision Floating-Point Values 5-140
VFMADD132PS/VFMADD213PS/VFMADD231PS - Fused Multiply-Add of Packed Single-Precision Floating-Point Values 5-193
VFMADD132SS/VFMADD213SS/VFMADD231SS - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values 5-196
VFMADD213PS/VFMADD231PS/VFMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values 5-199
VFMADD32PD/VFMADD213PD/VFMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values 5-206
VGATHERDPS/VGATHERDPD - Gather Packed Single, Packed Double with Signed Dword 5-255
VGATHERDPS/VGATHERQPS - Gather Packed SP FP values Using Signed Dword/Qword Indices 5-250
VGATHERFDPS/VGAHTHERFDQPS/VGATHERFDQPD/VGATHERFPDQPD - Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using TO Hint 5-258
VGATHERF1DPS/VGATHERF1QPS/VGATHERF1PD/VGATHERF1QPD - Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint 6-14
VGATHERQPS/VGATHERQPD - Gather Packed Single, Packed Double with Signed Qword Indices 5-258
VINSERTT128/VINSERTT32x4/VINSERTT64x4- Insert Packed Floating-Point Values 5-283
VINSERTT128/VINSERTT32x4/VINSERTT64x4- Insert Packed Integer Values 5-287
Virtual Machine Monitor 6-1
VM (virtual 8086 mode) flag, EFLAGS register 3-480
VMM - 6-1
VPBLENDMD- Blend Int32 Vectors Using an OpMask Control 5-298
VPBROADCASTM—Broadcast Mask to Vector Register 5-20
VPCTMOD/VPCMPUD - Compare Packed Integer Values into Mask 5-318
VPCTMM/VPCMPUQ - Compare Packed Integer Values into Mask 5-321
VPCONFLICTD/Q - Detect Conflicts Within a Vector of Packed Dword, Packed Qword Values into Dense Memory/Register 5-94
VPERM2128 - Permute Integer Values 5-336
VPERMIB2 - Full Permute of Bytes from Two Tables Overwriting the Index 5-5, 6-6
VPERMILDP- Permute Double-Precision Floating-Point Values 5-351
VPERMILPS- Permute Single-Precision Floating-Point Values 5-356
VPERMPD - Permute Double-Precision Floating-Point Elements 5-336
VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table 5-370
VPGATHERDD/VPGATHERQD - Gather Packed Dword, Packed Qword with Signed Dword Indices 5-385
VPGATHERQD/VPGATHERQQ - Gather Packed Qword values Using Signed Dword/Qword Indices 5-388
VPGATHERQD/VPGATHERQQ - Gather Packed Dword, Packed Qword with Signed Qword Indices 5-392
VPL2ZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values 5-395
INDEX

VPMOVDB/VPMOVSD/VPMOVUSDB - Down Convert DWord to Byte 5-408
VPMOVQD/VPMOVSEQD/VPMOVQUSDQ - Down Convert QWord to Word 5-423
VPMOVDW/VPMOVSDW/VPMOVUSDW - Down Convert DWord to Word 5-412
VPMOVQW/VPMOVSEQW/VPMOVQUSDW - Down Convert QWord to Word 5-427
VPTERNLOGD/VPTERNLOGQ - Bitwise Ternary Logic 5-466
VPTESTMD/VPTESTMQ - Logical AND and Set Mask 5-469
VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than 2^-28 Relative Error 5-499
VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error 5-499
VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error 6-22
VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error 6-26
VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than 2^-28 Relative Error 5-527
VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error 5-32
VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error 6-30
VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error 6-34
VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0QPD/VSCATTERPF0QPD —Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write 5-541
VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1QPD/VSCATTERPF1QPD —Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write 6-38

W
WAIT/FWAIT instructions 5-551
GETSEC 6-4
WBINVD instruction 5-552
WBINVD/INVd bit 3-193
Write-back and invalidate caches 5-552
WRMSR instruction 5-556
CPUID flag 3-212
X
x87 FPU
    checking for pending x87 FPU exceptions 5-551
    constants 3-358
    initialization 3-349
    instruction opcodes A-20
x87 FPU control word
load 3-360, 3-362
RC field 3-351, 3-358, 3-390
restoring 3-377
saving 3-379, 3-394
storing 3-392
x87 FPU data pointer 3-362, 3-377, 3-379, 3-394
x87 FPU instruction pointer 3-362, 3-377, 3-379, 3-394
x87 FPU last opcode 3-362, 3-377, 3-379, 3-394
x87 FPU status word
    condition code flags 3-328, 3-344, 3-404, 3-406, 3-409
    loading 3-362
    restoring 3-377
    saving 3-379, 3-394, 3-396
    TOP field 3-348
x87 FPU flags affected by instructions 3-15
x87 FPU tag word 3-362, 3-377, 3-379, 3-394
XABORT - Transaction Abort 5-563
XADD instruction 3-541, 5-565
Xchg instruction 3-541, 5-570
XCR 5-608, 5-609
XEND - Transaction End 5-572
XGETBV 5-574, 5-586, 5-591, B-42
XLAB instruction 5-576
XLAT instruction 5-576
XOR instruction 3-541, 5-578
XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values 5-580
XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values 5-583
XRSTOR B-42
XSAVE 5-574, 5-589, 5-590, 5-593, 5-594, 5-595, 5-596, 5-597, 5-598, 5-599, 5-600, 5-601, 5-602, 5-603, 5-604, 5-606, 5-607, 5-608, 5-609, B-42
XSETBV 5-601, 5-608, B-42
XTSET - Test If In Transactional Execution 5-610
Z
ZF (zero) flag, EFLAGS register 3-182, 3-519, 3-546, 3-548, 4-546, 5-92

Vol. 2D  INDEX-11