x86 and xv6

CS 450: Operating Systems
Michael Saelee <saelee@iit.edu>
To work on an OS kernel, we must be intimately familiar with the underlying ISA, hardware, and system conventions

- x86 ISA
- PC architecture
- Unix, GCC, ELF, etc.
§ x86 ISA
- Intel IA-32 Software Developer’s Manuals (linked on course website) are comprehensive references
- Volume 1: Architectural Overview (e.g., regs, addressing)
- Volume 2: Instruction Set Reference
- Volume 3: Systems Programming Guide (e.g., mechanisms that let operating system control/configure hardware)
- (Majority of diagrams in slides are from these manuals)
x86 Family of ISAs

- Started with Intel’s 8086 16-bit CPU in 1978
- Followed by 80186, 80286, 80386, 80486 (then Pentium …)
- 80386 introduced 32-bit addressing ("IA-32" architecture)
- CISC-style ISA
- Large instruction set, complex addressing modes
Backwards Compatibility

- “x86” implies backwards compatibility all the way to 8086
- All x86 CPUs boot into 16-bit “real address mode” (aka “real mode”)
  - Supported CPUs can switch into 32-bit “Protected Mode”
  - i.e., we need to understand real mode to write an OS!
Instruction Set Overview

- Arithmetic: add, sub, and, etc.
- Moving data: mov, push, pop, etc.
- Control flow: jmp, call, ret, etc.
- I/O: in, out
- Privileged: int, iret, hlt, etc.
Instruction formats:

- 0 operands, e.g., `ret`
- 1 operand, e.g., `pushl %ebp`
- 2 operands — OP SRC, DST — e.g., `movl $0xa, %eax`
- Operands may be immediate values (constants), registers, memory addresses
NB: we’ll be using *AT&T syntax* for x86 assembly

- output by gcc/gas
  
  - Constants are prefixed with $, Register names with %
  
  - Instruction suffixes (b=8-bit, w=16-bit, l=32-bit, etc.) used to indicate operand sizes

- not the same as official Intel syntax! (output by NASM)
General-Purpose Registers

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EBP</td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESI</td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EDI</td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESP</td>
</tr>
</tbody>
</table>

**Figure 3-5. Alternate General-Purpose Register Names**

- **EAX** — Accumulator for operands and results data
- **EBX** — Pointer to data in the DS segment
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)
EFLAGS register used for conditional operations
E.g., if last operation resulted in zero/nonzero/neg/etc.

```assembly
    movl $0, %eax
    .L0:
    addl $1, %eax
    cmpl $10, %eax
    jne .L0
```

```c
for (i=0; i<10; i++);  
    addl $1, %eax  
    cmpl $10, %eax  
    jne .L0  
```

# 10-eax
# jump if ZF≠0
As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

### 3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

- **CF (bit 0)** Carry flag — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

- **PF (bit 2)** Parity flag — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

- **AF (bit 4)** Auxiliary Carry flag — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

- **ZF (bit 6)** Zero flag — Set if the result is zero; cleared otherwise.

- **SF (bit 7)** Sign flag — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

- **OF (bit 11)** Overflow flag — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

---

**Figure 3-8. EFLAGS Register**

(only bottom 16-bits in real mode)
IP (16-bit) / EIP (32-bit) is *instruction pointer* register (PC elsewhere)
- always points to next instruction; automatically incremented
- change explicitly with jump, call, ret, etc.
Addressing modes:

- **Direct**: `movl 0x401000, %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t \ *) (0x401000) \]

- **Indirect**: `movl (%ebx), %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t \ *) \text{ebx} \]
Addressing modes (continued):

- **Base-Displacement**: `movl 8(%ebx), %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t \ *) (\text{ebx} + 8) \]

- **Indexed & Scaled**: `movl (%ebx, %ecx, 4), %eax`
  \[ \approx \text{eax} = *(\text{uint32}_t \ *) (\text{ebx} + \text{ecx} \times 4) \]
16-bit addressing modes:
\[
\begin{align*}
&\{\text{CS : }\} \\
&\{\text{DS : }\} \\
&\{\text{SS : }\} \\
&\{\text{ES : }\}
\end{align*}
\]
\[
\left[\begin{array}{c}
\text{BX} \\
\text{BP}
\end{array}\right] + \left[\begin{array}{c}
\text{SI} \\
\text{DI}
\end{array}\right] + \text{[displacement]}
\]

32-bit addressing modes:
\[
\begin{align*}
&\{\text{CS : }\} \\
&\{\text{DS : }\} \\
&\{\text{SS : }\} \\
&\{\text{ES : }\} \\
&\{\text{FS : }\} \\
&\{\text{GS : }\}
\end{align*}
\]
\[
\left[\begin{array}{c}
\text{EAX} \\
\text{EBX} \\
\text{ECX} \\
\text{EDX} \\
\text{ESP} \\
\text{EBP} \\
\text{ESI} \\
\text{EDI}
\end{array}\right] + \left[\begin{array}{c}
\text{EAX} \\
\text{EBX} \\
\text{ECX} \\
\text{EDX} \\
\text{ESP} \\
\text{EBP} \\
\text{ESI} \\
\text{EDI}
\end{array}\right] * \left[\begin{array}{c}
1 \\
2 \\
4 \\
8
\end{array}\right] + \text{[displacement]}
\]

(Courtesy WikiMedia Commons)
Real mode addressing

- 8086 has 16-bit registers, but **20-bit physical addresses**
- Use one of four 16-bit segment registers: CS, DS, SS, ES
  - Shift left by 4 bits (i.e., $\times 16$) to obtain a segment base address
- Add to virtual address to obtain physical address
Real mode addressing

- Code and Stack accesses using IP, SP, and BP automatically use CS (code segment) and SS (stack segment) values

- e.g., if IP=0x4000 and CS=0x1100, CS:IP refers to absolute address $0x1100 \times 16 + 0x4000 = 0x15000$

- Can be confusing and unwieldy (especially if data straddles segments)
Protected mode addressing

- Full 32-bit addresses stored in registers

- Segment registers (expanded to CS, DS, SS, ES, FS, GS, and still all 16-bit) no longer hold base addresses, but selectors

- Selectors are used to load segment descriptors from a descriptor table which describe location/size/status/etc. of segments
If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor’s address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

See also: Chapter 4, “Paging.”

3.4.1 Logical Address Translation in IA-32e Mode

In IA-32e mode, an Intel 64 processor uses the steps described above to translate a logical address to a linear address. In 64-bit mode, the offset and base address of the segment are 64-bits instead of 32 bits. The linear address format is also 64 bits wide and is subject to the canonical form requirement.

Each code segment descriptor provides an L bit. This bit allows a code segment to execute 64-bit code or legacy 32-bit code by code segment.

3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

- **Index** (Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).
- **TI (table indicator) flag** (Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

Figure 3-5. Logical Address to Linear Address Translation
Segmentation

- Recall: segmentation allows virtual addresses to be translated using segment base addresses

- Segment descriptors also allow for access control (e.g., restricted access to certain segments)

- Mapping from segmented to linear address can be simple/flat or arbitrarily complex!
PROTECTED-MODE MEMORY MANAGEMENT

programs to multi-segmented models that employ segmentation to create a robust operating environment in which multiple programs and tasks can be executed reliably.

The following sections give several examples of how segmentation can be employed in a system to improve memory management performance and reliability.

3.2.1 Basic Flat Model

The simplest memory model for a system is the basic "flat model," in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment (see Figure 3-2). Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes. By setting the segment limit to 4 GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address. ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFFF_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

Figure 3-2. Flat Model
More complexity can be added to this protected flat mode to provide more protection. For example, for the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0 in the linear address space. This flat segmentation model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other.

Similar designs are used by several popular multitasking operating systems.

3.2.3 Multi-Segment Model

A multi-segment model (such as the one shown in Figure 3-4) uses the full capabilities of the segmentation mechanism to provide hardware enforced protection of code, data structures, and programs and tasks. Here, each program (or task) is given its own table of segment descriptors and its own segments. The segments can be completely private to their assigned programs or shared among programs. Access to all segments and to the execution environments of individual programs running on the system is controlled by hardware.

Figure 3-3. Protected Flat Model
Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics. In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

3.2.5 Paging and Segmentation

Paging can be used with any of the segmentation models described in Figures 3-2, 3-3, and 3-4. The processor's paging mechanism divides the linear address space (into which segments are mapped) into pages (as shown in Figure 3-1). These linear-address-space pages are then mapped to pages in the physical address space. The paging mechanism offers several page-level protection facilities that can be used with or instead of the segment-
Segment Descriptor Tables

- Kernel is responsible for maintaining descriptor tables on a system wide (via Global Descriptor Table) or task-specific (via Local Descriptor Table) basis

- Part of growing list of kernel data structures!
Table Indicator (TI) flag

Index

Requested Privilege Level (RPL)

0 = GDT
1 = LDT

Figure 3-6. Segment Selector

Segment Selector

TI = 0

TI = 1

First Descriptor in GDT is Not Used

Figure 3-10. Global and Local Descriptor Tables
The flags and fields in a segment descriptor are as follows:

- **Segment limit field**: Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:
  - If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
  - If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP, for all segments other than SS) or stack-fault exceptions (#SS for the SS segment). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit plus 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment's address space, rather than at the top. IA-32 architecture stacks always grow downwards, making this mechanism convenient for expandable stacks.

- **Base address fields**: Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

- **Type field**: Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 5-1). See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for a description of how this field is used to specify code and data-segment types.

![Figure 3-8. Segment Descriptor](image)
CS and CPL

- Bottom 2 bits of CS indicate the CPL (current privilege level)

- Recall: 0 = highest, 3 = lowest — used to guard access to privileged/restricted instructions and memory

- CS segment selector cannot be manipulated directly (why?)

- Must be loaded from a descriptor (when switching segments)
xv6 and Segmentation

- xv6 mostly uses a flat model, so segmentation setup is simple

- But segmentation descriptors are very similar to those used in interrupt descriptor tables (IDTs)

- Used for carrying out interrupts and enforcing privilege level (CPL) policies — coming later
Paging

- Recall: paging allows for more granular mapping of linear address spaces onto physical memory
  - Fixed sized pages mapped from linear to physical address space
- We will use support for 2-level paging for 32-bit addresses:
  - 1K page directory entries $\rightarrow$ 1K page tables (each)
  - 4KB pages
Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the form of entries that map pages, those that reference other paging structures, and those that do neither because they are “not present”; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.
Segmentation & Paging

- Segmentation is set up to provide a (mostly) flat model in xv6
- Paging is used to do heavy lifting of virtual → physical memory mapping
- xv6 sets up paging structures on a per-process basis
If paging is not used, the linear address space of the processor is mapped directly into the physical address space of the processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage. When using paging, each segment is divided into pages (typically 4 KBytes each in size), which are stored either in physical memory or on the disk. The operating system or executive maintains a page directory and a set of page tables to keep track of the pages. When a program (or task) attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location.

If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a page-fault exception). The operating system or executive then reads the page into physical memory from the disk and continues executing the program.

When paging is implemented properly in the operating system or executive, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program. Even programs written for 16-bit IA-32 processors can be paged (transparently) when they are run in virtual-8086 mode.

3.2 USING SEGMENTS

The segmentation mechanism supported by the IA-32 architecture can be used to implement a wide variety of system designs. These designs range from flat models that make only minimal use of segmentation to protect...
### Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Address of page directory\(^1\) | Ignored | PCD | PWT | Ignored | CR3  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Address of page table | Ignored | 0   | Ign A | PCD | PWT | U/S | R/W | 1  | PDE: page table |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Ignored               | 0   | PDE: not present |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Address of 4KB page frame | Ignored | G   | PAT | D   | A   | PCD | PWT | U/S | R/W | 1  | PTE: 4KB page |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Ignored               | 0   | PTE: not present |

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.

2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.
Control & System Registers

- Transitioning between real & protected mode, activating paging, switching descriptor tables, etc., are all governed by control & system register settings

- Modifying these settings is restricted by CPL
CR8 — Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

CR0.PG — Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of register CR0) is not also set; setting the PG flag when the PE flag is clear causes a general-protection exception (#GP). See also: Chapter 4, “Paging.”

On Intel 64 processors, enabling and disabling IA-32e mode operation also requires modifying CR0.PG.

CR0.CD — Cache Disable (bit 30 of CR0) — When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor’s internal (and external) caches is enabled. When the CD flag is set, caching is restricted as described in Table 11-5. To prevent the processor from accessing and updating its caches, the CD flag must be set and the caches must be invalidated so that no cache hits can occur. See also: Section 11.5.3, “Preventing Caching,” and Section 11.5, “Cache Control.”

CR0.NW — Not Write-through (bit 29 of CR0) — When the NW and CD flags are clear, write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and invalidation cycles are enabled. See Table 11-5 for detailed information about the effect of the NW flag on caching for other settings of the CD and NW flags.

Figure 2-7. Control Registers
control registers are:

When loading a control register, reserved bits should always be set to the values previously read. The flags in Compatibility mode.

Register CR0 contains the following fields:

- **PE** (Protection enable): Determines whether the processor is operating in real-address mode, protected mode, or virtual-8086 mode.
- **LME** (Large Memory Extensions): Enables the processor to use physical addresses up to 16 TB in 64-bit mode.
- **CR0.PG** (Page-Guard): Enables the page-guarantee feature, which provides security for pages.
- **CR8** (Cache Disable): Enables or disables the processor's internal caches.
- **CR8.NW** (Not Write-through): Enables or disables write-back caching.
- **CR8.CD** (Cache Disable): Enables or disables write-through caching.
- **CR8.K** (K-mapping): Enables or disables K-mapping.
- **CR8.D** (Write-back): Enables or disables write-back caching.
- **CR8.P** (Write-protection): Enables or disables write protection.
- **CR8.W** (Write-through): Enables or disables write-through caching.
- **CR8.M** (Memory-type): Enables or disables memory-type caching.
- **CR8.E** (Execute): Enables or disables execution.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disables safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disables memory enable.
- **CR8.E** (Execute): Enables or disables execute.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disables safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disables memory enable.
- **CR8.E** (Execute): Enables or disables execute.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disabled safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disables memory enable.
- **CR8.E** (Execute): Enables or disables execute.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disables safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disables memory enable.
- **CR8.E** (Execute): Enables or disables execute.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disables safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disables memory enable.
- **CR8.E** (Execute): Enables or disables execute.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disables safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disables memory enable.
- **CR8.E** (Execute): Enables or disables execute.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disables safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disables memory enable.
- **CR8.E** (Execute): Enables or disables execute.
- **CR8.V** (Virtual): Enables or disables virtual mode.
- **CR8.U** (Unpredictable): Enables or disables unpredictable behavior.
- **CR8.S** (Safe): Enables or disables safe mode.
- **CR8.D** (Debug): Enables or disables debug mode.
- **CR8.W** (Write): Enables or disables write enable.
- **CR8.M** (Memory): Enables or disable...
.code16
.globl start
.start:
 cli

# Zero data segment registers DS, ES, and SS.
xorw %ax,%ax
# Set %ax to zero
movw %ax,%ds
# -> Data Segment

lgdt gdt descr
movl %cr0, %eax
orl $CR0_PE, %eax
movl %eax, %cr0

# Complete transition to 32-bit protected mode by using long jmp
# to reload %cs and %eip. The segment descriptors are set up with no
# translation, so that the mapping is still the identity mapping.
ljmp $(SEG_KCODE<<3), $start32

.code32
.globl start32
.start32:

# Set up the protected-mode data segment registers
movw $(SEG_KDATA<<3), %ax
# Our data segment selector
movw %ax, %ds
# -> DS: Data Segment

# Set up the stack pointer and call into C.
movl $start, %esp
call bootmain

xv6 bootstrap assembly
I/O

- Two basic models for accessing I/O devices:
  - Separate *I/O port* address space from data address space
  - Memory-mapped I/O
I/O Ports

- Separated, limited address space

- Special instructions (\texttt{in/out}) to explicitly read from /write to I/O ports

  - e.g., \texttt{in $0x60, %al} # get last typed character (PS/2 kbd)
Memory-mapped I/O

- I/O controllers responsible for mapping ranges of physical addresses to I/O devices
- Look like regular memory accesses (no special instructions)
- Accessing these special addresses read/write I/O devices
- Devices can also directly access physical memory (DMA)