Memory Hierarchy & Caching

Science

Computer Science

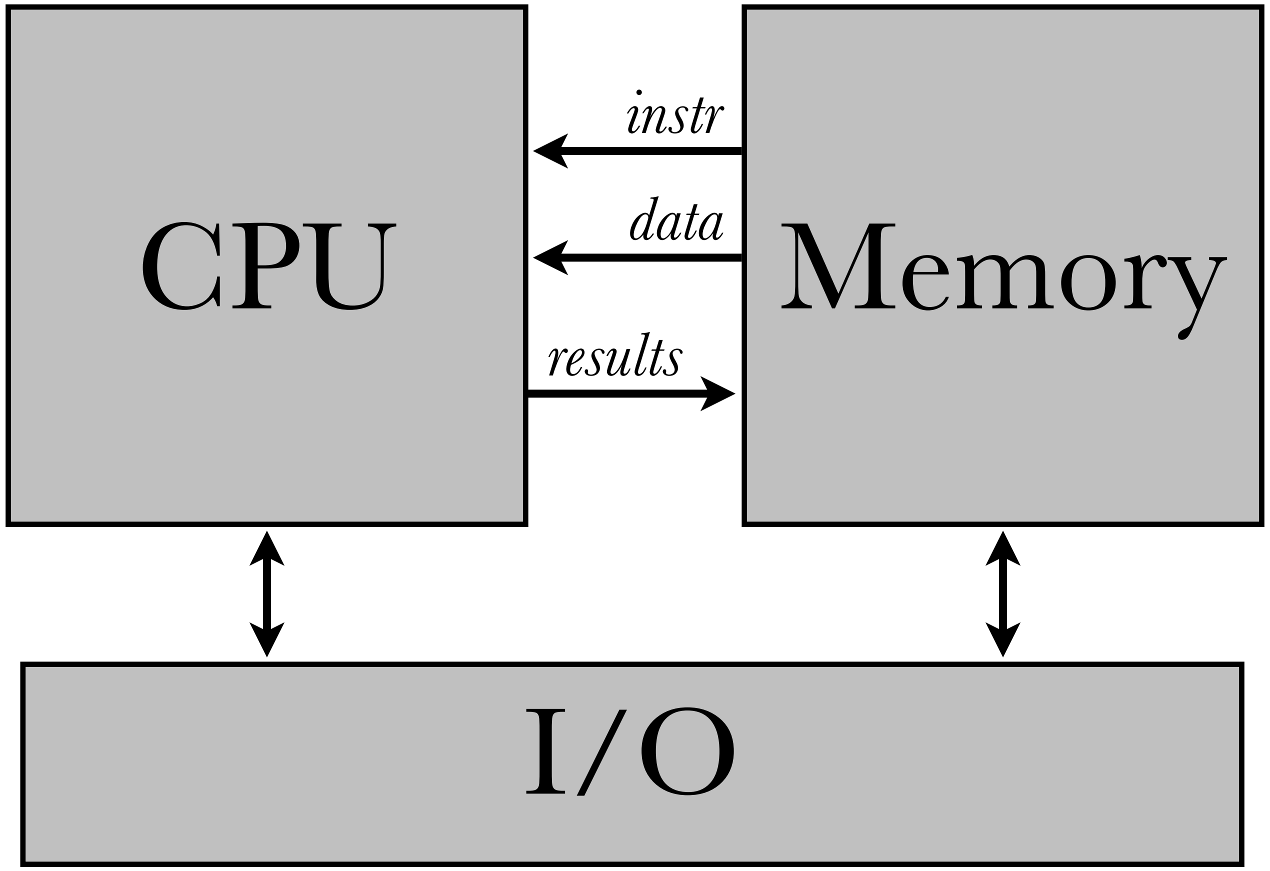
CS 351: Systems Programming

Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

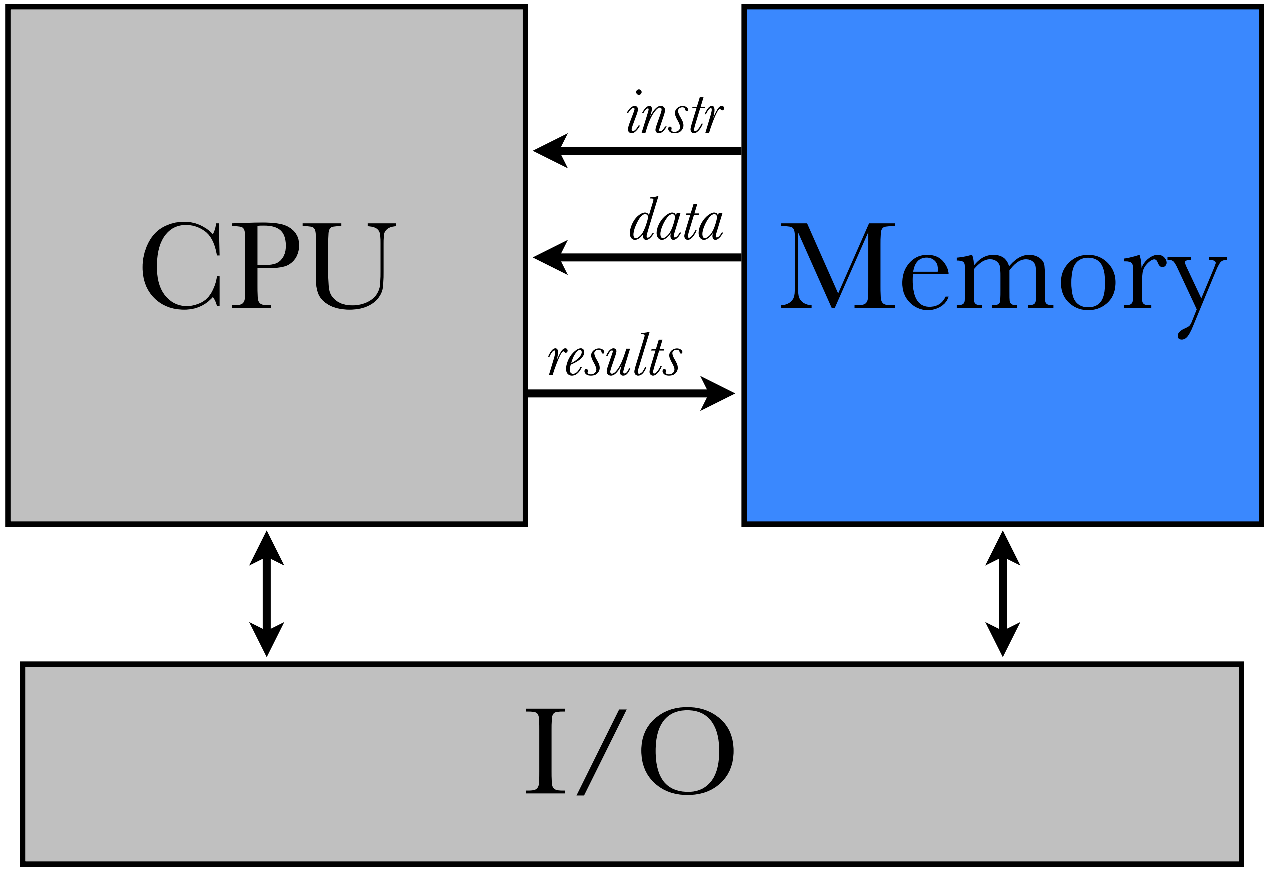
Why skip from process mgmt to memory?!

- recall: kernel facilitates process execution
    - via numerous *abstractions*
- exceptional control flow & process mgmt abstract functions of the CPU
- next big thing to abstract: memory!

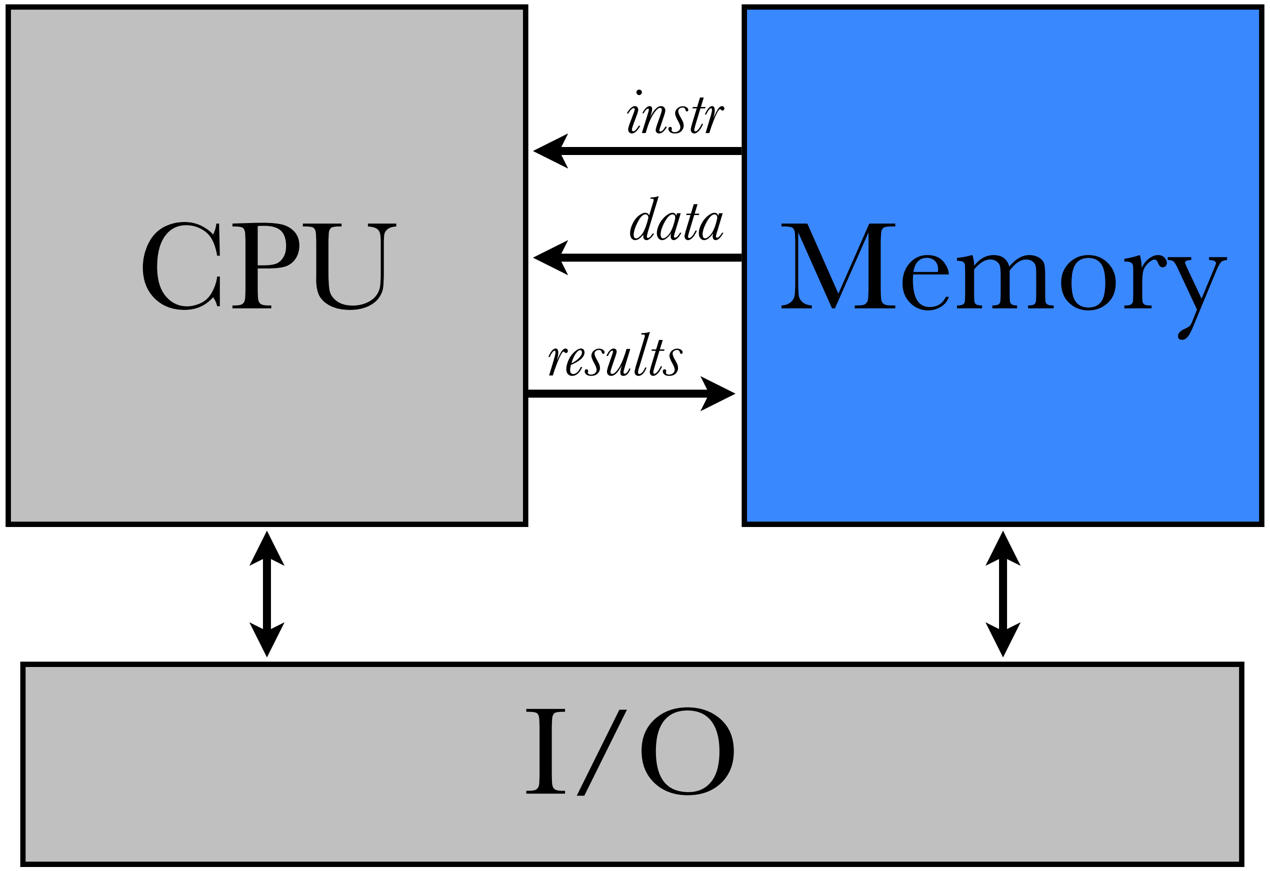


again, recall the *Von Neumann architecture*

— a *stored-program computer* with programs and data stored in the same memory

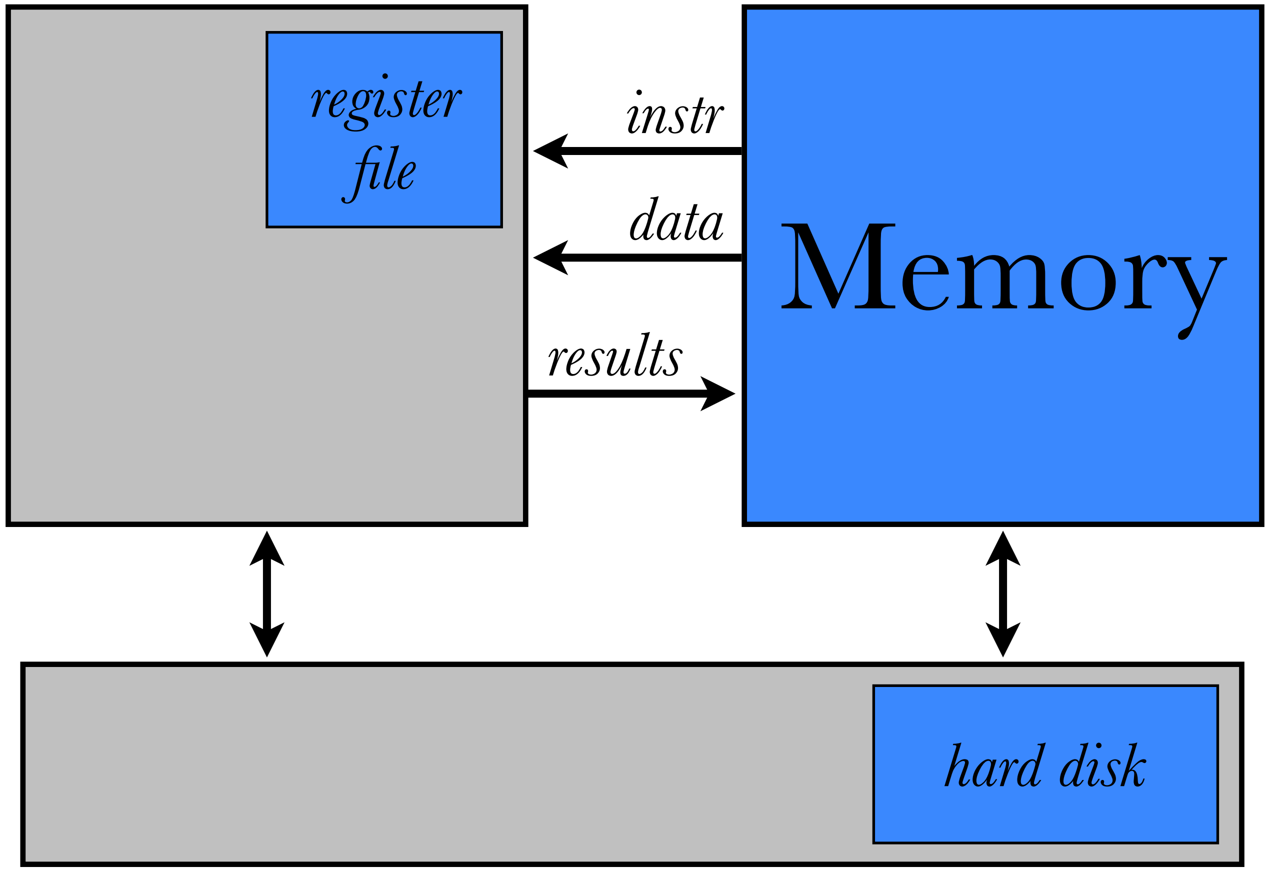


"memory" is an *idealized* storage device that holds our programs (instructions) and data (operands)



colloquially: “RAM”, *random access memory*

~ big array of byte-accessible data



in reality, “memory” is a combination of storage systems with very different access characteristics common types of "memory":

SRAM, DRAM, NVRAM, HDD

SRAM

- **S**tatic **R**andom **A**ccess **M**emory
- Data stable as long as power applied
- 6+ transistors (e.g. D-flip-flop) per bit
    - Complex & expensive, but fast!

DRAM

- **D**ynamic **R**andom **A**ccess **M**emory
- 1 capacitor + 1 transistor per bit
    - Requires period “refresh” @ 64ms
    - Much denser & cheaper than SRAM

NVRAM, e.g., Flash

- **N**on-**V**olatile **R**andom **A**ccess **M**emory
  - Data persists without power
- 1+ bits/transistor (low read/write granularity)
- Updates may require block erasure
- Flash has limited writes per block (100K+)

HDD

- **H**ard **D**isk **D**rive
- Spinning magnetic platters with multiple read/write "heads"
  - Data access requires *mechanical seek*

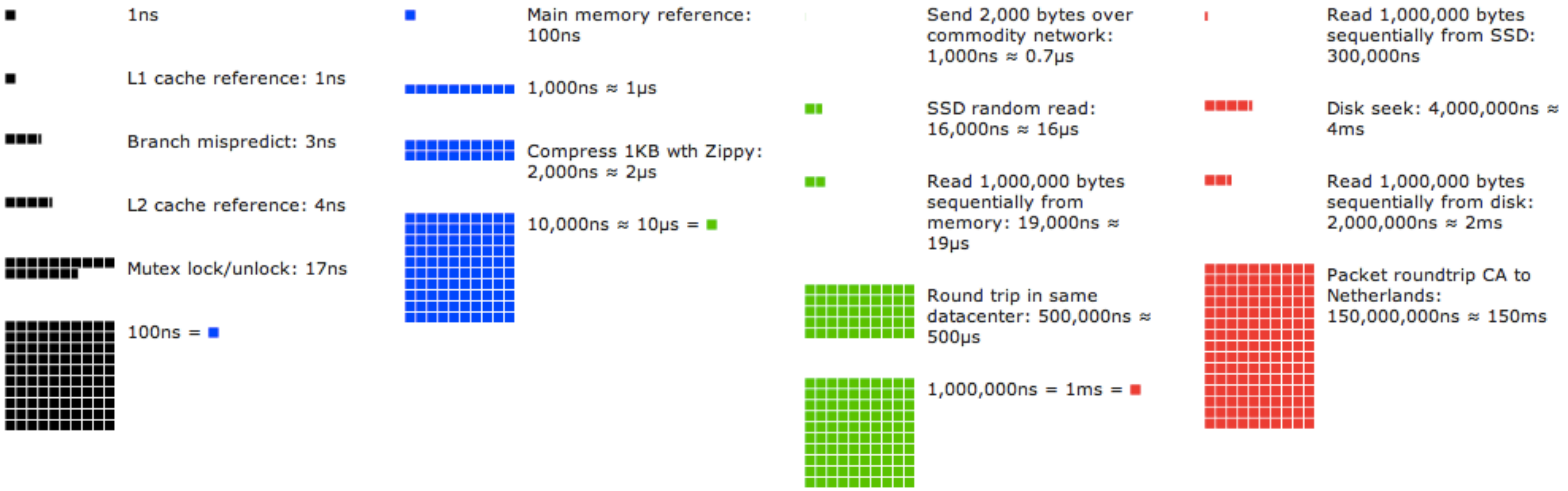
On *Distance*

- Speed of light $\approx 1 \times 10^9$ ft/s $\approx$ 1ft/ns
  - i.e., in 3GHz CPU, 4in / cycle
    - max access dist (round trip) = 2 in!
- Pays to keep things we need often *close* to the CPU!

Relative Speeds

Type	Size	Access latency	Unit
Registers	8 - 32 words	0 - 1 cycles	(ns)
On-board SRAM	32 - 256 KB	1 - 3 cycles	(ns)
Off-board SRAM	256 KB - 16 MB	~10 cycles	(ns)
DRAM	128 MB - 64 GB	~100 cycles	(ns)
SSD	≤ 1 TB	~10,000 cycles	(µs)
HDD	≤ 4 TB	~10,000,000 cycles	(ms)

human blink ≈ 350,000 µs



1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1µs

Compress 1KB wth Zippy: 2,000ns ≈ 2µs

10,000ns ≈ 10µs = 🟩

Send 2,000 bytes over commodity network: 1,000ns ≈ 0.7µs

SSD random read: 16,000ns ≈ 16µs

Read 1,000,000 bytes sequentially from memory: 19,000ns ≈ 19µs

Round trip in same datacenter: 500,000ns ≈ 500µs

1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 300,000ns

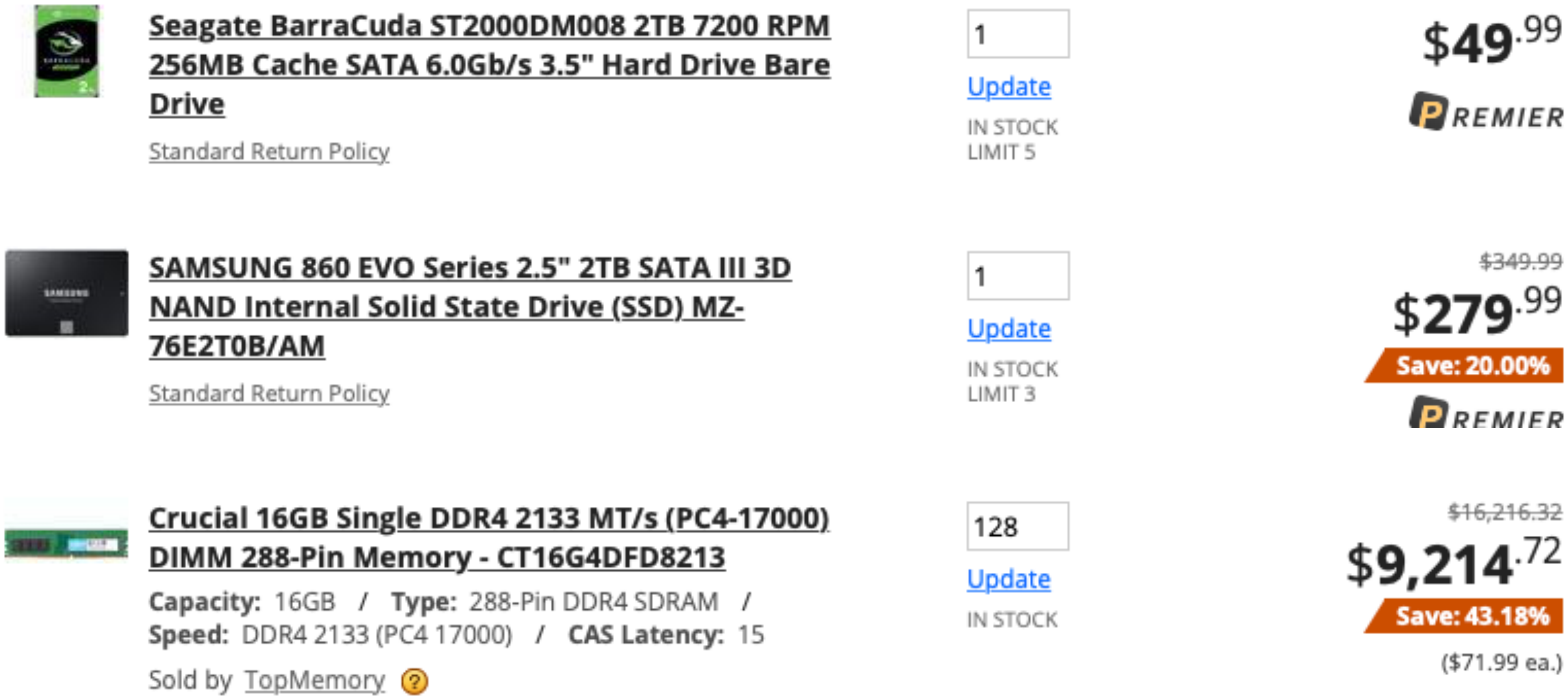
Disk seek: 4,000,000ns ≈ 4ms

Read 1,000,000 bytes sequentially from disk: 2,000,000ns ≈ 2ms

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

“Numbers Every Programmer Should Know”

http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html

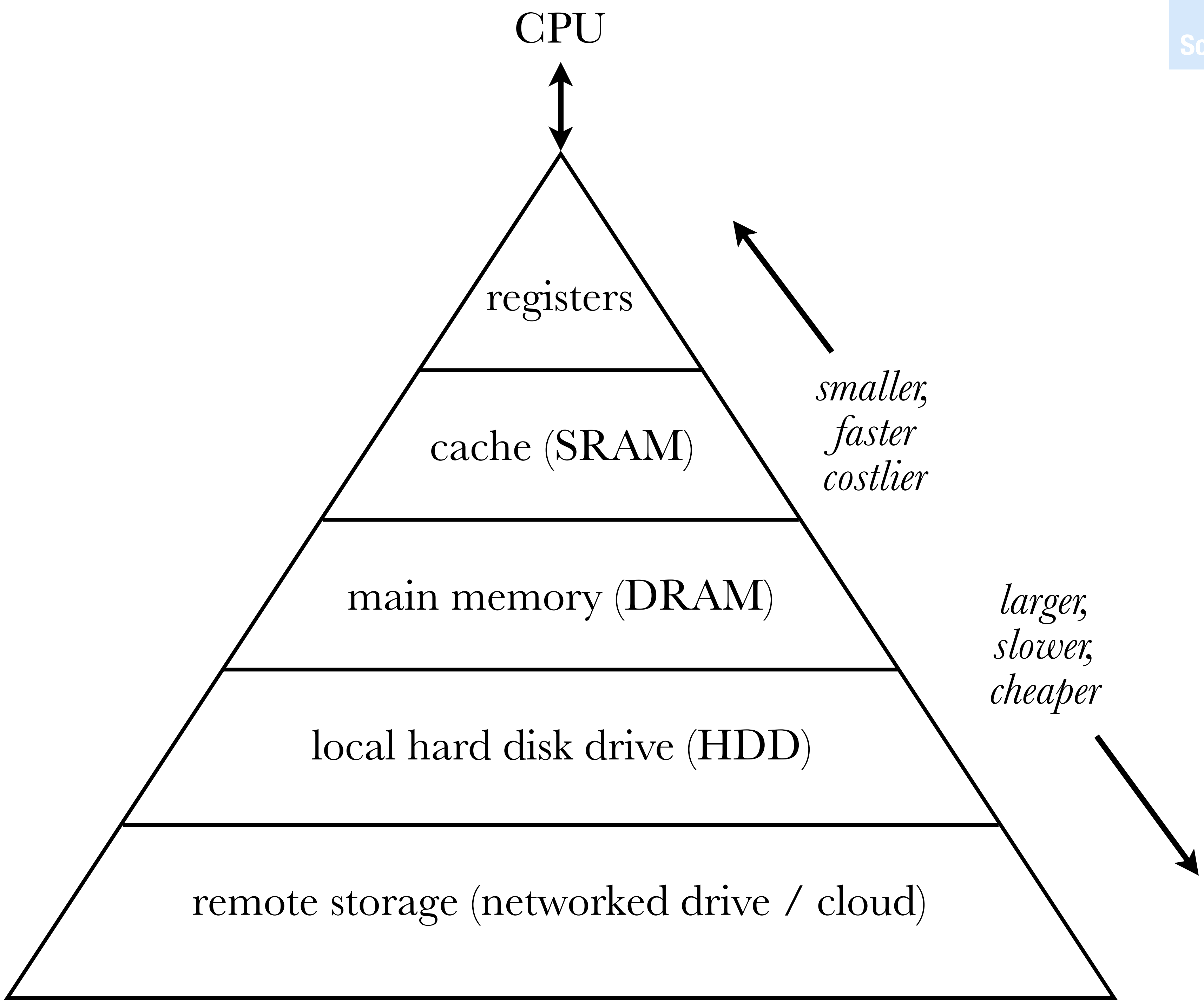


Seagate BarraCuda ST2000DM008 2TB 7200 RPM 256MB Cache SATA 6.0Gb/s 3.5" Hard Drive Bare Drive Standard Return Policy	1 Update IN STOCK LIMIT 5	$49.99 PREMIER
SAMSUNG 860 EVO Series 2.5" 2TB SATA III 3D NAND Internal Solid State Drive (SSD) MZ-76E2T0B/AM Standard Return Policy	1 Update IN STOCK LIMIT 3	~~$349.99~~ $279.99 Save: 20.00% PREMIER
Crucial 16GB Single DDR4 2133 MT/s (PC4-17000) DIMM 288-Pin Memory - CT16G4DFD8213 **Capacity:** 16GB / **Type:** 288-Pin DDR4 SDRAM / **Speed:** DDR4 2133 (PC4 17000) / **CAS Latency:** 15 Sold by TopMemory ?	128 Update IN STOCK	~~$16,216.32~~ $9,214.72 Save: 43.18% ($71.99 ea.)

(from newegg.com)

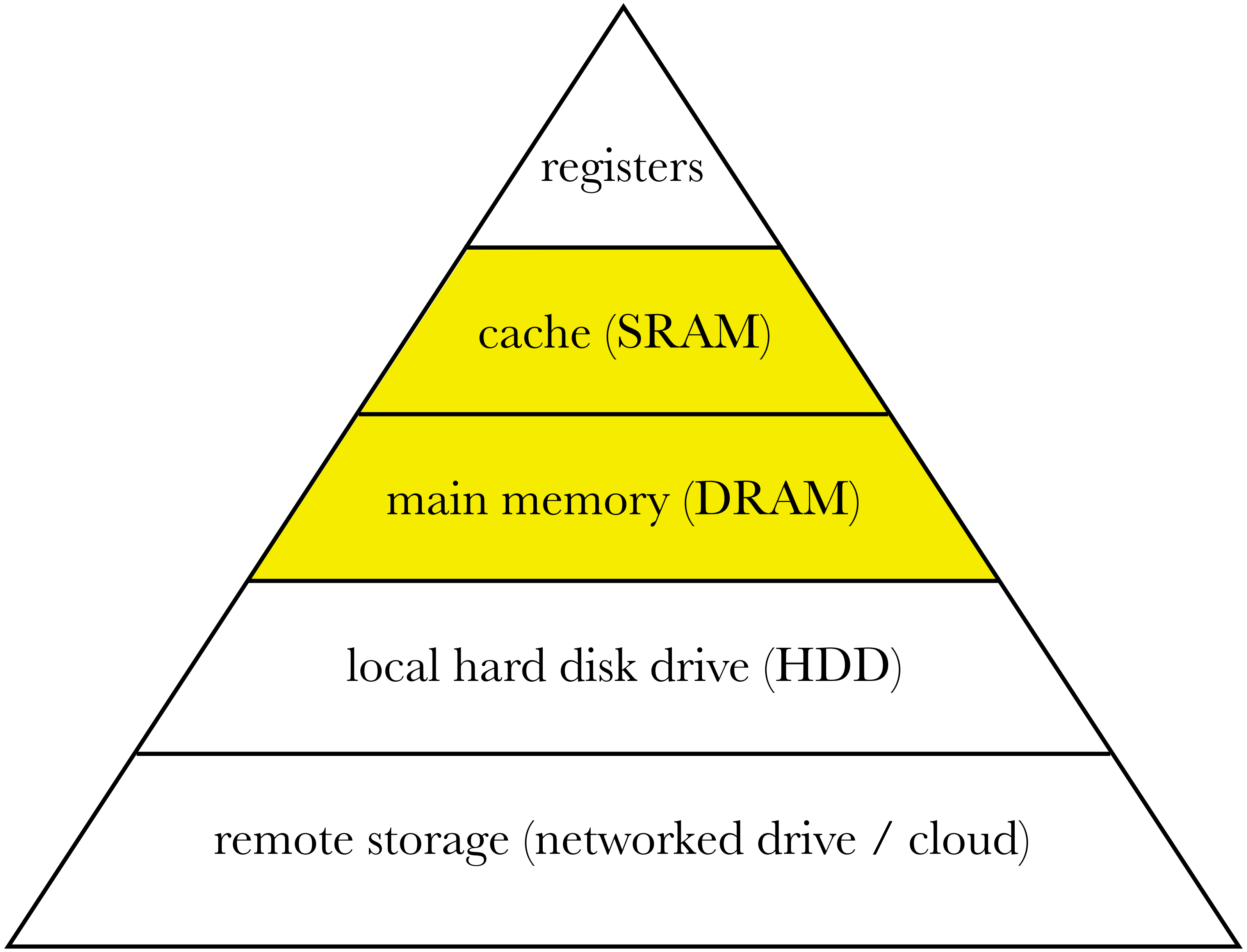
would like:

1. a lot of memory
2. fast access to memory
3. to not spend $$$ on memory


CPU
registers
cache (SRAM)
main memory (DRAM)
local hard disk drive (HDD)
remote storage (networked drive / cloud)
smaller, faster costlier
larger, slower, cheaper

an exercise in compromise: *the memory hierarchy*

idea: use the *fast but scarce* kind as much as possible; fall back on the *slow but plentiful* kind when necessary


registers
cache (SRAM)
main memory (DRAM)
local hard disk drive (HDD)
remote storage (networked drive / cloud)

boundary 1: SRAM ⇔ DRAM

§Caching

cache |kaSH|

verb

store away in hiding or for future use.

cache |kaSH|

noun

- a hidden or inaccessible storage place for valuables, provisions, or ammunition.
- (also **cache memory**) Computing an auxiliary memory from which high-speed retrieval is possible.

assuming SRAM cache starts out empty:

1. CPU requests data at memory address *k*
2. Fetch data from DRAM (or lower)
3. *Cache* data in SRAM for later use

after SRAM cache has been populated:

1. CPU requests data at memory address *k*
2. Check SRAM for *cached* data first; if there ("hit"), return it directly
3. If not there, update from DRAM

essential issues:

1. *what* data to cache
2. *where* to store cached data; i.e., how to *map* address k → cache slot
   - keep in mind SRAM ≪ DRAM

1. take advantage of *localities of reference*
    a. **temporal** locality
    b. **spatial** locality

a. **temporal** (time-based) locality:

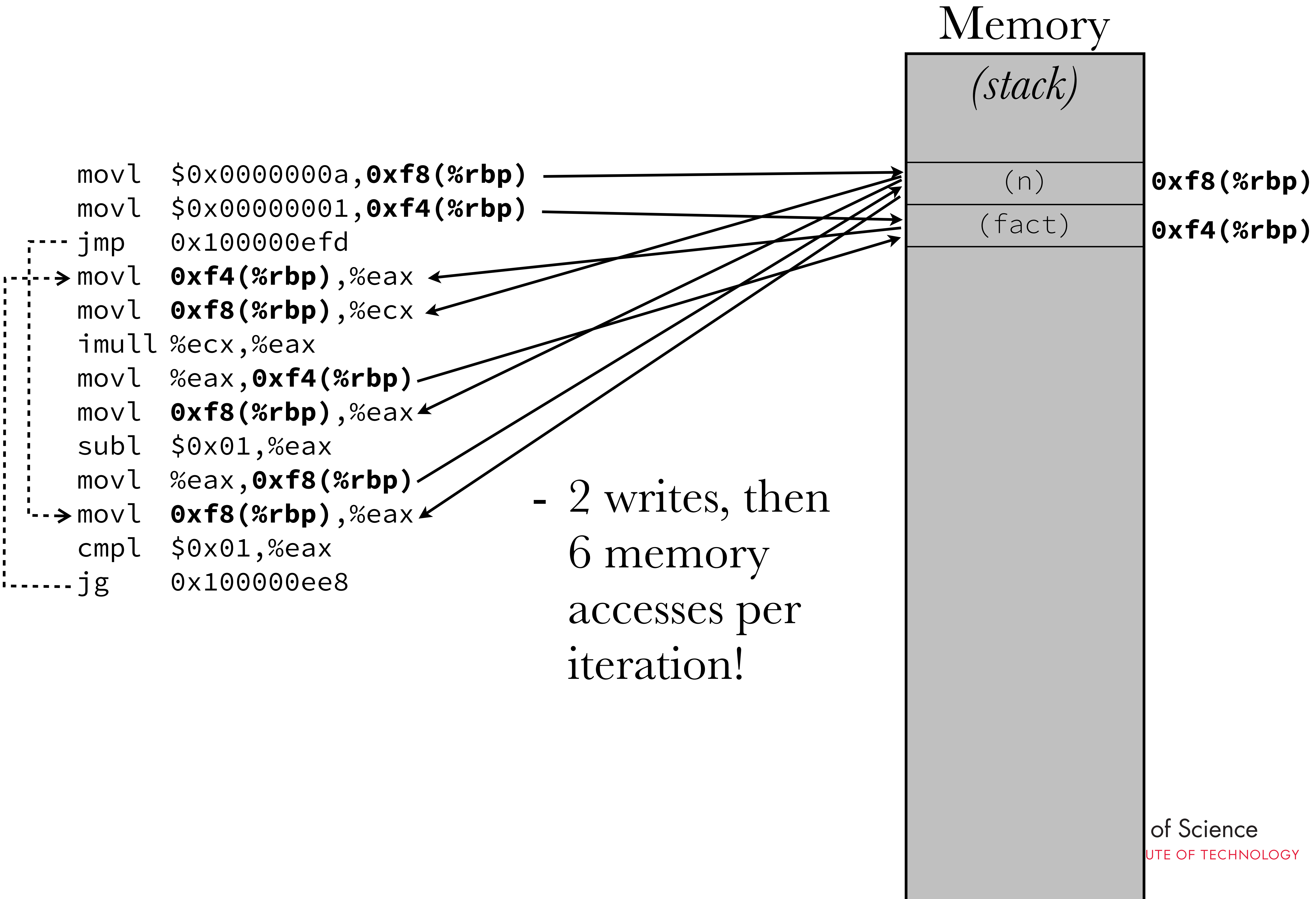
- if a datum was accessed recently, it’s likely to be accessed again soon

- e.g., accessing a loop counter;
  calling a function repeatedly

```
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}
```

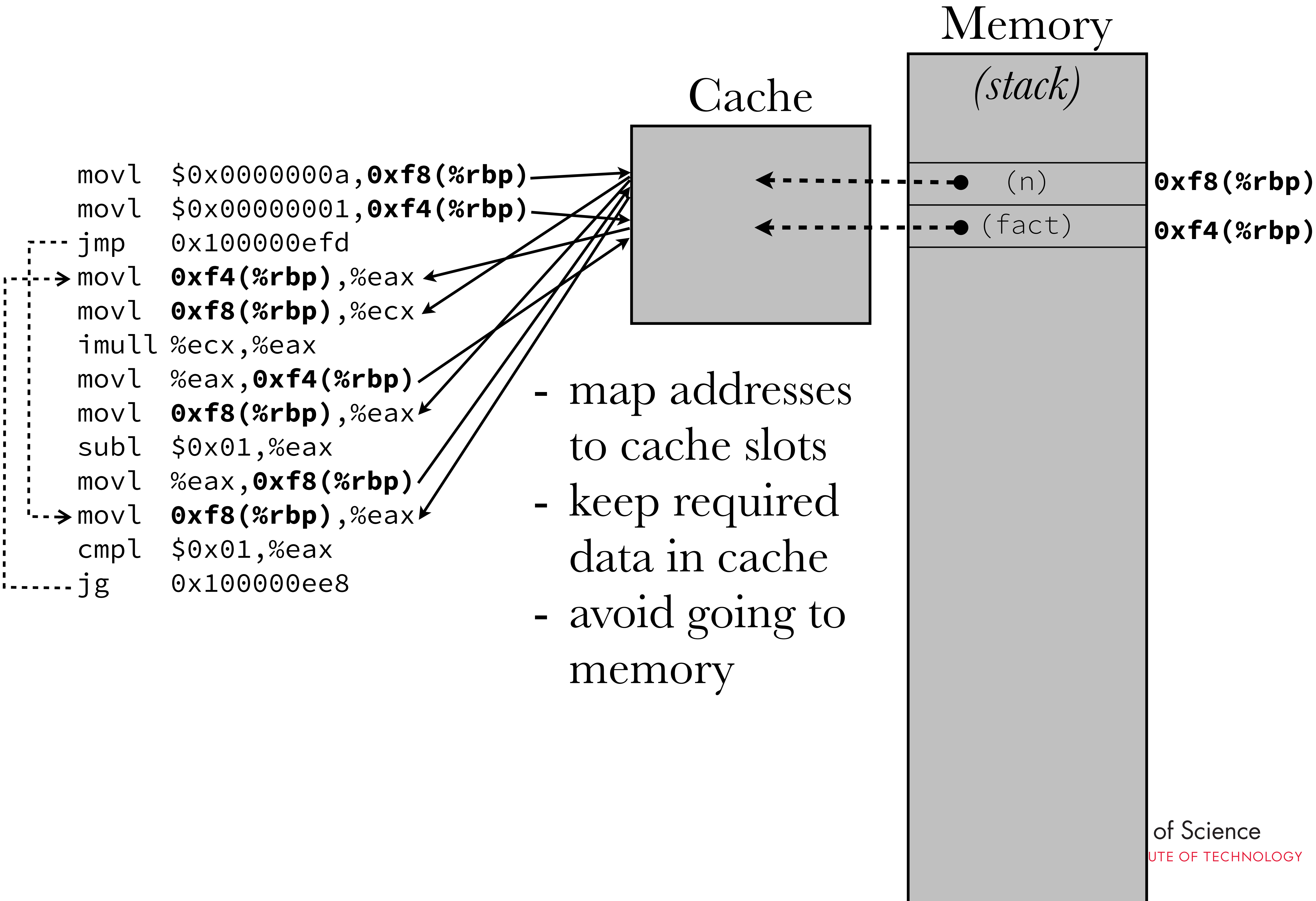
```
movl  $0x0000000a,0xf8(%rbp)   ; store n
movl  $0x00000001,0xf4(%rbp)   ; store fact
jmp   0x100000efd
movl  0xf4(%rbp),%eax          ; load fact
movl  0xf8(%rbp),%ecx          ; load n
imull %ecx,%eax                ; fact * n
movl  %eax,0xf4(%rbp)          ; store fact
movl  0xf8(%rbp),%eax          ; load n
subl  $0x01,%eax               ; n - 1
movl  %eax,0xf8(%rbp)          ; store n
movl  0xf8(%rbp),%eax          ; load n
cmpl  $0x01,%eax               ; if n>1
jg    0x100000ee8              ;   loop
```

(memory references in bold)



```
    movl  $0x0000000a,0xf8(%rbp)
    movl  $0x00000001,0xf4(%rbp)
    jmp   0x100000efd
    movl  0xf4(%rbp),%eax
    movl  0xf8(%rbp),%ecx
    imull %ecx,%eax
    movl  %eax,0xf4(%rbp)
    movl  0xf8(%rbp),%eax
    subl  $0x01,%eax
    movl  %eax,0xf8(%rbp)
    movl  0xf8(%rbp),%eax
    cmpl  $0x01,%eax
    jg    0x100000ee8
```

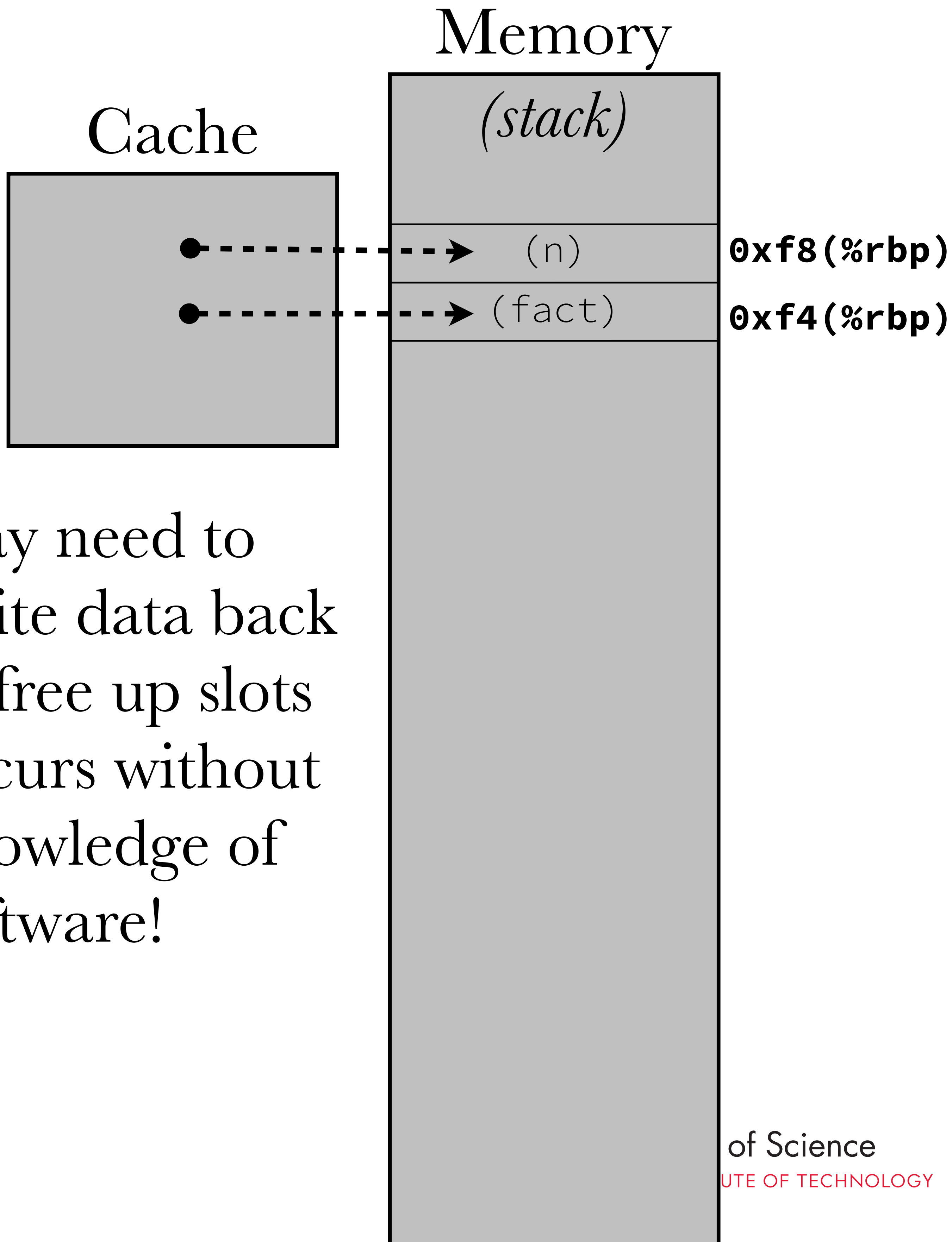
- 2 writes, then 6 memory accesses per iteration!



```
    movl  $0x0000000a,0xf8(%rbp)
    movl  $0x00000001,0xf4(%rbp)
    jmp   0x100000efd
    movl  0xf4(%rbp),%eax
    movl  0xf8(%rbp),%ecx
    imull %ecx,%eax
    movl  %eax,0xf4(%rbp)
    movl  0xf8(%rbp),%eax
    subl  $0x01,%eax
    movl  %eax,0xf8(%rbp)
    movl  0xf8(%rbp),%eax
    cmpl  $0x01,%eax
    jg    0x100000ee8
```

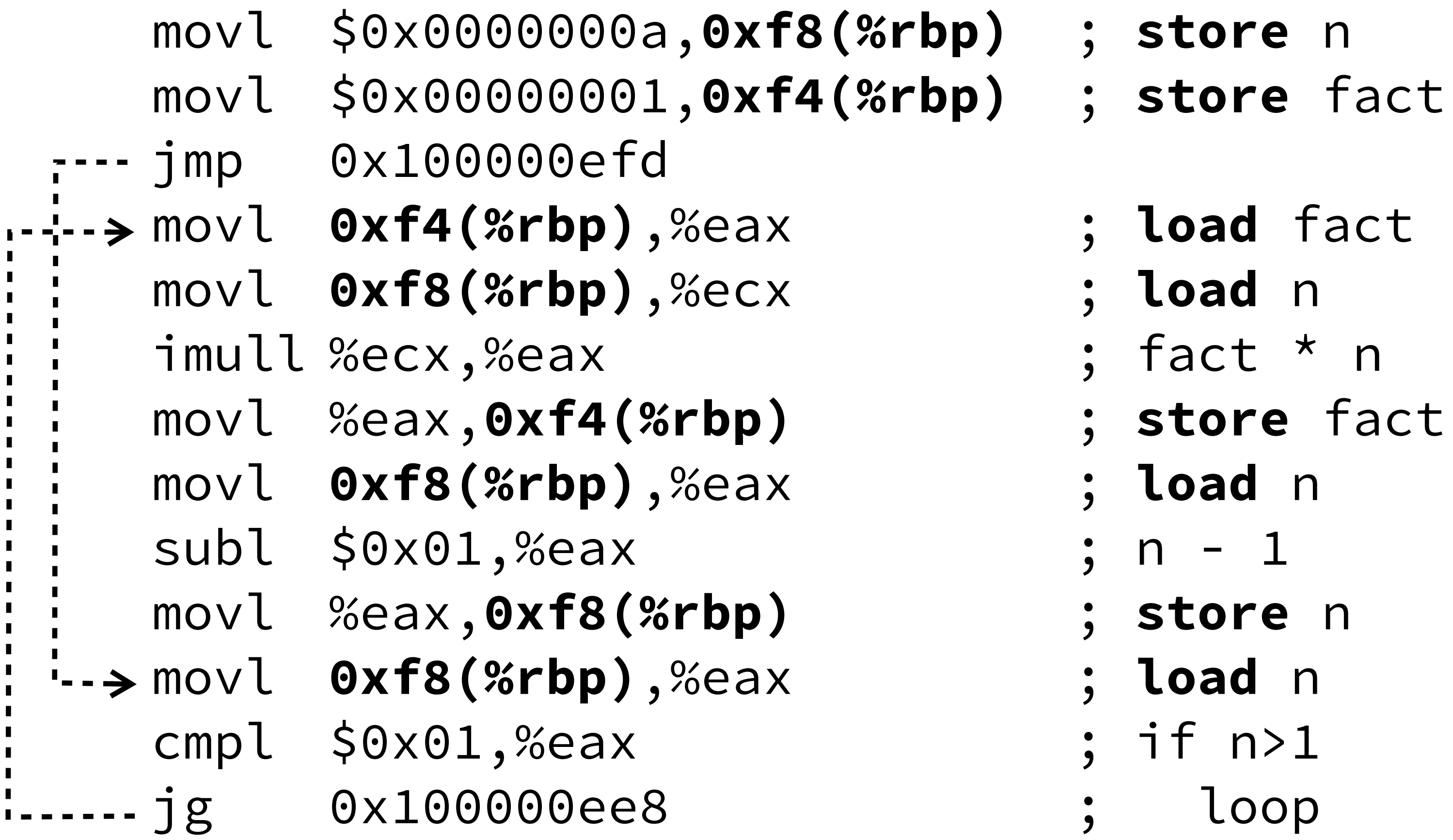
- map addresses to cache slots
- keep required data in cache
- avoid going to memory

```
     movl  $0x0000000a,0xf8(%rbp)
     movl  $0x00000001,0xf4(%rbp)
     jmp   0x100000efd
     movl  0xf4(%rbp),%eax
     movl  0xf8(%rbp),%ecx
     imull %ecx,%eax
     movl  %eax,0xf4(%rbp)
     movl  0xf8(%rbp),%eax
     subl  $0x01,%eax
     movl  %eax,0xf8(%rbp)
     movl  0xf8(%rbp),%eax
     cmpl  $0x01,%eax
     jg    0x100000ee8
```



- may need to write data back to free up slots
- occurs without knowledge of software!

```
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}
```



```
movl  $0x0000000a,0xf8(%rbp)  ; store n
movl  $0x00000001,0xf4(%rbp)  ; store fact
jmp   0x100000efd
movl  0xf4(%rbp),%eax         ; load fact
movl  0xf8(%rbp),%ecx         ; load n
imull %ecx,%eax               ; fact * n
movl  %eax,0xf4(%rbp)         ; store fact
movl  0xf8(%rbp),%eax         ; load n
subl  $0x01,%eax              ; n - 1
movl  %eax,0xf8(%rbp)         ; store n
movl  0xf8(%rbp),%eax         ; load n
cmpl  $0x01,%eax              ; if n>1
jg    0x100000ee8             ;   loop
```

… but this is really inefficient to begin with

```
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}
```

```
;; produced with gcc -O1
movl  $0x00000001,%esi  ; n
movl  $0x0000000a,%eax  ; fact
imull %eax,%esi         ; fact *= n
decl  %eax              ; n -= 1
cmpl  $0x01,%eax        ; if n≠1
jne   0x100000f10       ;   loop
```

compiler optimization: registers as “cache”

reduce/eliminate memory references *in code* using registers is an important technique, but doesn’t scale to even moderately large data sets (e.g., arrays)

one option: manage cache mapping directly from code

```
;; fictitious assembly
movl  $0x00000001,0x0000(%cache)
movl  $0x0000000a,0x0004(%cache)
imull 0x0004(%cache),0x0000(%cache)
decl  0x0004(%cache)
cmpl  $0x01,0x0004(%cache)
jne   0x100000f10
movl 0x0000(%cache),0xf4(%rbp)
movl 0x0004(%cache),0xf8(%rbp)
```

awful idea!

- code is tied to cache implementation; can't take advantage of hardware upgrades (e.g., larger cache)
- cache must be shared between processes (how to do this efficiently?)

caching is a hardware-level concern — job of the *memory management unit* (MMU)

but it's very useful to know how it works, so we can write *cache-friendly code*!

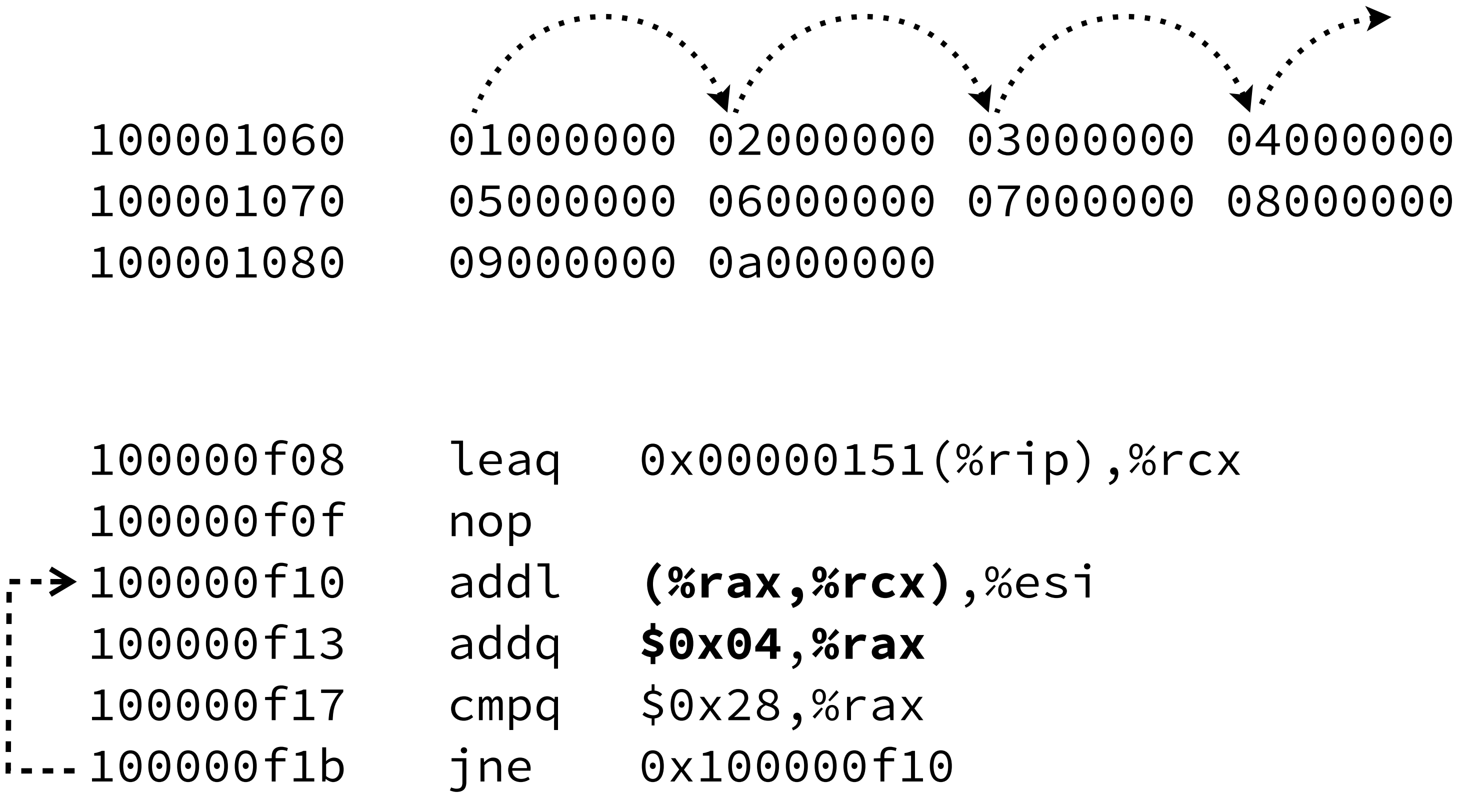
b. **spatial** (location-based) locality:

- after accessing data at a given address, data nearby are likely to be accessed

- e.g., sequential control flow;
  array access (with *stride n*)

stride length = 1 `int` (4 bytes)

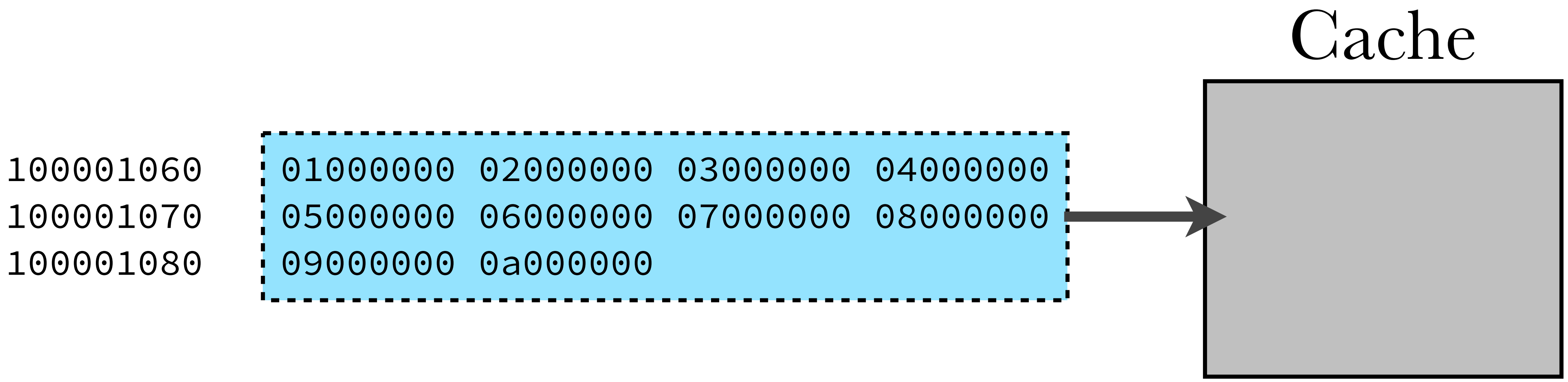
```
int arr[] = {1, 2, 3, 4, 5,
             6, 7, 8, 9, 10};

main() {
    int i, sum = 0;
    for (i=0; i<10; i++) {
        sum += arr[i];
    }
}
```



```
100001060    01000000 02000000 03000000 04000000
100001070    05000000 06000000 07000000 08000000
100001080    09000000 0a000000
```

```
   100000f08    leaq    0x00000151(%rip),%rcx
   100000f0f    nop
-->100000f10    addl    (%rax,%rcx),%esi
   100000f13    addq    $0x04,%rax
   100000f17    cmpq    $0x28,%rax
---100000f1b    jne     0x100000f10
```

Modern DRAM is designed to transfer *bursts* of data (~32-64 bytes) efficiently

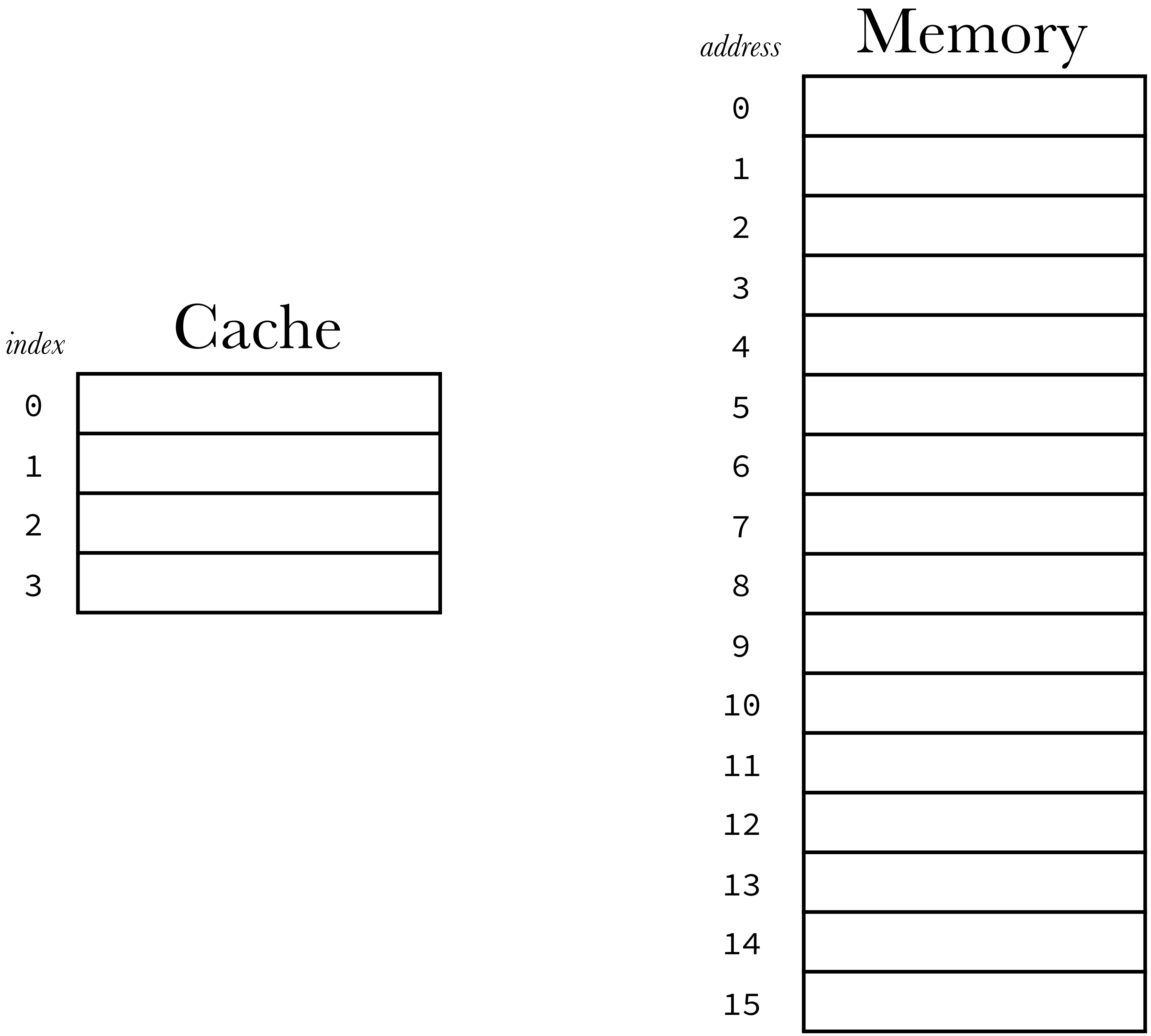


```
100001060    01000000 02000000 03000000 04000000
100001070    05000000 06000000 07000000 08000000
100001080    09000000 0a000000
```

idea: transfer array from memory to cache on accessing *first item*, then only access cache!

2. *where* to store cached data?
i.e., how to *map* address *k* → cache slot

§Cache Organization



Cache

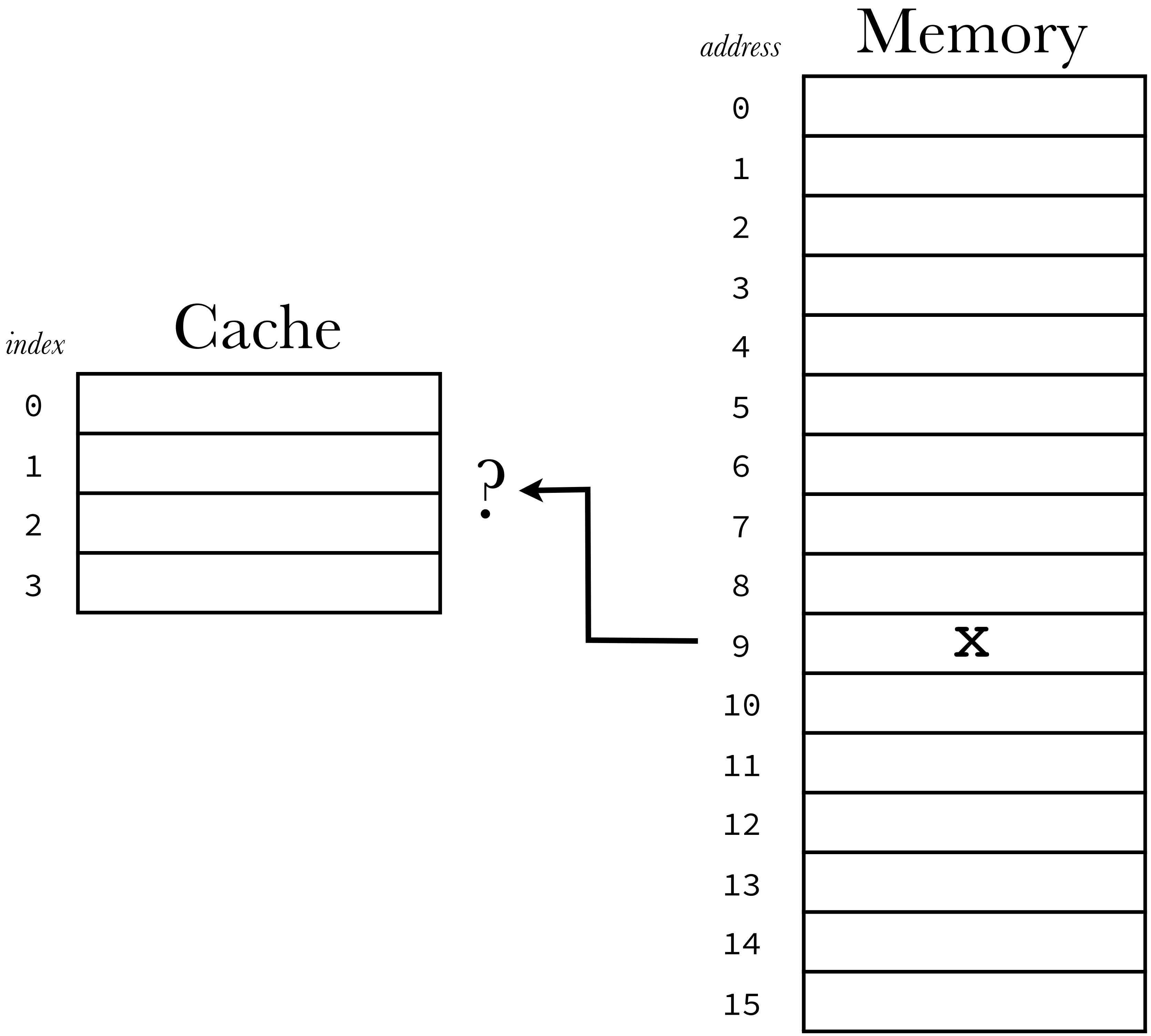
index

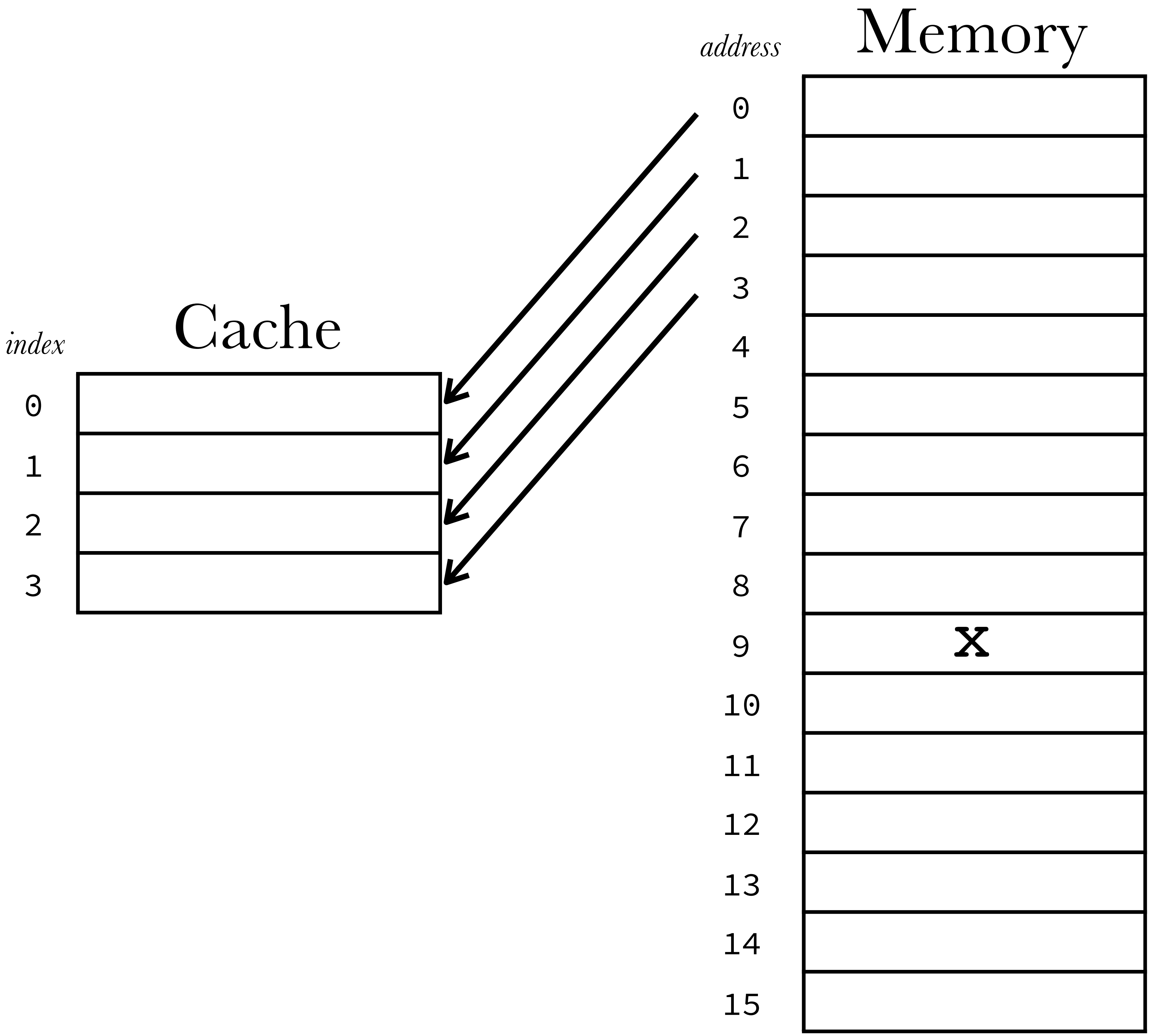
0
1
2
3

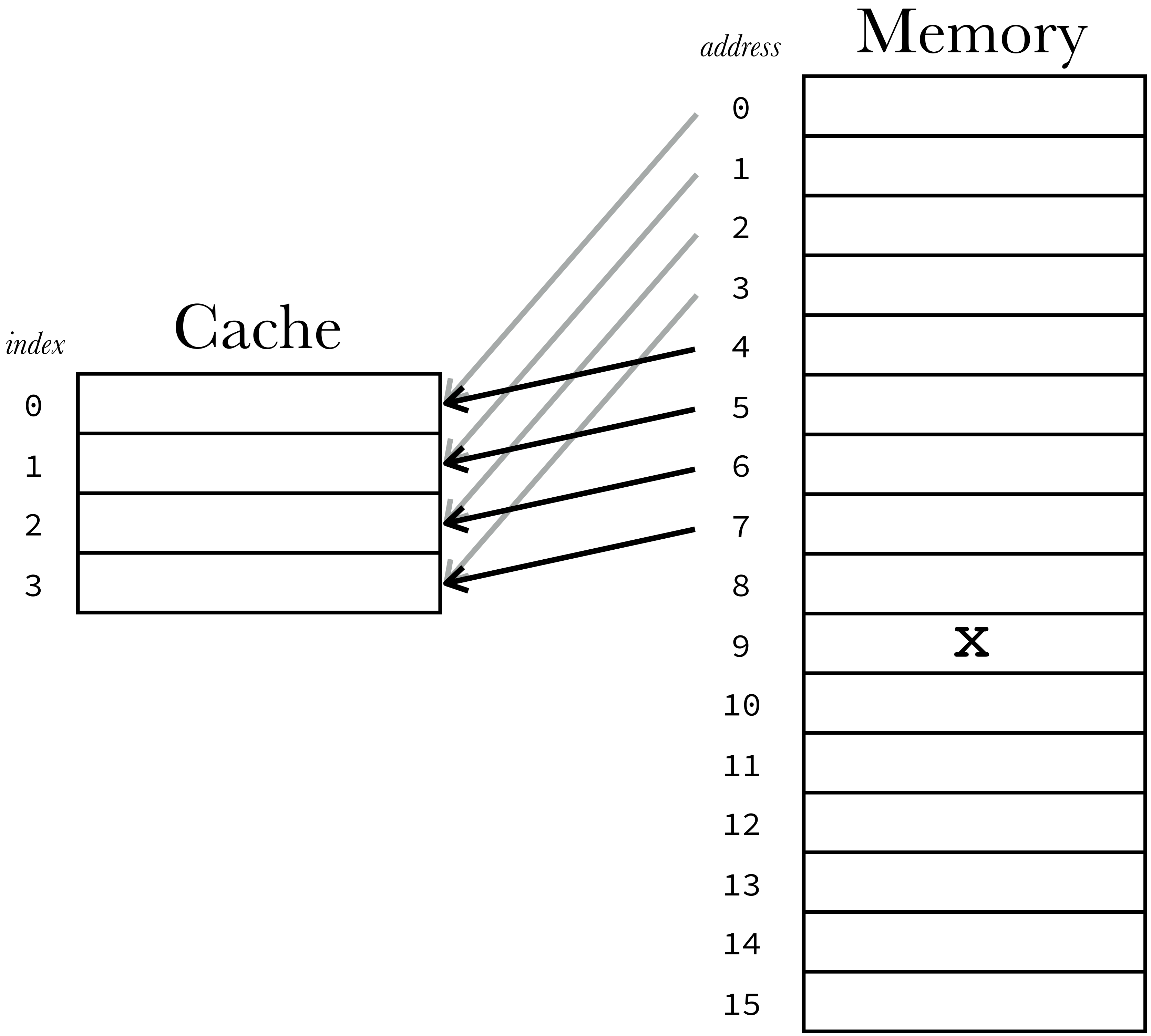
Memory

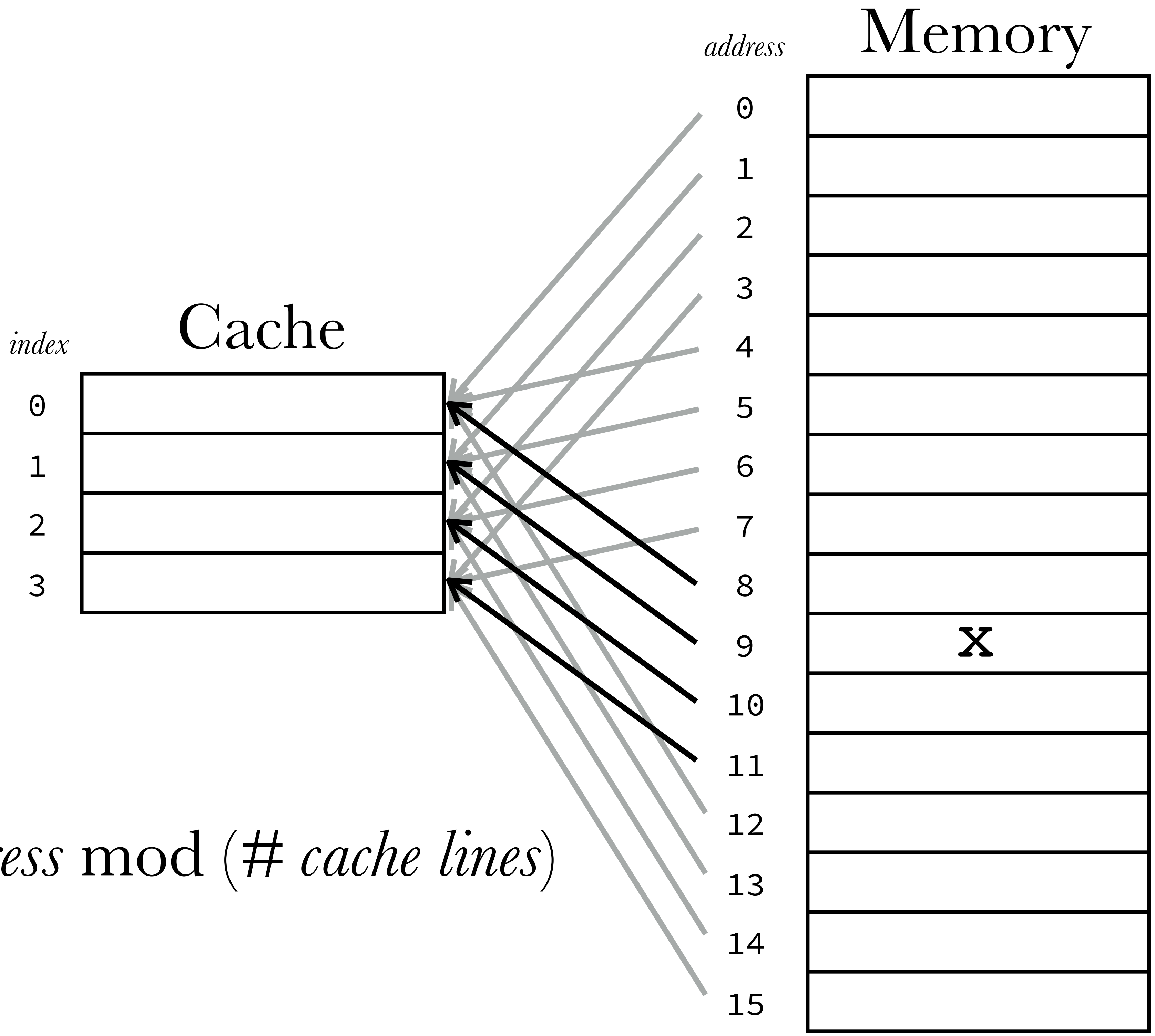
address

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

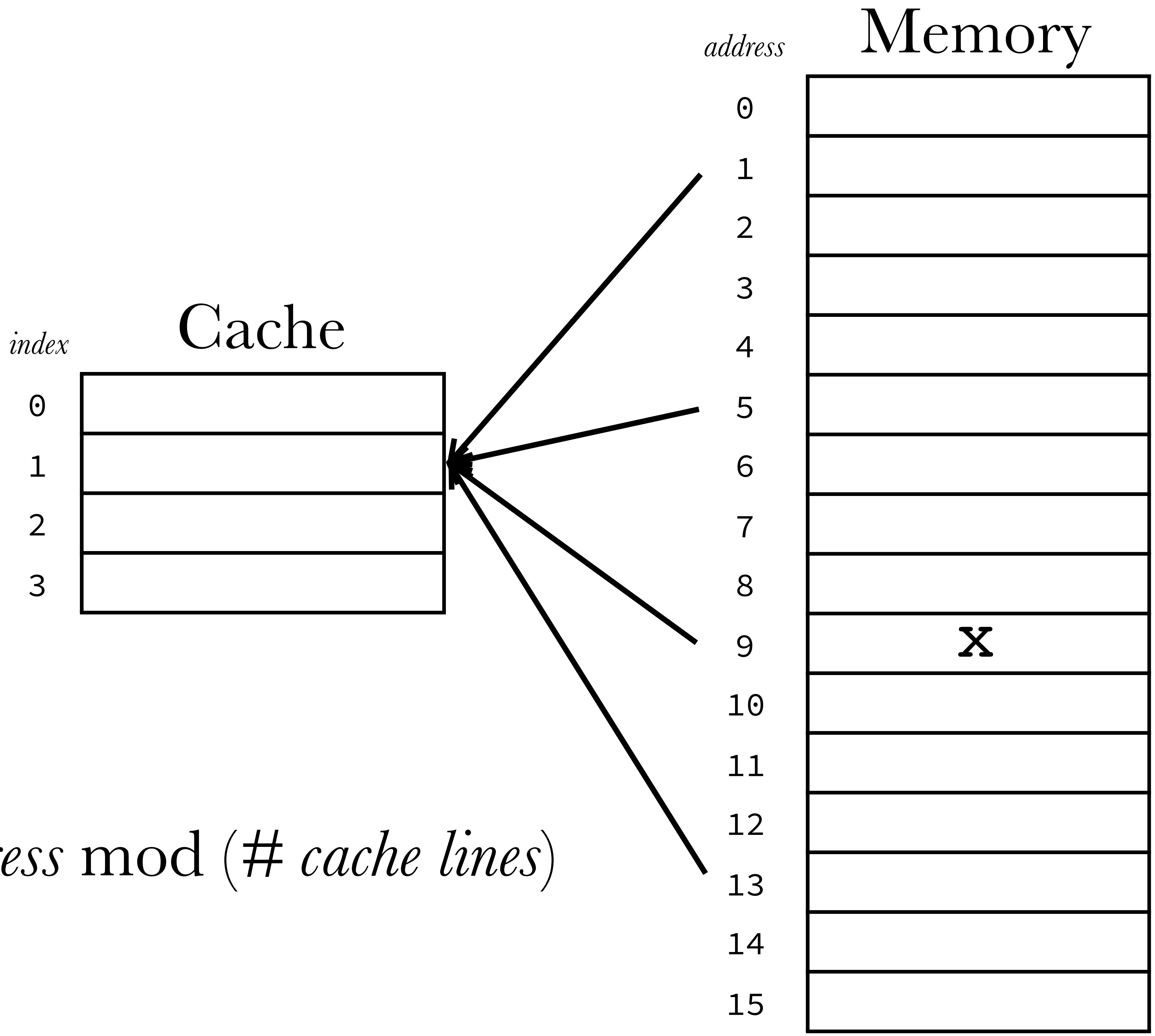

address
Memory
0
1
2
3
4
5
6
7
8
9
x
10
11
12
13
14
15
index
Cache
0
1
2
3
?


address
Memory
0
1
2
3
4
5
6
7
8
9
x
10
11
12
13
14
15
index
Cache
0
1
2
3

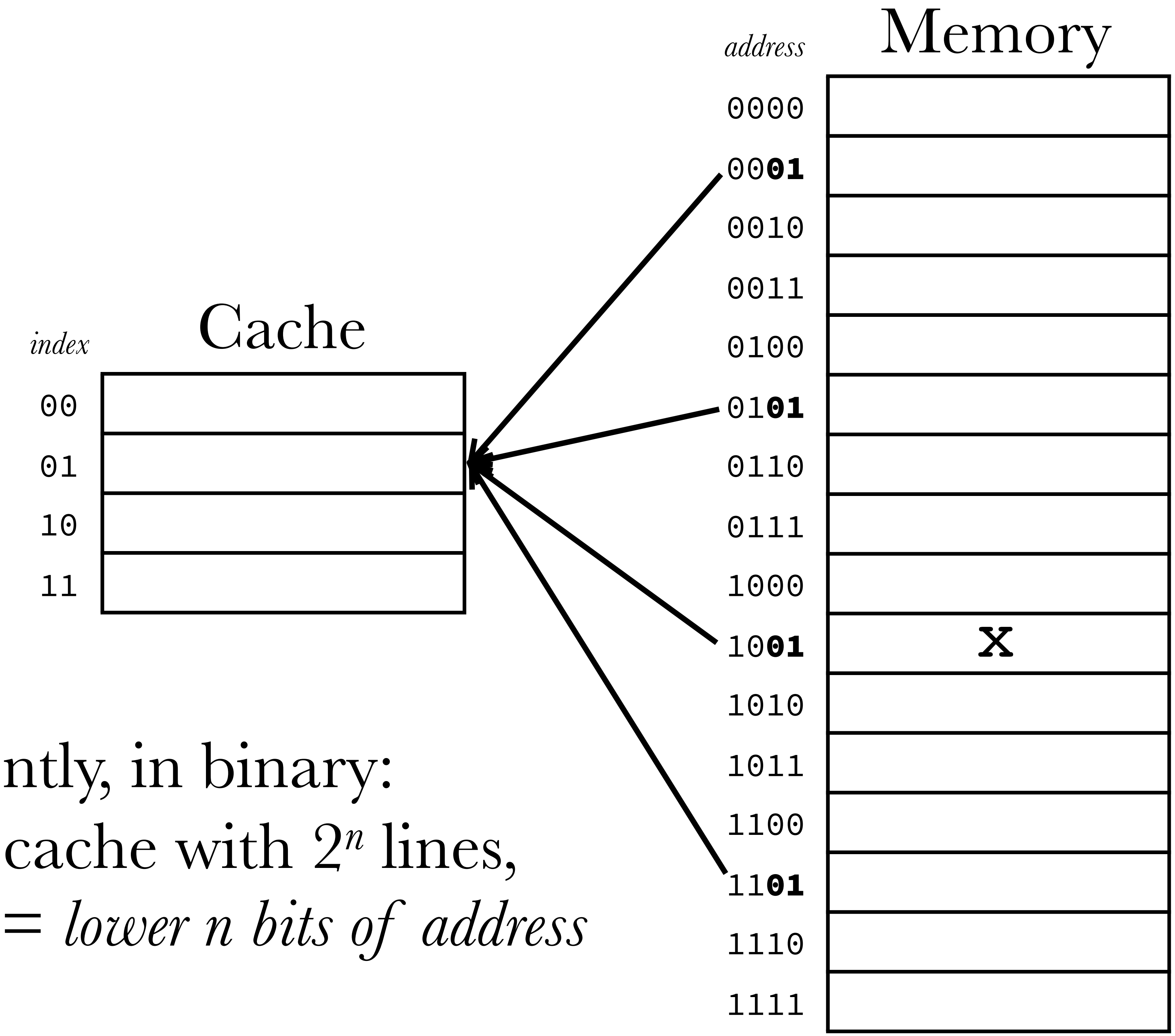

address
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
x
Cache
index
0
1
2
3



$$\textit{index} = \textit{address} \bmod (\#\ \textit{cache lines})$$



$$\textit{index} = \textit{address} \bmod (\#\ \textit{cache lines})$$

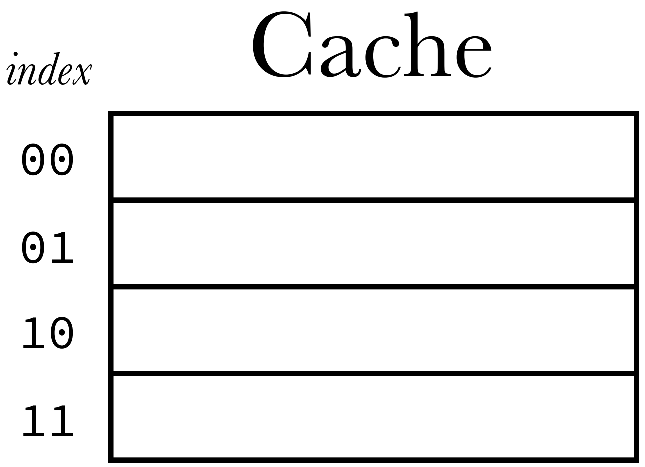


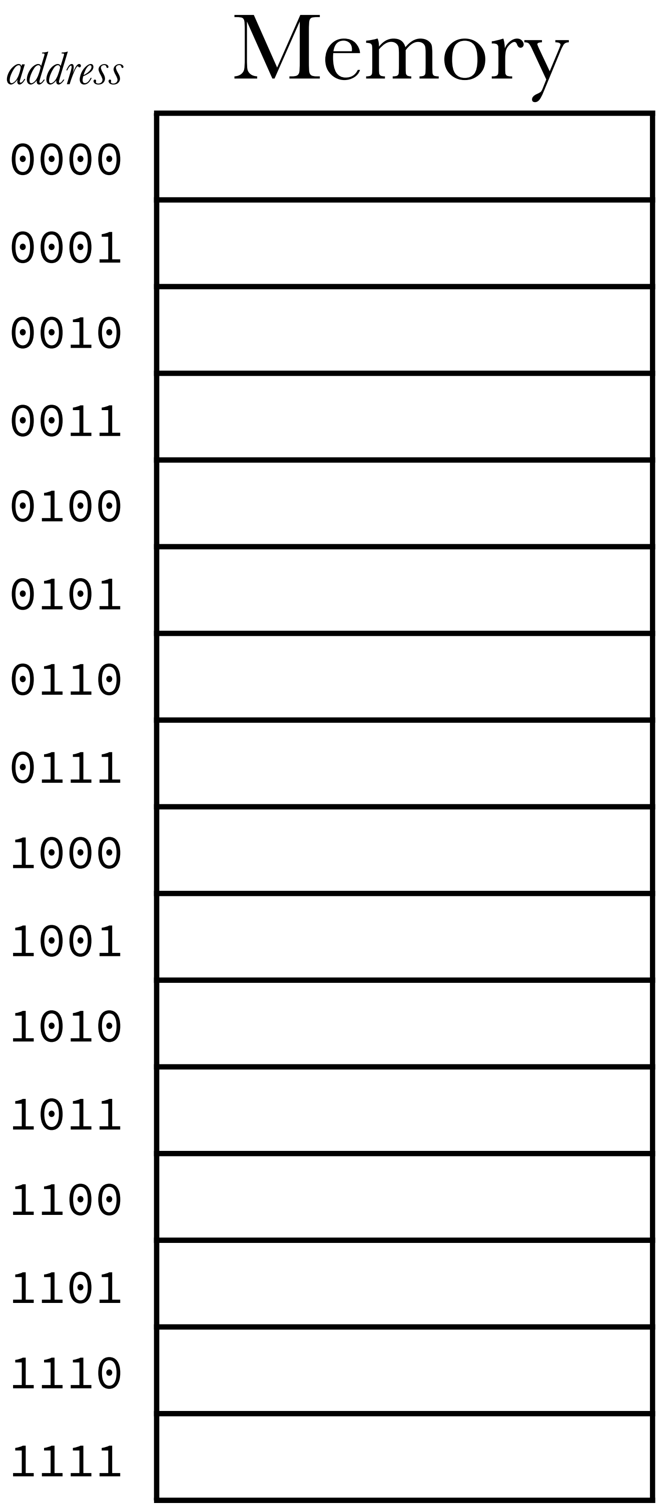
equivalently, in binary:

for a cache with 2^n lines,

index = lower n bits of address

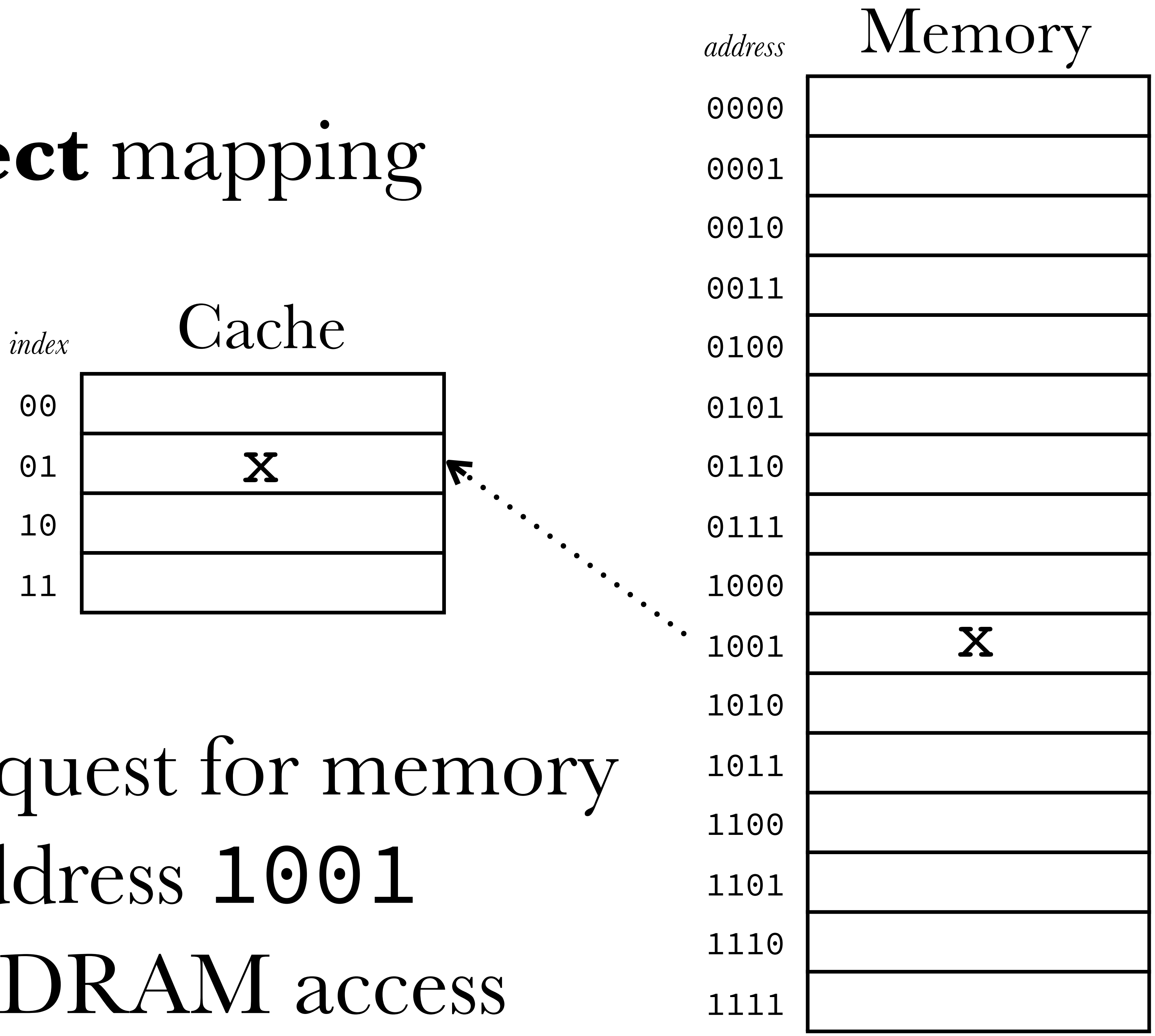
1) **direct** mapping





each address is mapped to a single, unique line in the cache

1) **direct** mapping



Cache

index	
`00`	
`01`	`x`
`10`	
`11`	

Memory

address	
`0000`	
`0001`	
`0010`	
`0011`	
`0100`	
`0101`	
`0110`	
`0111`	
`1000`	
`1001`	`x`
`1010`	
`1011`	
`1100`	
`1101`	
`1110`	
`1111`	

e.g., request for memory address `1001`
→ DRAM access

1) **direct** mapping

Cache

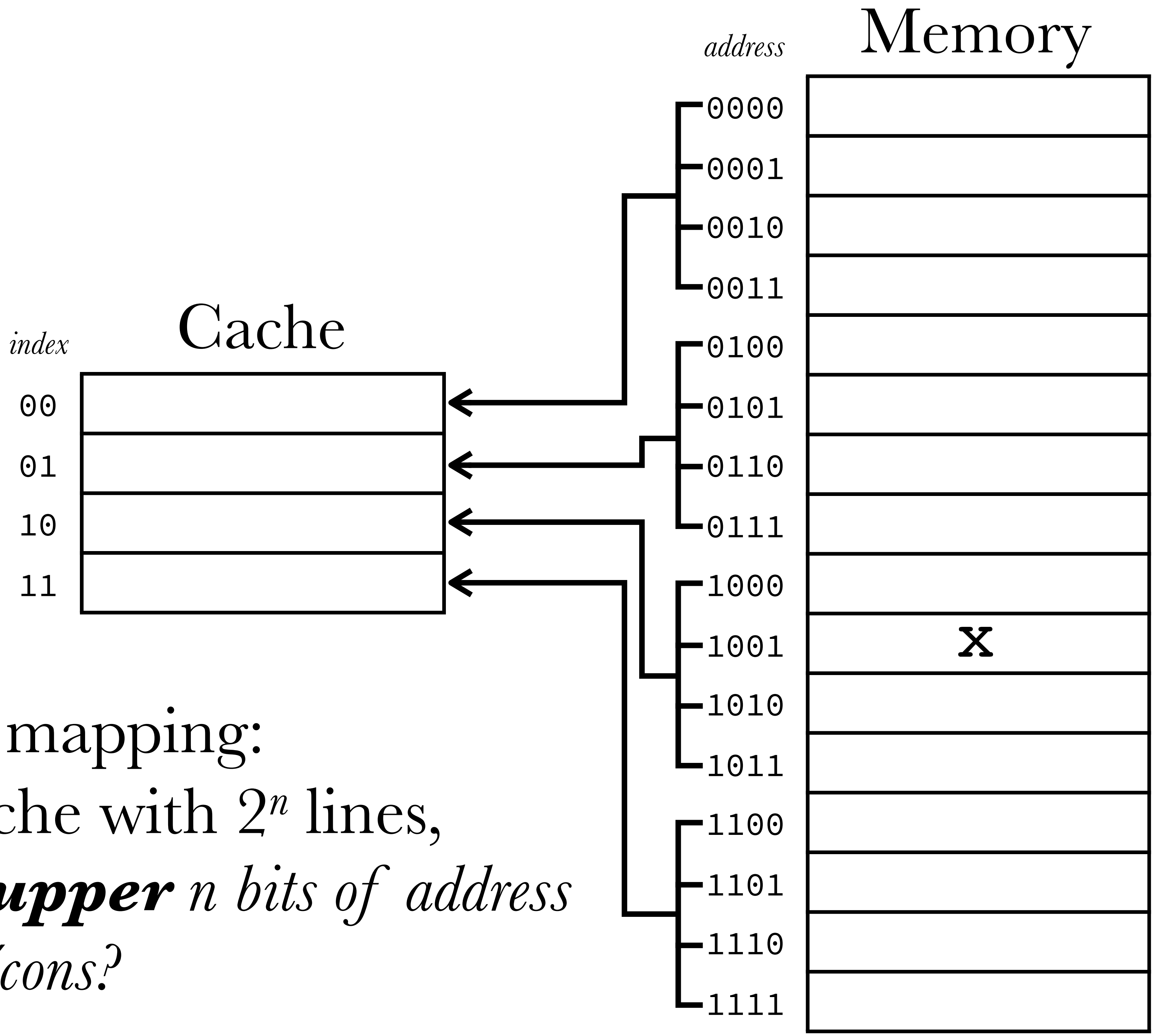
index	
00	
01	x
10	
11	

Memory

address	
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	x
1010	
1011	
1100	
1101	
1110	
1111	

e.g., repeated request for address `1001`

→ cache "hit"

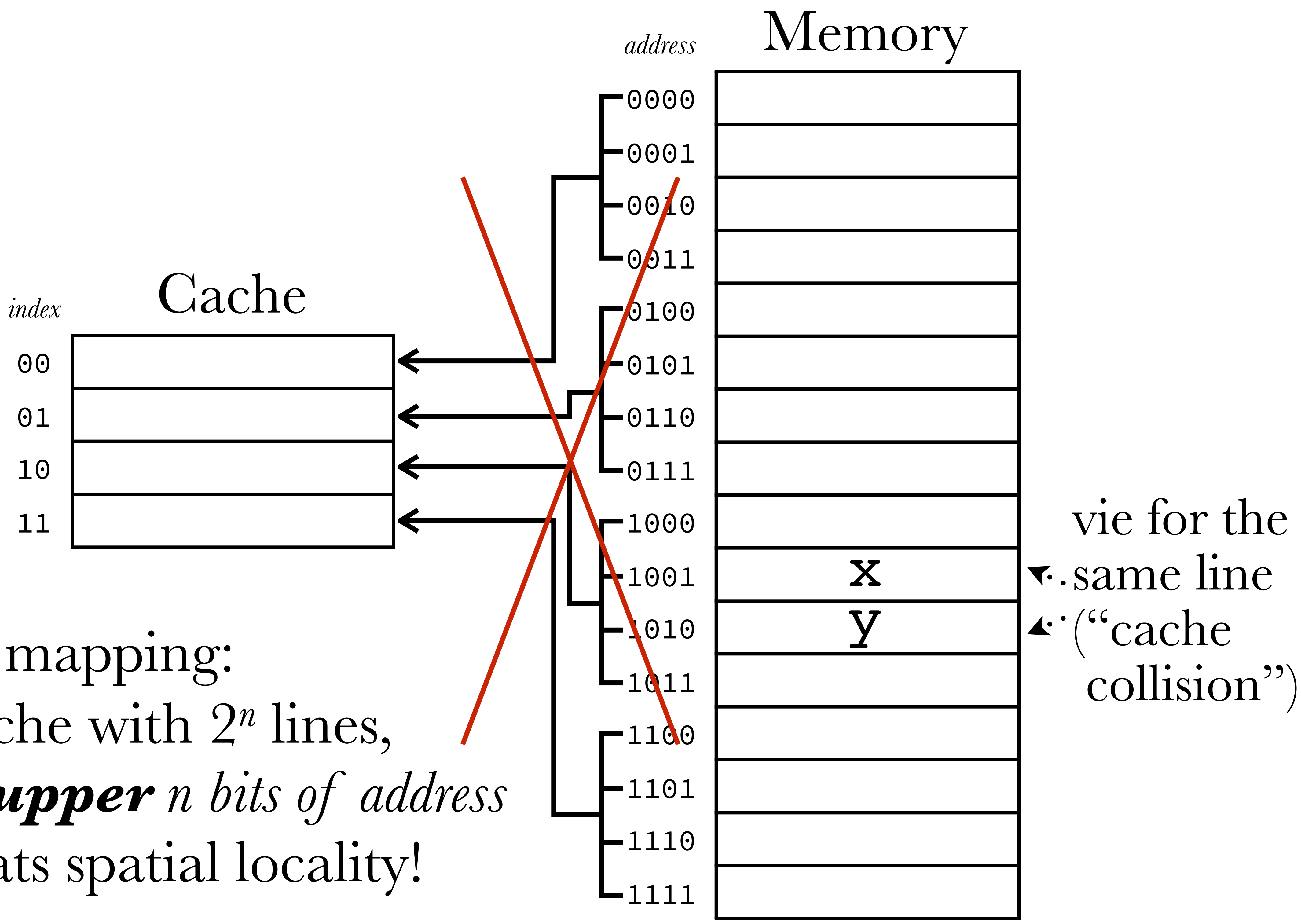


alternative mapping:

for a cache with 2^n lines,

index = ***upper*** *n bits of address*

— *pros/cons?*



alternative mapping:

for a cache with 2^n lines,

index = ***upper*** *n bits of address*

— defeats spatial locality!

1) **direct** mapping

index	Cache
00	
01	**x**
10	
11	

reverse mapping: where did **x** come from? (and is it valid data or garbage?)

address	Memory
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
00			
01			x
10			
11			

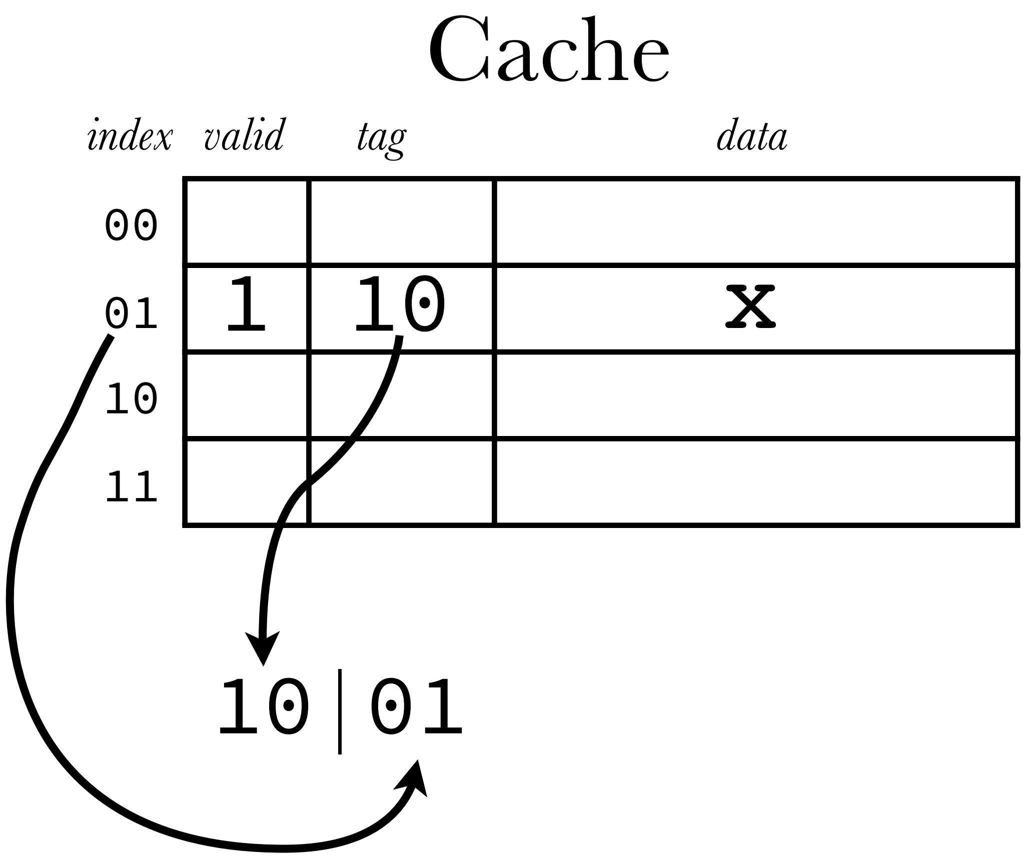
must add some fields

- *tag* field: top part of mapped address
- *valid bit*: is it valid?

Memory

address	Memory
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

1) **direct** mapping



Cache

index	*valid*	*tag*	*data*
00			
01	1	10	x
10			
11			

10|01

i.e., **x** "belongs to" address **1001**

Memory

address	
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
00	1	01	w
01	1	11	x
10	1	00	y
11	0	01	z

assuming memory & cache are in sync, "fill in" memory

Memory

address	
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
00	1	01	w
01	1	11	x
10	1	00	y
11	0	01	z

assuming memory
& cache are in sync,
“fill in” memory

address	Memory
0000	
0001	
0010	y
0011	
0100	w
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	x
1110	
1111	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
00	1	01	w
01	1	11	x
10	1	00	y
11	0	01	z

what if new request arrives for `1011`?

Memory

address	Memory
0000	
0001	
0010	y
0011	
0100	w
0101	
0110	
0111	
1000	
1001	
1010	
1011	a
1100	
1101	x
1110	
1111	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
`00`	`1`	`01`	`w`
`01`	`1`	`11`	`x`
`10`	`1`	`00`	`y`
`11`	**`1`**	**`10`**	**`a`**

what if new request arrives for `1011`?

- *cache "miss": fetch* `a`

Memory

address	Memory
`0000`	
`0001`	
`0010`	`y`
`0011`	
`0100`	`w`
`0101`	
`0110`	
`0111`	
`1000`	
`1001`	
`1010`	
`1011`	`a`
`1100`	
`1101`	`x`
`1110`	
`1111`	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
00	1	01	w
01	1	11	x
10	1	00	y
11	1	10	a

what if new request arrives for `0010`?

address	Memory
0000	
0001	
0010	y
0011	
0100	w
0101	
0110	
0111	
1000	
1001	
1010	
1011	a
1100	
1101	x
1110	
1111	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
`00`	`1`	`01`	`w`
`01`	`1`	`11`	`x`
`10`	`1`	`00`	`y`
`11`	`1`	`10`	`a`

Memory

address	
`0000`	
`0001`	
`0010`	`y`
`0011`	
`0100`	`w`
`0101`	
`0110`	
`0111`	
`1000`	
`1001`	
`1010`	
`1011`	`a`
`1100`	
`1101`	`x`
`1110`	
`1111`	

what if new request arrives for `0010`?

- *cache "hit"*; just return **`y`**

1) **direct** mapping

Cache

index	valid	tag	data
00	1	01	w
01	1	11	x
10	1	00	y
11	1	10	a

what if new request arrives for `1000`?

Memory

address	
0000	
0001	
0010	y
0011	
0100	w
0101	
0110	
0111	
1000	b
1001	
1010	
1011	a
1100	
1101	x
1110	
1111	

1) **direct** mapping

Cache

index	*valid*	*tag*	*data*
00	1	**10**	**b**
01	1	11	x
10	1	00	y
11	1	10	a

what if new request arrives for `1000`?

- *evict* old mapping to make room for new

Memory

address	Memory
0000	
0001	
0010	y
0011	
0100	w
0101	
0110	
0111	
1000	b
1001	
1010	
1011	a
1100	
1101	x
1110	
1111	

1) **direct** mapping

- implicit *replacement policy* — always keep most recently accessed data for a given cache line
- motivated by temporal locality

Requests

address	*hit/miss?*
0x89	
0xAB	
0x60	
0xAB	
0x83	
0x67	
0xAB	
0x12	

Initial Cache

index	*valid*	*tag*
000	0	00101
001	0	10010
010	0	00010
011	1	10101
100	1	00000
101	0	10011
110	1	11110
111	1	11001

Given initial contents of a *direct-mapped* cache, determine if each request is a *hit* or *miss*. Also, show the final cache.

Problem: our cache (so far) implicitly deals with *single bytes* of data at a time

```
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact *= n;
        n -= 1;
    }
}
```

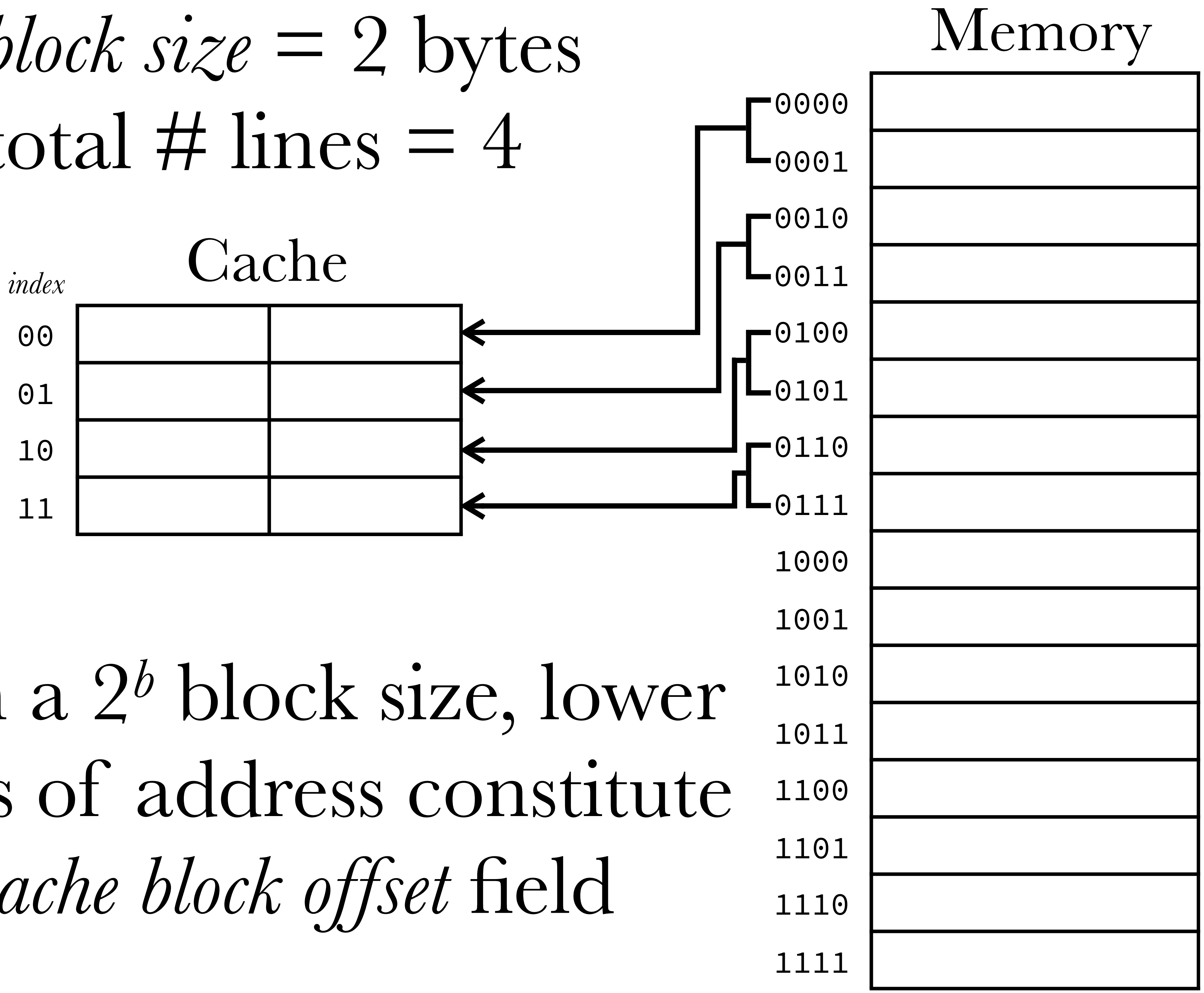
But we frequently deal with > 1 byte of data at a time (e.g., words)

Solution: adjust minimum granularity of memory ⇔ cache mapping

Use a "cache *block*" of 2^b bytes

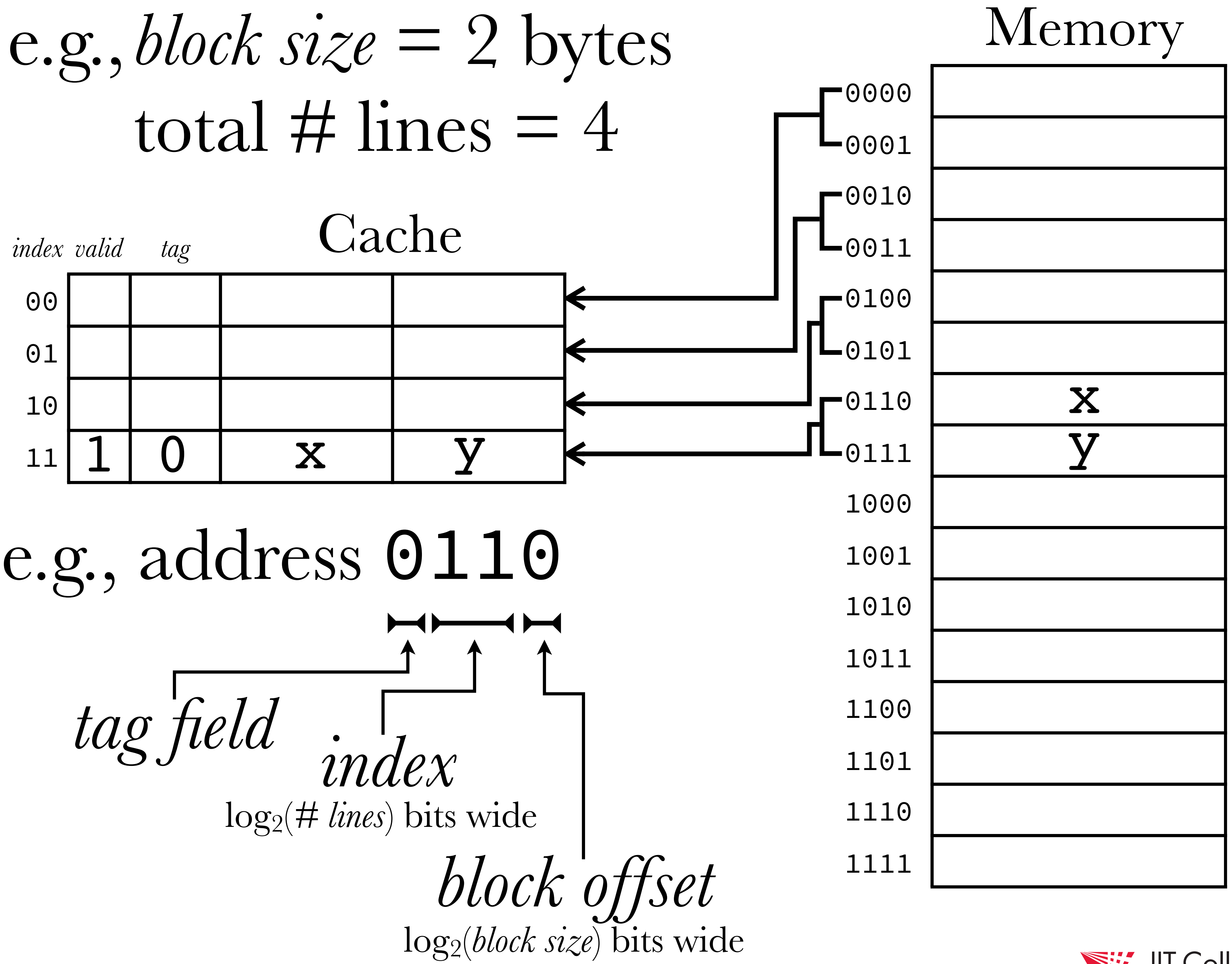
† memory remains byte-addressable!

e.g., *block size* = 2 bytes
total # lines = 4



With a 2^b block size, lower b bits of address constitute the *cache block offset* field e.g., *block size* = 2 bytes

total # lines = 4



Cache

index	*valid*	*tag*		
00				
01				
10				
11	1	0	x	y

Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	x
0111	y
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

e.g., address `0110`

tag field

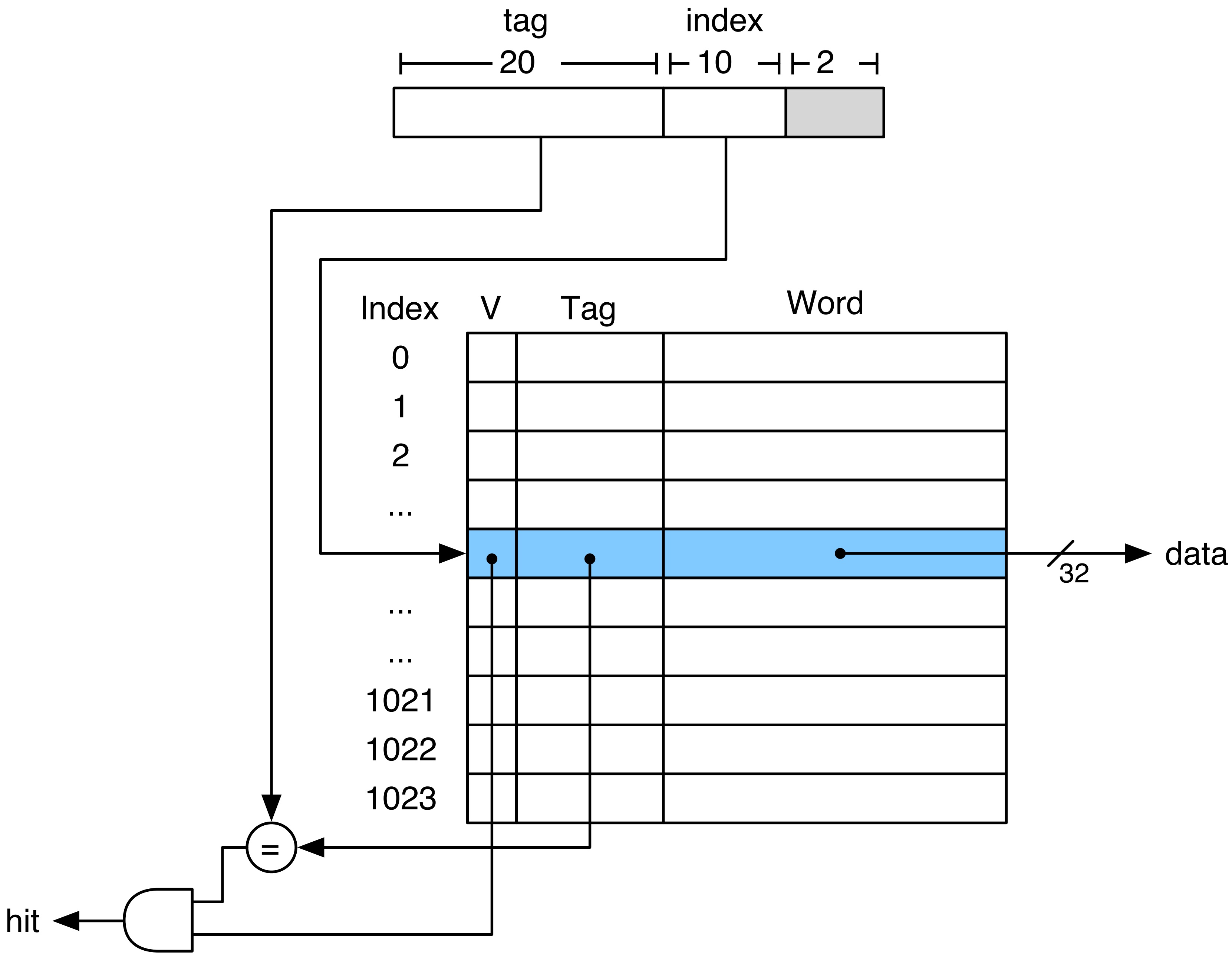
index

log$_2$(# *lines*) bits wide

block offset

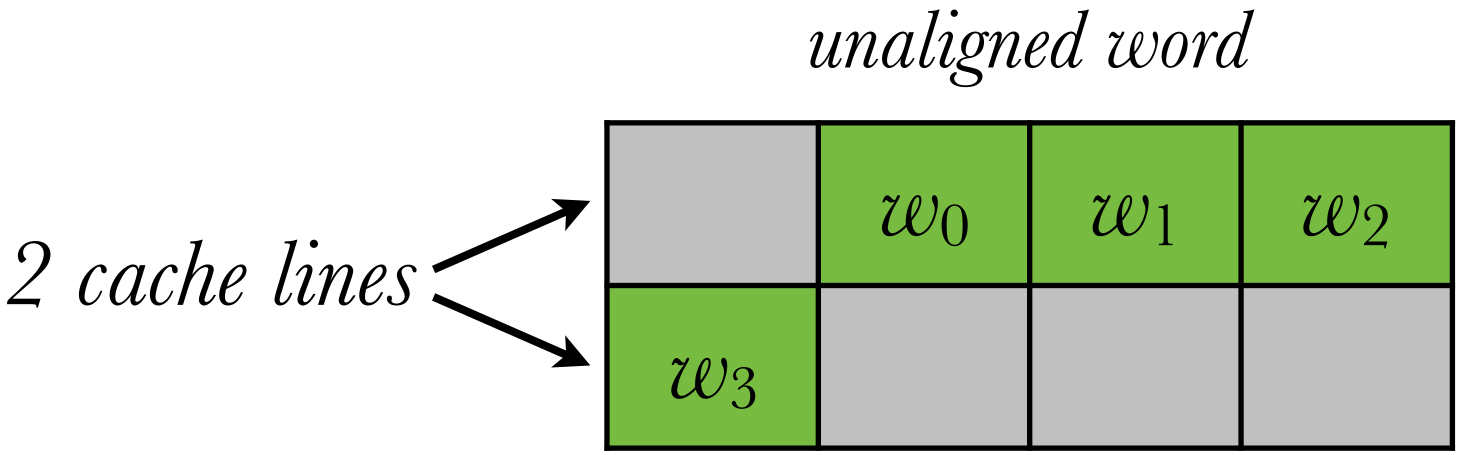
log$_2$(*block size*) bits wide

e.g., cache with 2^{10} lines of 4-byte blocks



note: words in memory should be *aligned*; i.e., they start at addresses that are
multiples of the word size

otherwise, must fetch > 1 word-sized block to access a single word!



```c
struct foo {
    char c;
    int i;
    char buf[10];
    long l;
};

struct foo f = { 'a', 0xDEADBEEF, "abcdefghi", 0x123456789DEFACED };

main() {
    printf("%d %d %d\n", sizeof(int), sizeof(long), sizeof(struct foo));
}
```

```
$ ./a.out
4 8 32

$ objdump -s -j .data a.out
a.out:     file format elf64-x86-64
Contents of section .data:
 61000000 efbeadde 61626364 65666768  a.......abcdefgh
 69000000 00000000 edacef9d 78563412  i...........xV4.
```

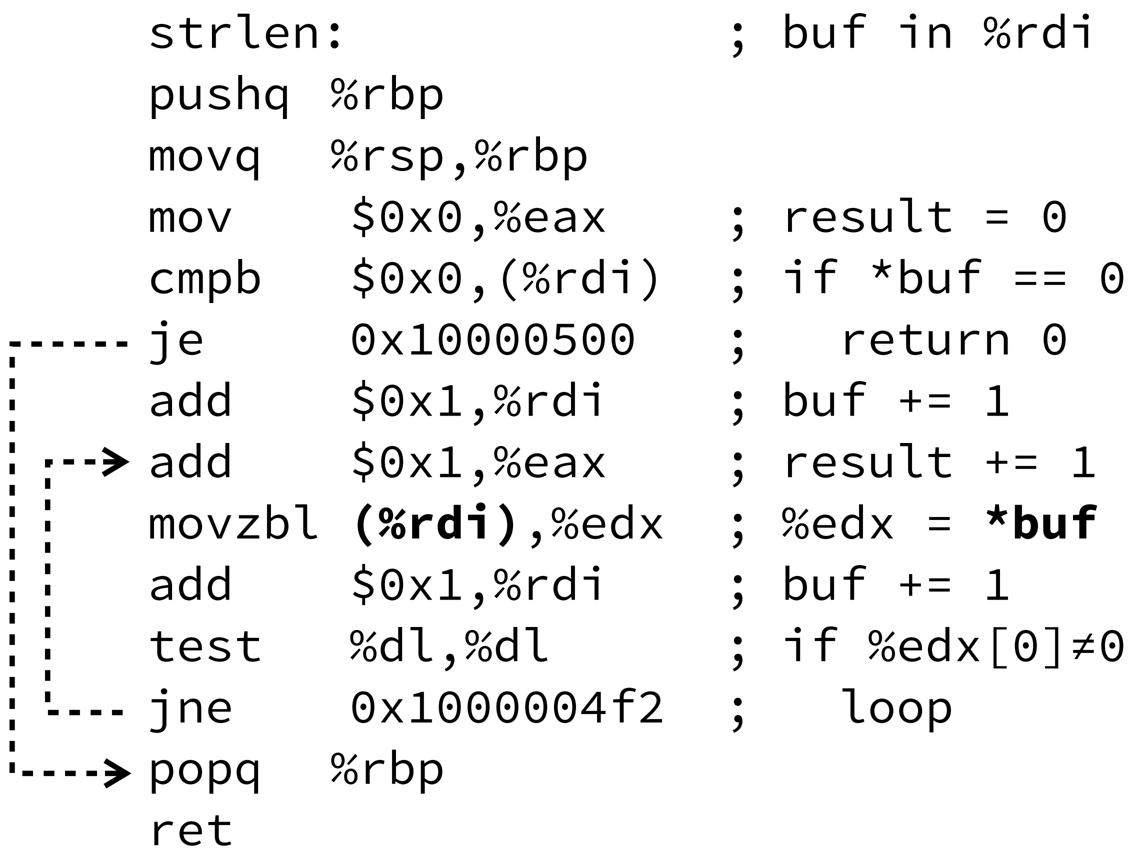
(i.e., C auto-aligns structure components)

```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
strlen:                     ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax           ; result = 0
cmpb    $0x0,(%rdi)         ; if *buf == 0
je      0x10000500          ;   return 0
add     $0x1,%rdi           ; buf += 1
add     $0x1,%eax           ; result += 1
movzbl  (%rdi),%edx         ; %edx = *buf
add     $0x1,%rdi           ; buf += 1
test    %dl,%dl             ; if %edx[0]≠0
jne     0x1000004f2         ;   loop
popq   %rbp
ret
```

Given: *direct-mapped* cache with *4-byte blocks*. Determine the average *hit rate* of `strlen` (i.e., the *fraction of cache hits* to total requests)

```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```



```
strlen:                    ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax          ; result = 0
cmpb    $0x0,(%rdi)        ; if *buf == 0
je      0x10000500         ;   return 0
add     $0x1,%rdi          ; buf += 1
add     $0x1,%eax          ; result += 1
movzbl (%rdi),%edx         ; %edx = *buf
add     $0x1,%rdi          ; buf += 1
test    %dl,%dl            ; if %edx[0]≠0
jne     0x1000004f2        ;   loop
popq   %rbp
ret
```

Assumptions:

- ignore code caching (in separate cache)
- `buf` contents are not initially cached

```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
strlen:                     ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax         ; result = 0
cmpb    $0x0,(%rdi)       ; if *buf == 0
je      0x10000500        ;   return 0
add     $0x1,%rdi         ; buf += 1
add     $0x1,%eax         ; result += 1
movzbl (%rdi),%edx        ; %edx = *buf
add     $0x1,%rdi         ; buf += 1
test    %dl,%dl           ; if %edx[0]≠0
jne     0x1000004f2       ;   loop
popq   %rbp
ret
```

strlen(\0)

strlen(a \0)

strlen(a b c d e \0)

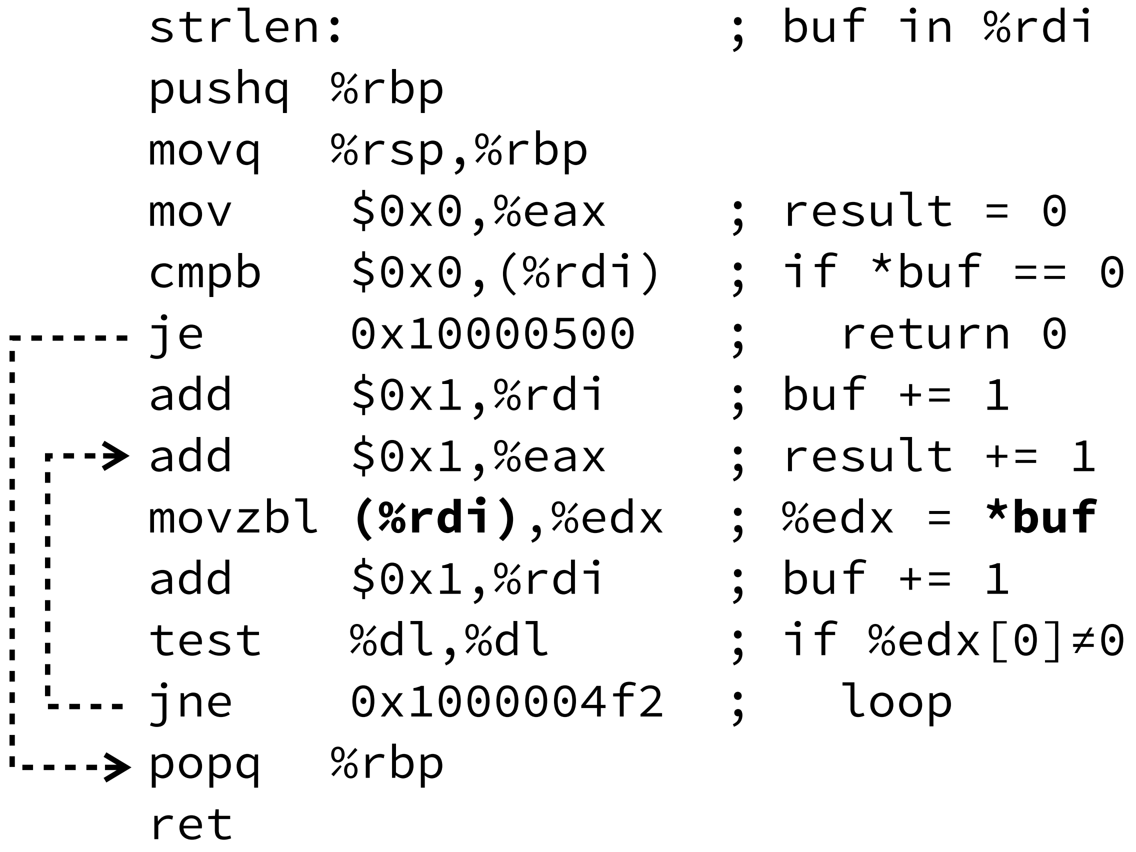
strlen(a b c d e f g h i j k l ...)

```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
strlen:                    ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax          ; result = 0
cmpb    $0x0,(%rdi)        ; if *buf == 0
je      0x10000500         ;   return 0
add     $0x1,%rdi          ; buf += 1
add     $0x1,%eax          ; result += 1
movzbl  (%rdi),%edx        ; %edx = *buf
add     $0x1,%rdi          ; buf += 1
test    %dl,%dl            ; if %edx[0]≠0
jne     0x1000004f2        ;   loop
popq   %rbp
ret
```

```
strlen( \0 )
strlen( a | \0 )
strlen( a | b | c | d | e | \0 )
strlen( a | b | c | d | e | f | g | h | i | j | k | l | ... )
```

```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```



```
strlen:                      ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax          ; result = 0
cmpb    $0x0,(%rdi)        ; if *buf == 0
je      0x10000500         ;   return 0
add     $0x1,%rdi          ; buf += 1
add     $0x1,%eax          ; result += 1
movzbl  (%rdi),%edx        ; %edx = *buf
add     $0x1,%rdi          ; buf += 1
test    %dl,%dl            ; if %edx[0]≠0
jne     0x1000004f2        ;   loop
popq   %rbp
ret
```

strlen(\0)

strlen(a | \0) *or, if unlucky:* a | \0

strlen(a | b | c | d | e | \0)

strlen(a | b | c | d | e | f | g | h | i | j | k | l | ...)

```
strlen:                      ; buf in %rdi
        pushq  %rbp
        movq   %rsp,%rbp
        mov     $0x0,%eax    ; result = 0
        cmpb    $0x0,(%rdi)  ; if *buf == 0
 ------ je      0x10000500   ;   return 0
|       add     $0x1,%rdi    ; buf += 1
|  ---> add     $0x1,%eax    ; result += 1
| |     movzbl (%rdi),%edx   ; %edx = *buf
| |     add     $0x1,%rdi    ; buf += 1
| |     test    %dl,%dl      ; if %edx[0]≠0
|  ---- jne     0x1000004f2  ;   loop
 -----> popq   %rbp
        ret
```

```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

`strlen(` `\0` `)`

`strlen(` `a` `\0` `)` *or, if unlucky:* `a` `\0`

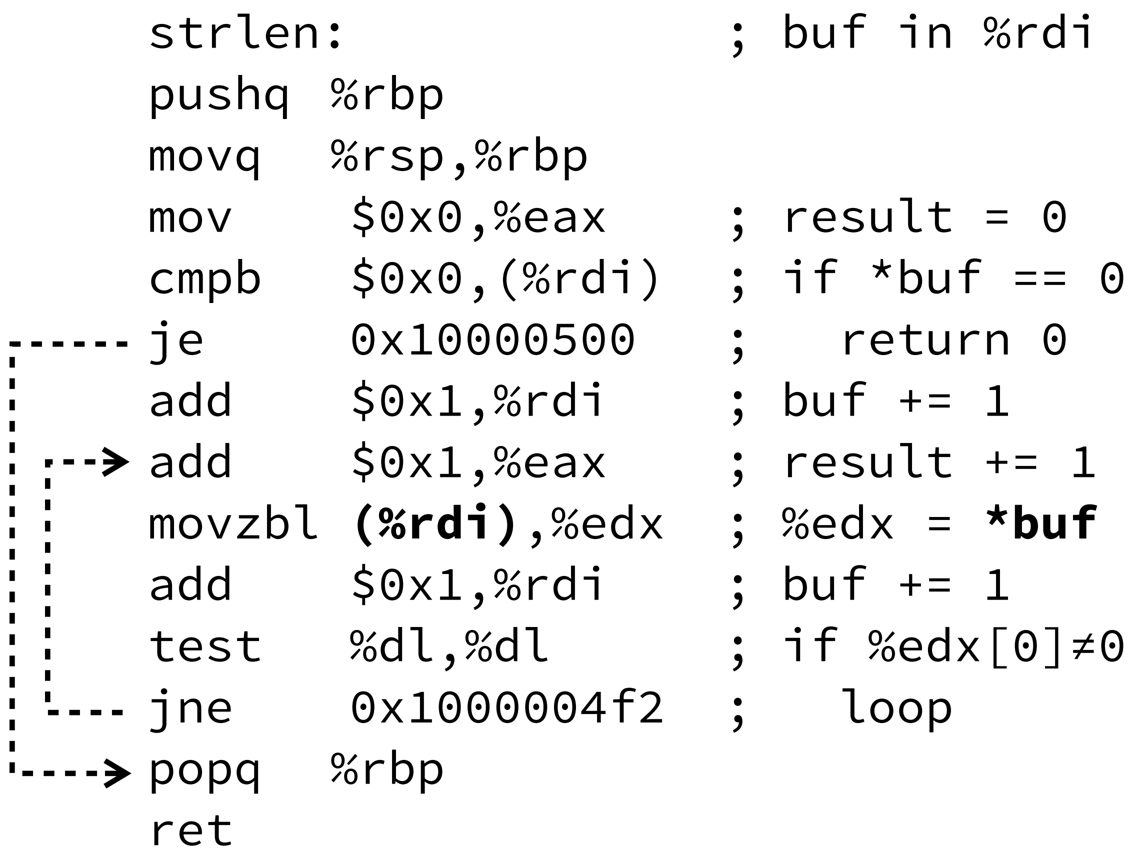
— simplifying assumption: first byte of `buf` is aligned

```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

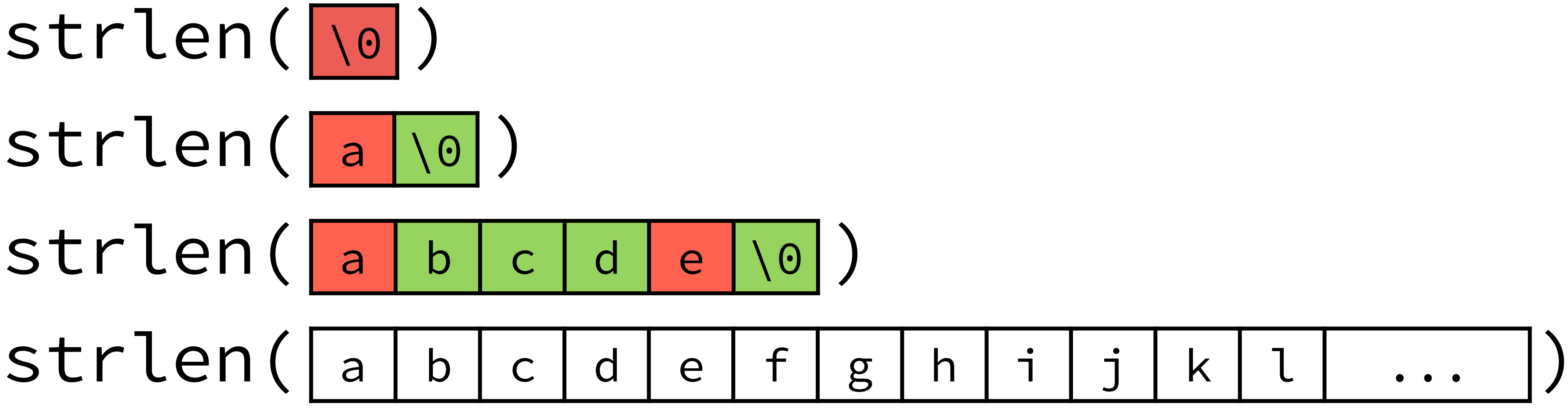
```
strlen:                    ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax          ; result = 0
cmpb    $0x0,(%rdi)        ; if *buf == 0
je      0x10000500         ;   return 0
add     $0x1,%rdi          ; buf += 1
add     $0x1,%eax          ; result += 1
movzbl  (%rdi),%edx        ; %edx = *buf
add     $0x1,%rdi          ; buf += 1
test    %dl,%dl            ; if %edx[0]≠0
jne     0x1000004f2        ;   loop
popq   %rbp
ret
```

```
strlen(|\0|)
strlen(|a|\0|)
strlen(|a|b|c|d|e|\0|)
strlen(|a|b|c|d|e|f|g|h|i|j|k|l|...|)
```

```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```



```
strlen:                  ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax        ; result = 0
cmpb    $0x0,(%rdi)      ; if *buf == 0
je      0x10000500       ;   return 0
add     $0x1,%rdi        ; buf += 1
add     $0x1,%eax        ; result += 1
movzbl (%rdi),%edx       ; %edx = *buf
add     $0x1,%rdi        ; buf += 1
test    %dl,%dl          ; if %edx[0]≠0
jne     0x1000004f2      ;   loop
popq   %rbp
ret
```



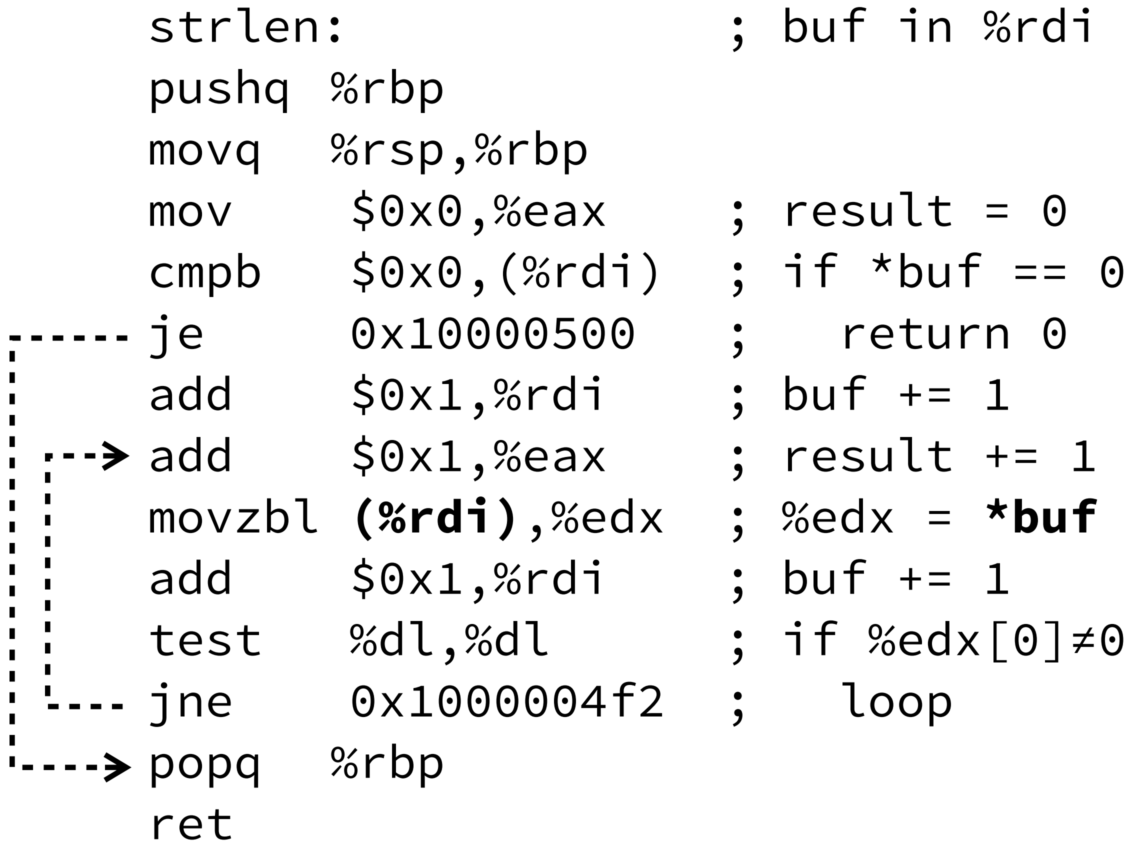
strlen(\0)

strlen(a | \0)

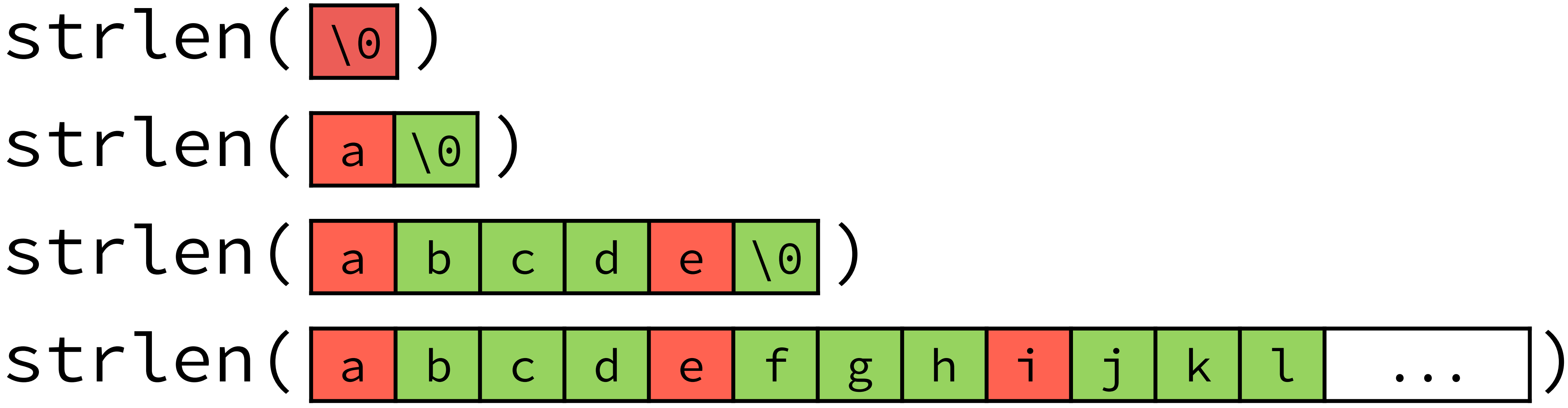
strlen(a | b | c | d | e | \0)

strlen(a | b | c | d | e | f | g | h | i | j | k | l | ...)

```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```



```
strlen:                       ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax             ; result = 0
cmpb    $0x0,(%rdi)           ; if *buf == 0
je      0x10000500            ;   return 0
add     $0x1,%rdi             ; buf += 1
add     $0x1,%eax             ; result += 1
movzbl  (%rdi),%edx           ; %edx = *buf
add     $0x1,%rdi             ; buf += 1
test    %dl,%dl               ; if %edx[0]≠0
jne     0x1000004f2           ;   loop
popq   %rbp
ret
```



strlen(\0)

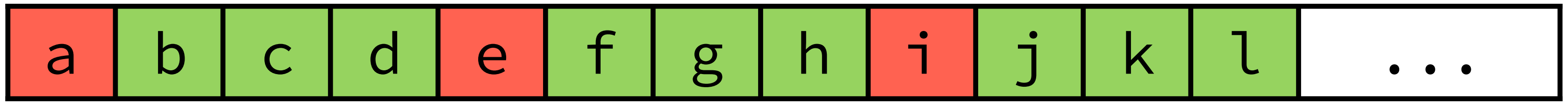
strlen(a \0)

strlen(a b c d e \0)

strlen(a b c d e f g h i j k l ...)

```c
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
strlen:                      ; buf in %rdi
pushq  %rbp
movq   %rsp,%rbp
mov     $0x0,%eax            ; result = 0
cmpb    $0x0,(%rdi)          ; if *buf == 0
je      0x10000500           ;   return 0
add     $0x1,%rdi            ; buf += 1
add     $0x1,%eax            ; result += 1
movzbl  (%rdi),%edx          ; %edx = *buf
add     $0x1,%rdi            ; buf += 1
test    %dl,%dl              ; if %edx[0]≠0
jne     0x1000004f2          ;   loop
popq   %rbp
ret
```



strlen(a b c d e f g h i j k l ...)

In the long run, hit rate = ¾ = 75%

```c
int sum(int *arr, int n) {
    int i, r = 0;
    for (i=0; i<n; i++)
        r += arr[i];
    return r;
}
```

```
sum:                                ; arr,n in %rdi,%rsi
pushq   %rbp
movq    %rsp,%rbp
mov     $0x0,%eax                   ; r = 0
test    %esi,%esi                   ; if n == 0
jle     0x10000527                  ;   return 0
sub     $0x1,%esi                   ; n -= 1
lea     0x4(,%rsi,4),%rcx           ; %rcx = 4*n+4
mov     $0x0,%edx                   ; %rdx = 0
add     (%rdi,%rdx,1),%eax          ; r += arr[%rdx]
add     $0x4,%rdx                   ; %rdx += 4
cmp     %rcx,%rdx                   ; if %rcx == %rdx
jne     0x1000051b                  ;   return r
popq    %rbp
ret
```

Again: *direct-mapped* cache with *4-byte blocks*.
Average *hit rate* of `sum`? (`arr` not cached)

```c
int sum(int *arr, int n) {
    int i, r = 0;
    for (i=0; i<n; i++)
        r += arr[i];
    return r;
}
```

```
sum:                                   ; arr,n in %rdi,%rsi
pushq   %rbp
movq    %rsp,%rbp
mov     $0x0,%eax                      ; r = 0
test    %esi,%esi                      ; if n == 0
jle     0x10000527                     ;   return 0
sub     $0x1,%esi                      ; n -= 1
lea     0x4(,%rsi,4),%rcx              ; %rcx = 4*n+4
mov     $0x0,%edx                      ; %rdx = 0
add     (%rdi,%rdx,1),%eax             ; r += arr[%rdx]
add     $0x4,%rdx                      ; %rdx += 4
cmp     %rcx,%rdx                      ; if %rcx == %rdx
jne     0x1000051b                     ;   return r
popq    %rbp
ret
```

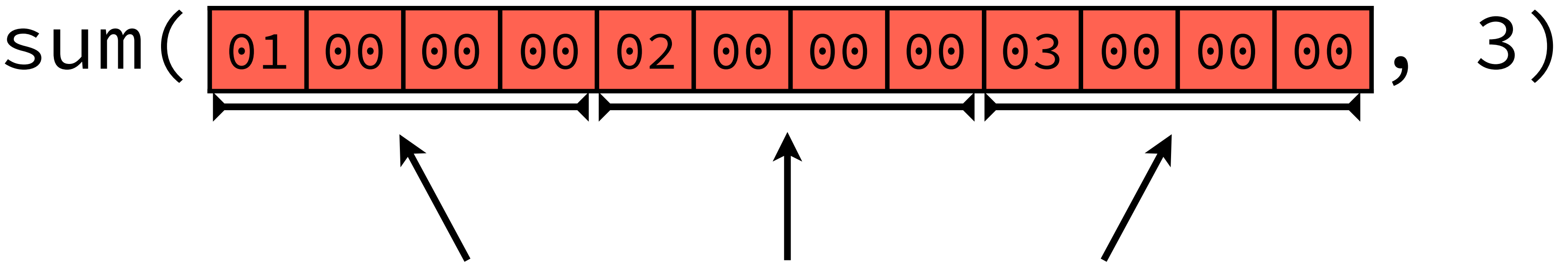
sum(

01	00	00	00	02	00	00	00	03	00	00	00

, 3)

```c
int sum(int *arr, int n) {
    int i, r = 0;
    for (i=0; i<n; i++)
        r += arr[i];
    return r;
}
```

```
sum:                                ; arr,n in %rdi,%rsi
pushq   %rbp
movq    %rsp,%rbp
mov     $0x0,%eax                   ; r = 0
test    %esi,%esi                   ; if n == 0
jle     0x10000527                  ;   return 0
sub     $0x1,%esi                   ; n -= 1
lea     0x4(,%rsi,4),%rcx           ; %rcx = 4*n+4
mov     $0x0,%edx                   ; %rdx = 0
add     (%rdi,%rdx,1),%eax          ; r += arr[%rdx]
add     $0x4,%rdx                   ; %rdx += 4
cmp     %rcx,%rdx                   ; if %rcx == %rdx
jne     0x1000051b                  ;   return r
popq    %rbp
ret
```

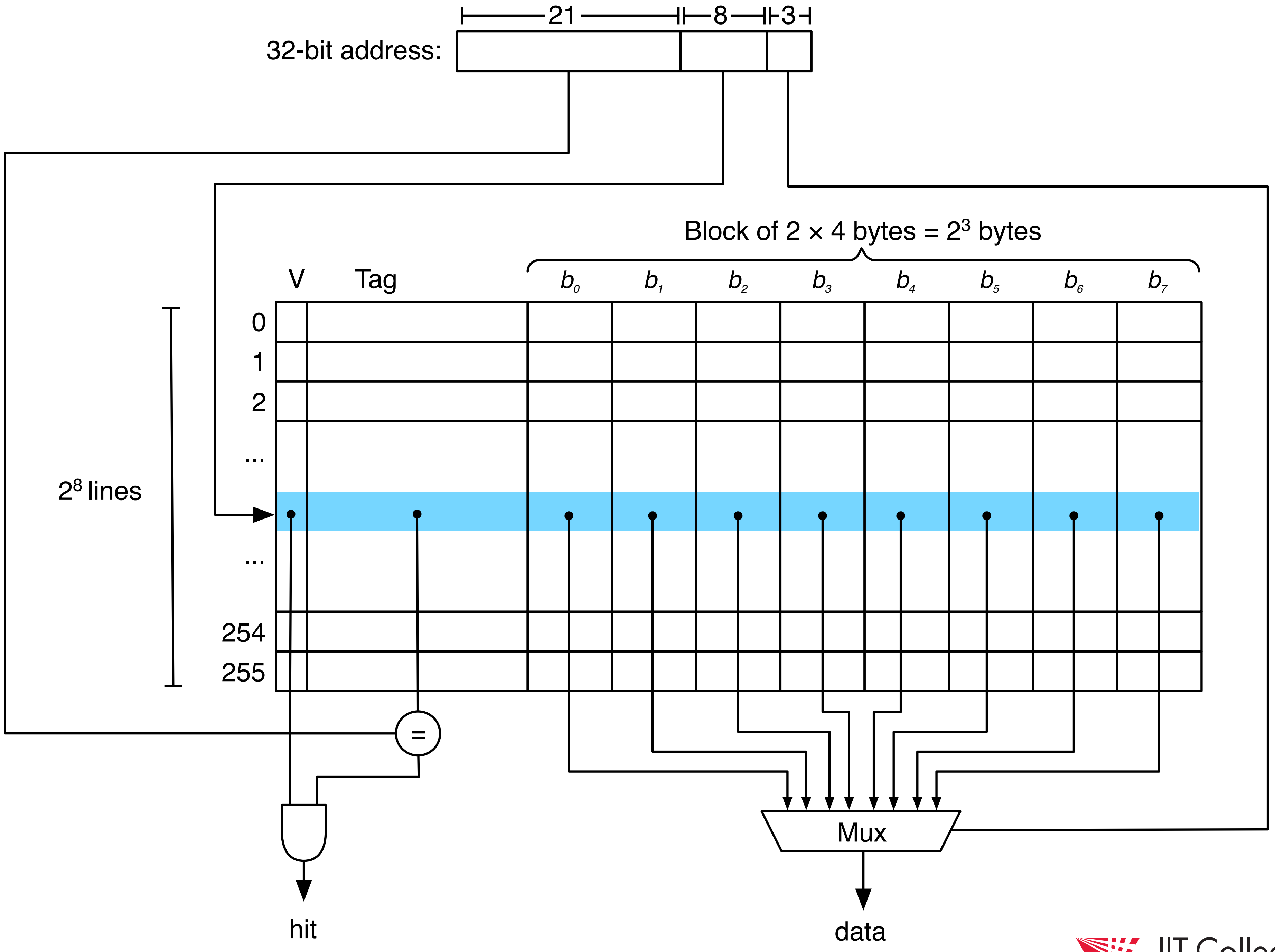


```
sum( 01 00 00 00 02 00 00 00 03 00 00 00 , 3)
```

each *block* is a miss! (hit rate=0%)

use *multi-word* blocks to help with larger array strides (e.g., for word-sized data)

e.g., cache with 2^8 lines of 2 × 4 byte blocks



Cache										
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0	173	1	05	E2	6C	05	3B	53	0C	8E
1	2FB	1	9B	26	58	E0	EB	05	4A	4C
2	316	0	F8	3E	29	92	B2	52	B9	2E
3	03A	1	95	07	51	3F	7B	00	DA	AC
4	1B9	0	9A	AB	9E	E3	20	03	C0	06
5	2C2	1	FB	7C	EC	25	C8	2B	3E	D6
6	315	1	E0	05	FB	E8	72	79	BE	D4
7	2C7	1	45	2D	92	74	C8	CB	92	85

Are the following (byte) requests hits?
If so, what data is returned by the cache?

1. `0x0E9C`
2. `0xBEF0`

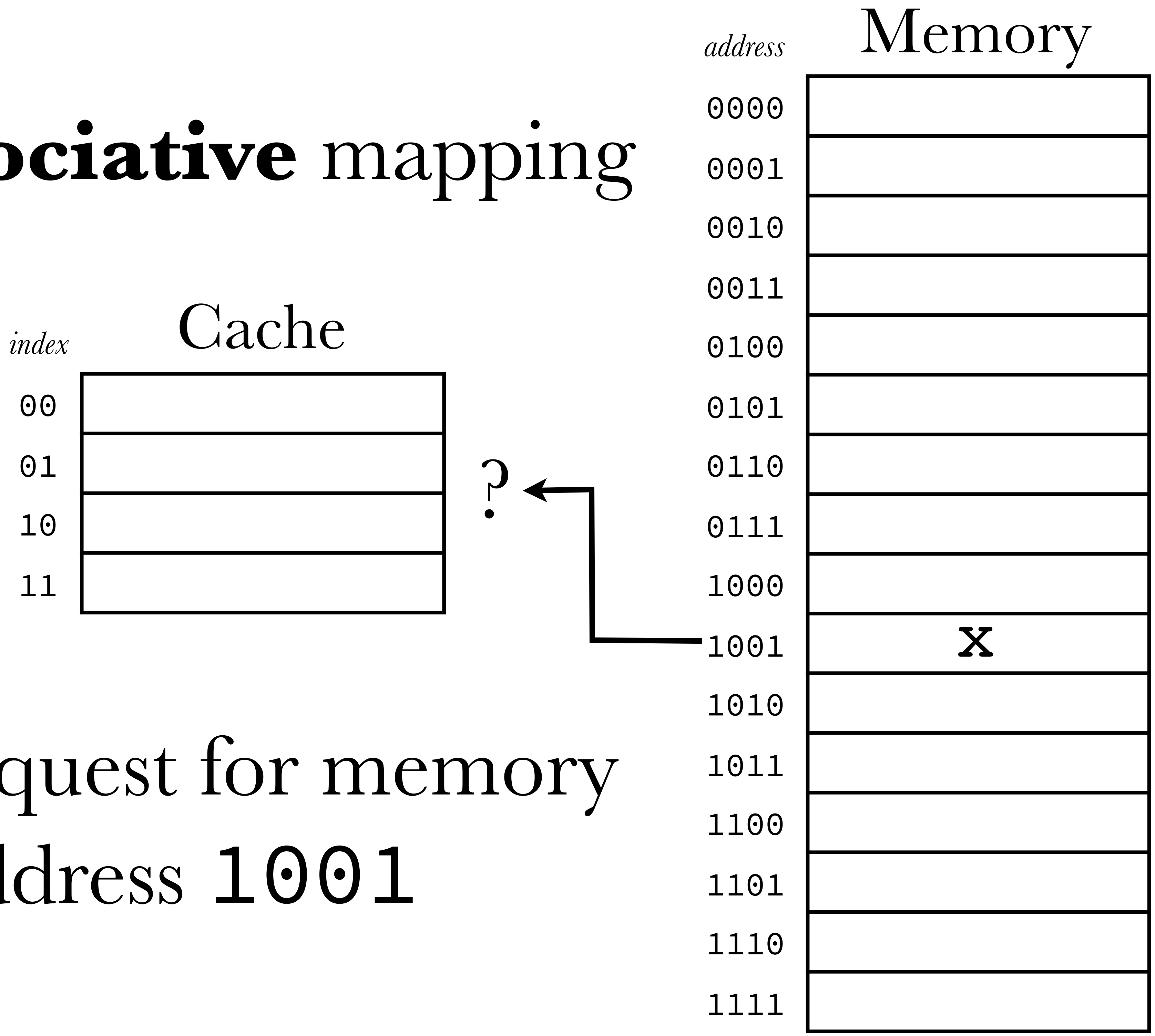
Cache										
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0	173	1	05	E2	6C	05	3B	53	0C	8E
1	2FB	1	9B	26	58	E0	EB	05	4A	4C
2	316	0	F8	3E	29	92	B2	52	B9	2E
3	03A	1	95	07	51	3F	7B	00	DA	AC
4	1B9	0	9A	AB	9E	E3	20	03	C0	06
5	2C2	1	FB	7C	EC	25	C8	2B	3E	D6
6	315	1	E0	05	FB	E8	72	79	BE	D4
7	2C7	1	45	2D	92	74	C8	CB	92	85

What happens when we receive the following sequence of requests?

- `0x9697A`, `0x3A478`, `0x34839`, `0x3A478`, `0x9697B`, `0x3483A` problem: when a *cache collision* occurs, we must evict the old (direct) mapping

— no way to use a different cache slot

2) **associative** mapping



index Cache

00
01
10
11

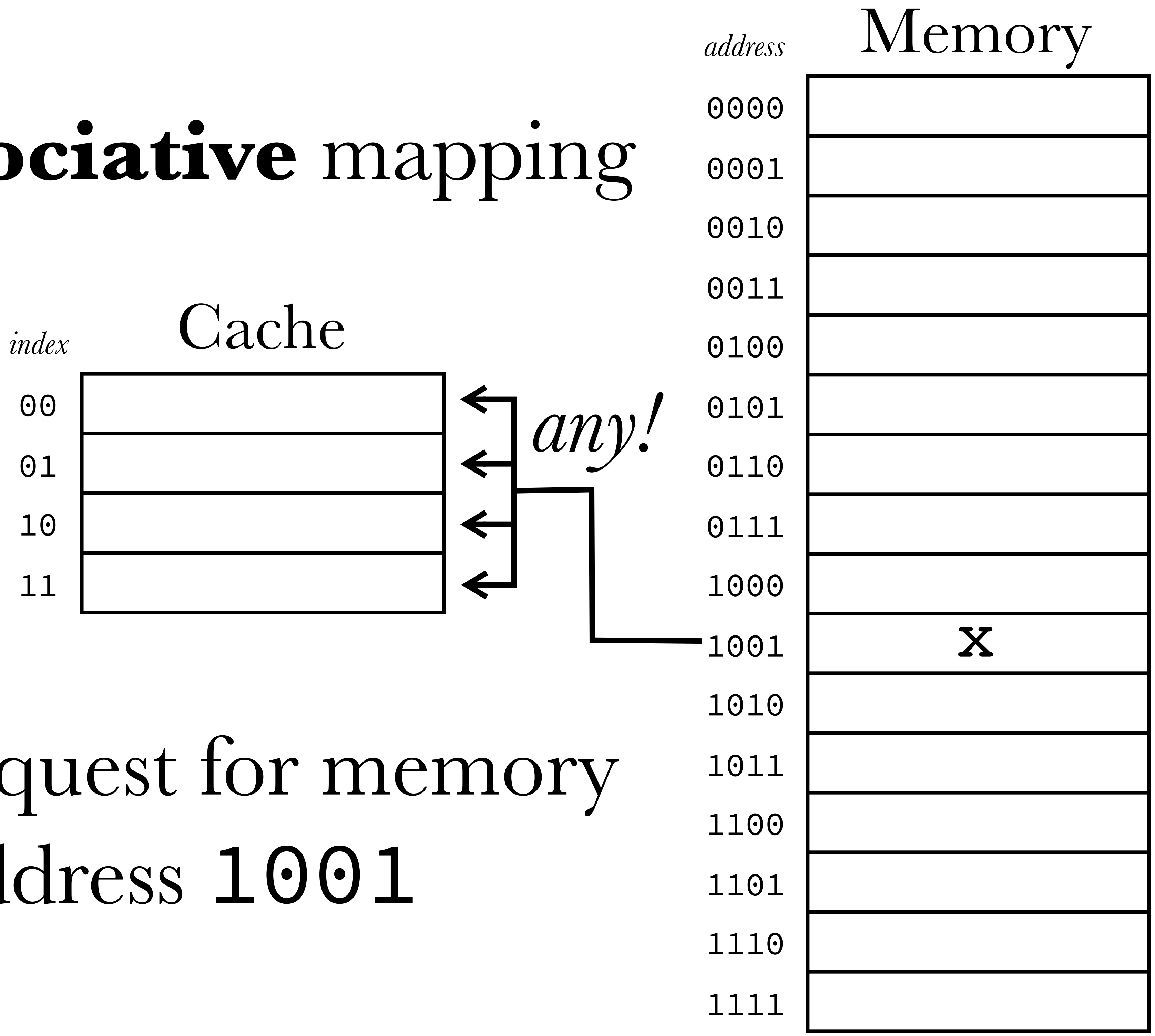
?

address Memory

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001 x
1010
1011
1100
1101
1110
1111

e.g., request for memory address `1001`

2) **associative** mapping

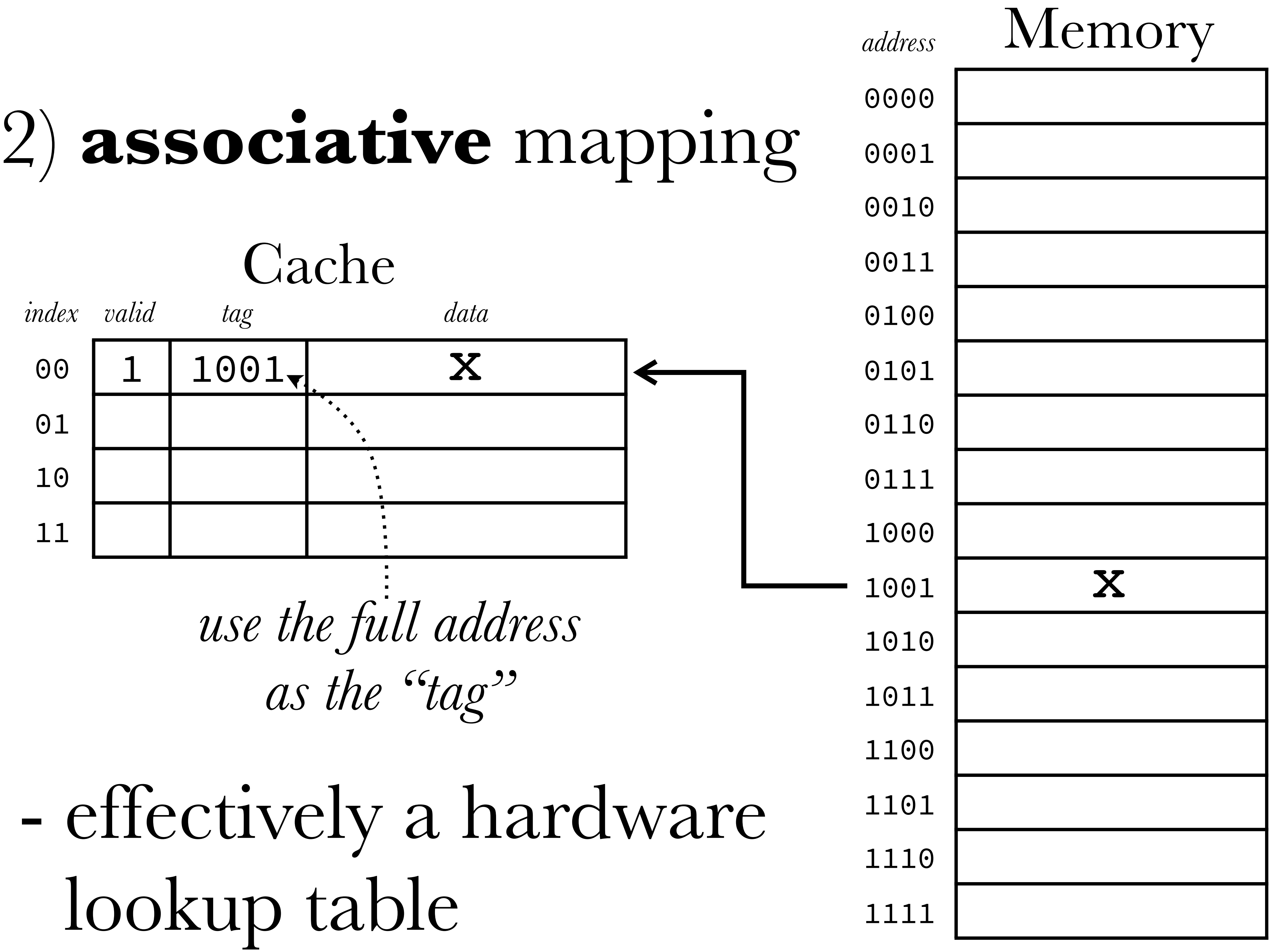


index	Cache
00	
01	
10	
11	

address	Memory
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	x
1010	
1011	
1100	
1101	
1110	
1111	

e.g., request for memory address `1001`

2) **associative** mapping



Cache

index	*valid*	*tag*	*data*
00	1	1001	X
01			
10			
11			

Memory

address	
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	X
1010	
1011	
1100	
1101	
1110	
1111	

use the full address as the "tag"

- effectively a hardware lookup table

2) **associative** mapping

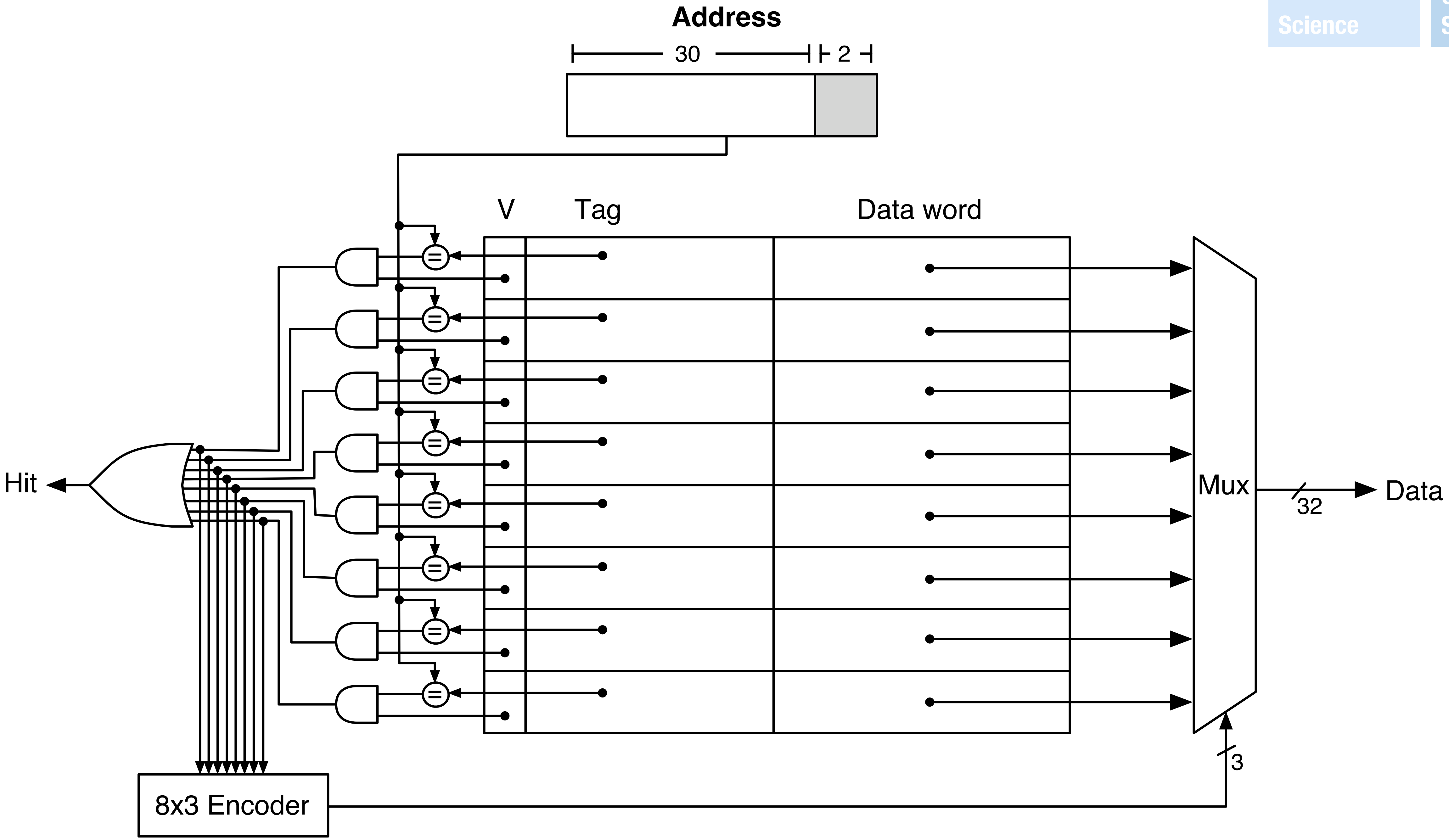
Cache

index	*valid*	*tag*	*data*
00	1	1001	x
01	1	1100	y
10	1	0001	w
11	1	0101	z

- can accommodate requests = # lines without conflict

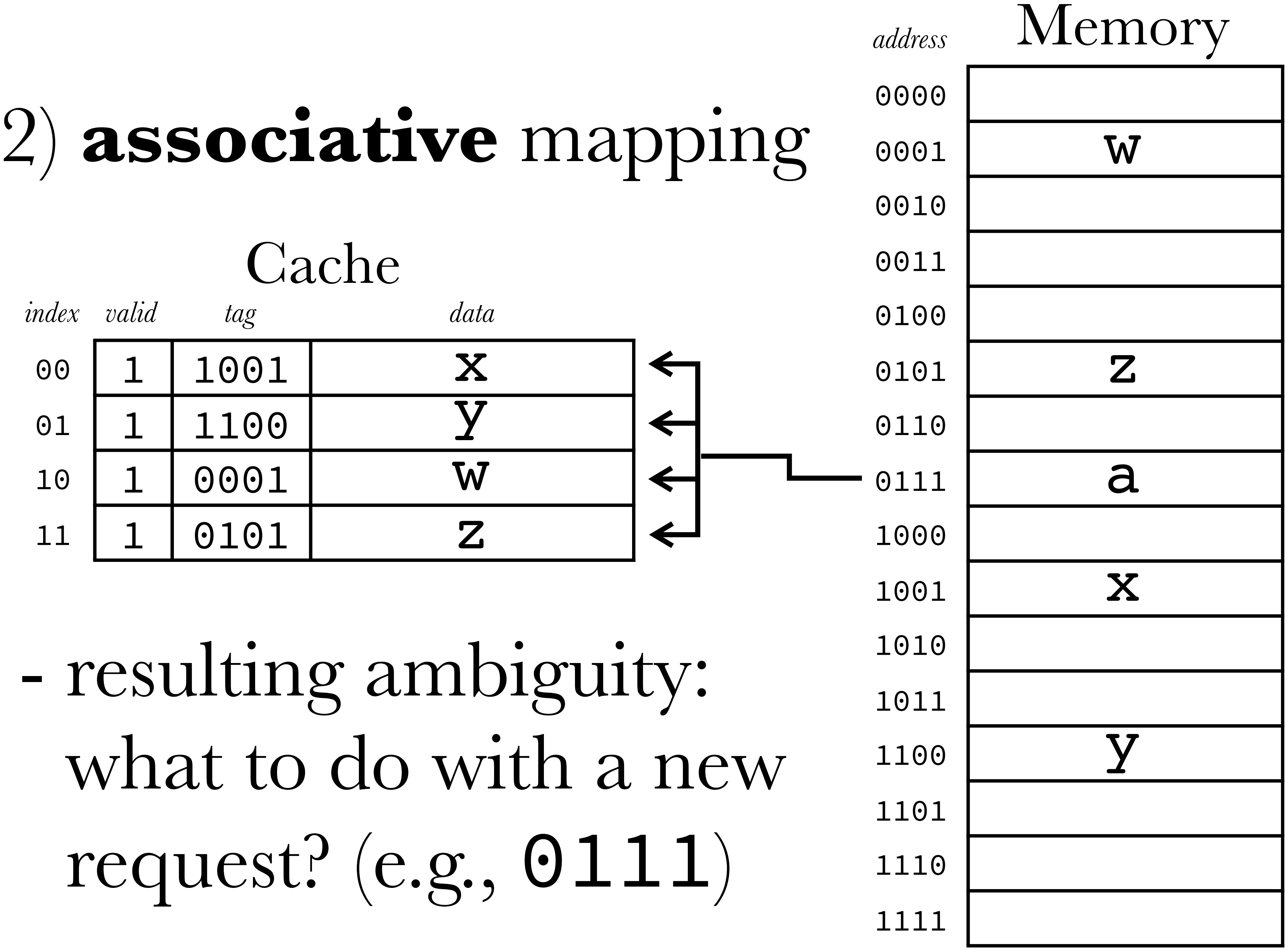
Memory

address	
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	
1000	
1001	x
1010	
1011	
1100	y
1101	
1110	
1111	



comparisons done in parallel (h/w): fast!

2) **associative** mapping



Cache

index	*valid*	*tag*	*data*
00	1	1001	x
01	1	1100	y
10	1	0001	w
11	1	0101	z

Memory

address	
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	a
1000	
1001	x
1010	
1011	
1100	y
1101	
1110	
1111	

- resulting ambiguity: what to do with a new request? (e.g., `0111`) associative caches require a *replacement policy* to decide which slot to evict, e.g.,

- FIFO (oldest is evicted)
- least frequently used (LFU)
- least recently used (LRU)

e.g., LRU replacement

Cache

index	*valid*	*tag*	*data*
00			
01			
10			
11			

- requests: 0101, 1001
1100, 0001
1010, 1001
0111,0001

Memory

address	Memory
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	a
1000	
1001	x
1010	b
1011	
1100	y
1101	
1110	
1111	

e.g., LRU replacement

Cache

index	*valid*	*tag*	*data*	*last used*
00	1	0101	z	*0*
01	1	1001	x	*1*
10	1	1100	y	*2*
11	1	0001	w	*3*

- requests: ~~0101~~, ~~1001~~
~~1100~~, ~~0001~~
1010, 1001
0111,1001

Memory

address	Memory
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	a
1000	
1001	x
1010	b
1011	
1100	y
1101	
1110	
1111	

e.g., LRU replacement

Cache

index	*valid*	*tag*	*data*	*last used*
00	1	1010	b	*4*
01	1	1001	x	*1*
10	1	1100	y	*2*
11	1	0001	w	*3*

- requests: ~~0101~~, ~~1001~~, ~~1100~~, ~~0001~~, ~~1010~~, 1001, 0111, 1001

Memory

address	Memory
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	a
1000	
1001	x
1010	b
1011	
1100	y
1101	
1110	
1111	

e.g., LRU replacement

Cache

index	valid	tag	data	last used
00	1	1010	b	*4*
01	1	1001	x	*5*
10	1	1100	y	*2*
11	1	0001	w	*3*

- requests: ~~0101~~, ~~1001~~
~~1100~~, ~~0001~~
~~1010~~, ~~1001~~
0111, 1001

Memory

address	Memory
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	a
1000	
1001	x
1010	b
1011	
1100	y
1101	
1110	
1111	

e.g., LRU replacement

Cache

index	*valid*	*tag*	*data*	*last used*
00	1	1010	b	*4*
01	1	1001	x	*5*
10	1	0111	a	*6*
11	1	0001	w	*3*

- requests: ~~0101~~, ~~1001~~
~~1100~~, ~~0001~~
~~1010~~, ~~1001~~
~~0111~~, 1001

Memory

address	Memory
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	a
1000	
1001	x
1010	b
1011	
1100	y
1101	
1110	
1111	

e.g., LRU replacement

Cache

index	*valid*	*tag*	*data*	*last used*
00	1	1010	b	*4*
01	1	1001	x	*7*
10	1	0111	a	*6*
11	1	0001	w	*3*

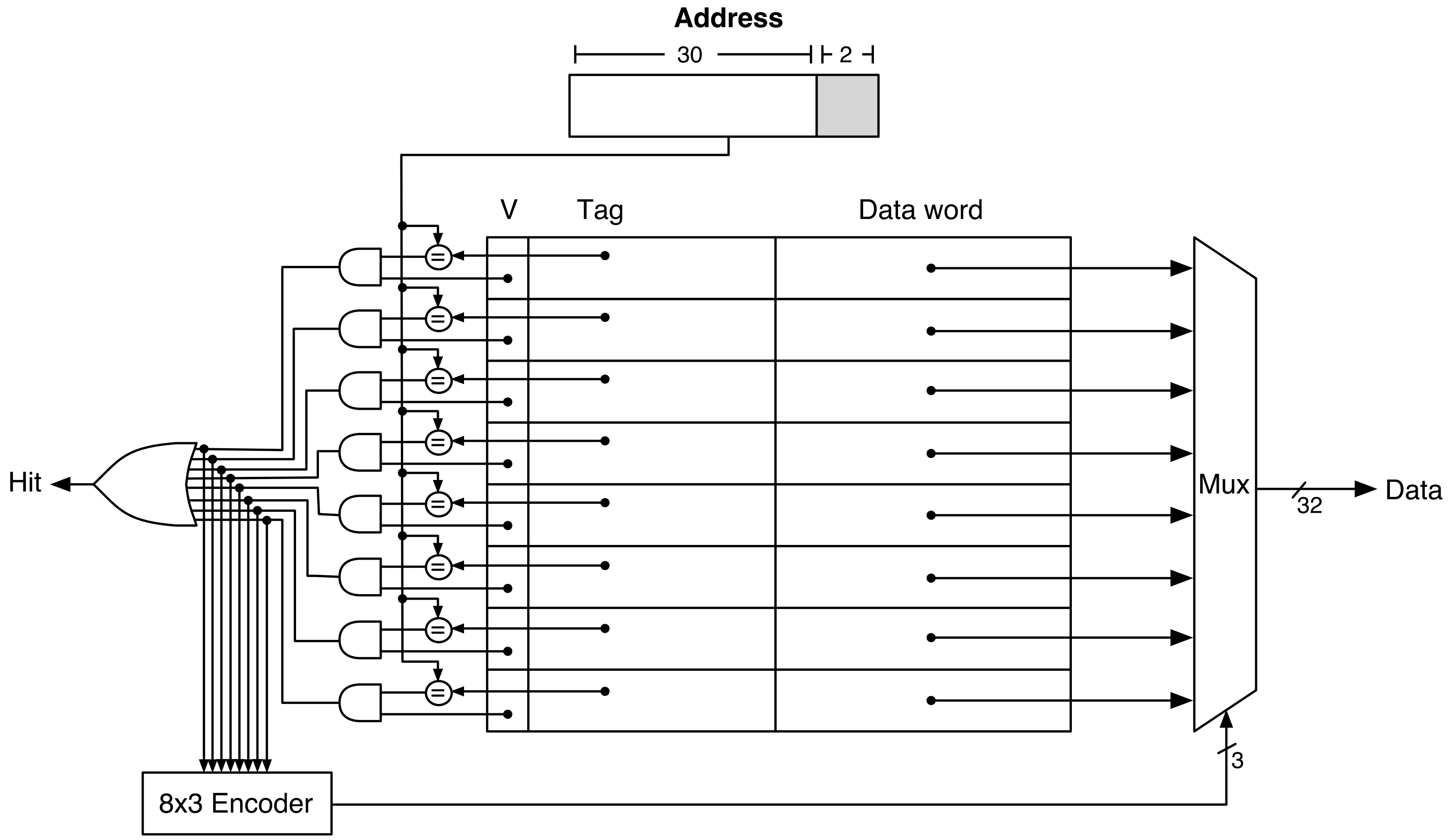
- requests: ~~0101~~, ~~1001~~
~~1100~~, ~~0001~~
~~1010~~, ~~1001~~
~~0111~~, ~~1001~~

address	Memory
0000	
0001	w
0010	
0011	
0100	
0101	z
0110	
0111	a
1000	
1001	x
1010	b
1011	
1100	y
1101	
1110	
1111	

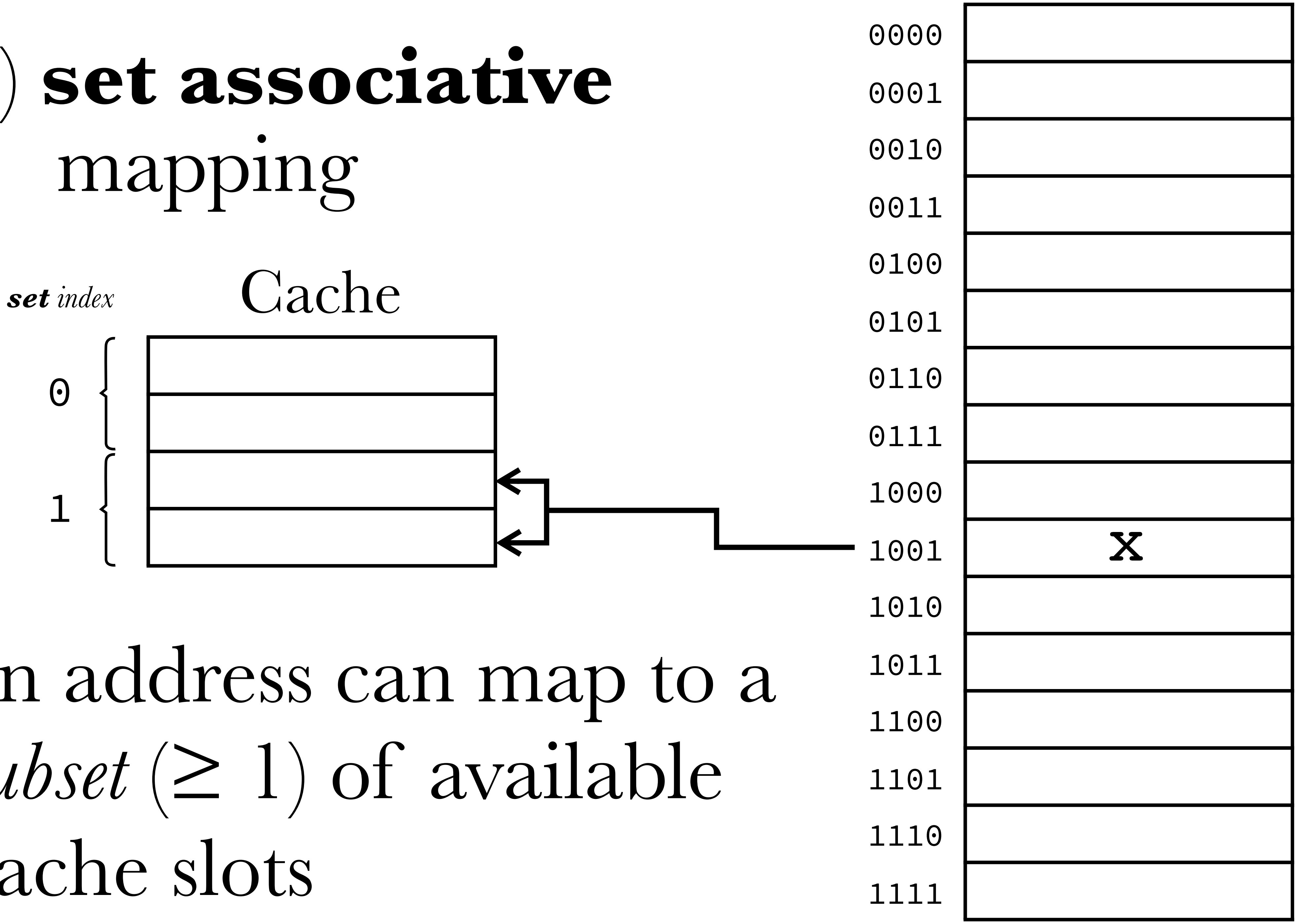
in practice, LRU is too complex (slow/expensive) to implement in hardware

use pseudo-LRU instead — e.g., track just MRU item, evict any other

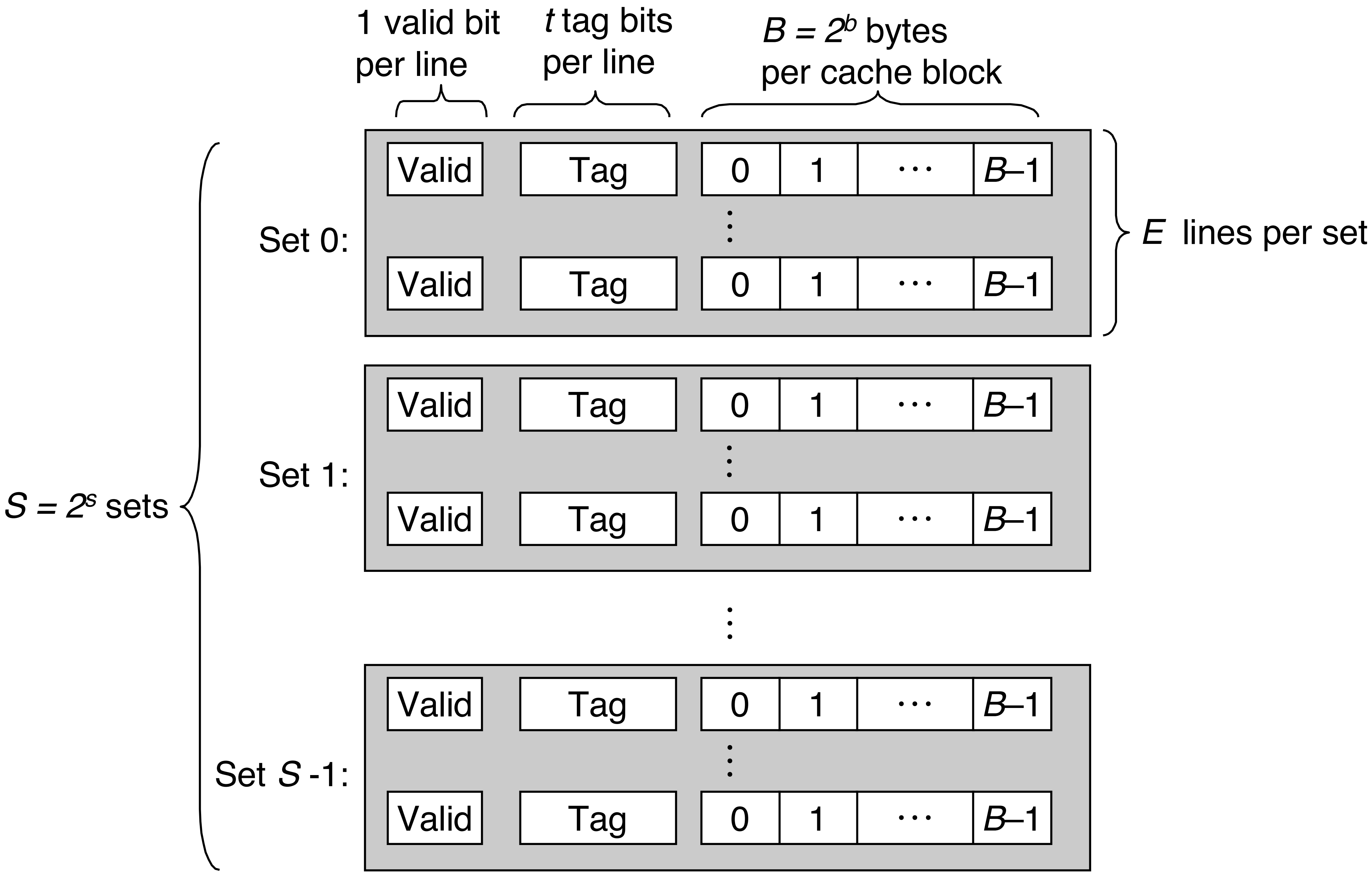
even with optimization, a *fully associative* cache with more than a few lines is prohibitively complex / expensive



3) **set associative** mapping



an address can map to a *subset* (≥ 1) of available cache slots



1 valid bit per line

t tag bits per line

$B = 2^b$ bytes per cache block

Set 0:

Valid	Tag	0	1	···	*B*–1
Valid	Tag	0	1	···	*B*–1

E lines per set

Set 1:

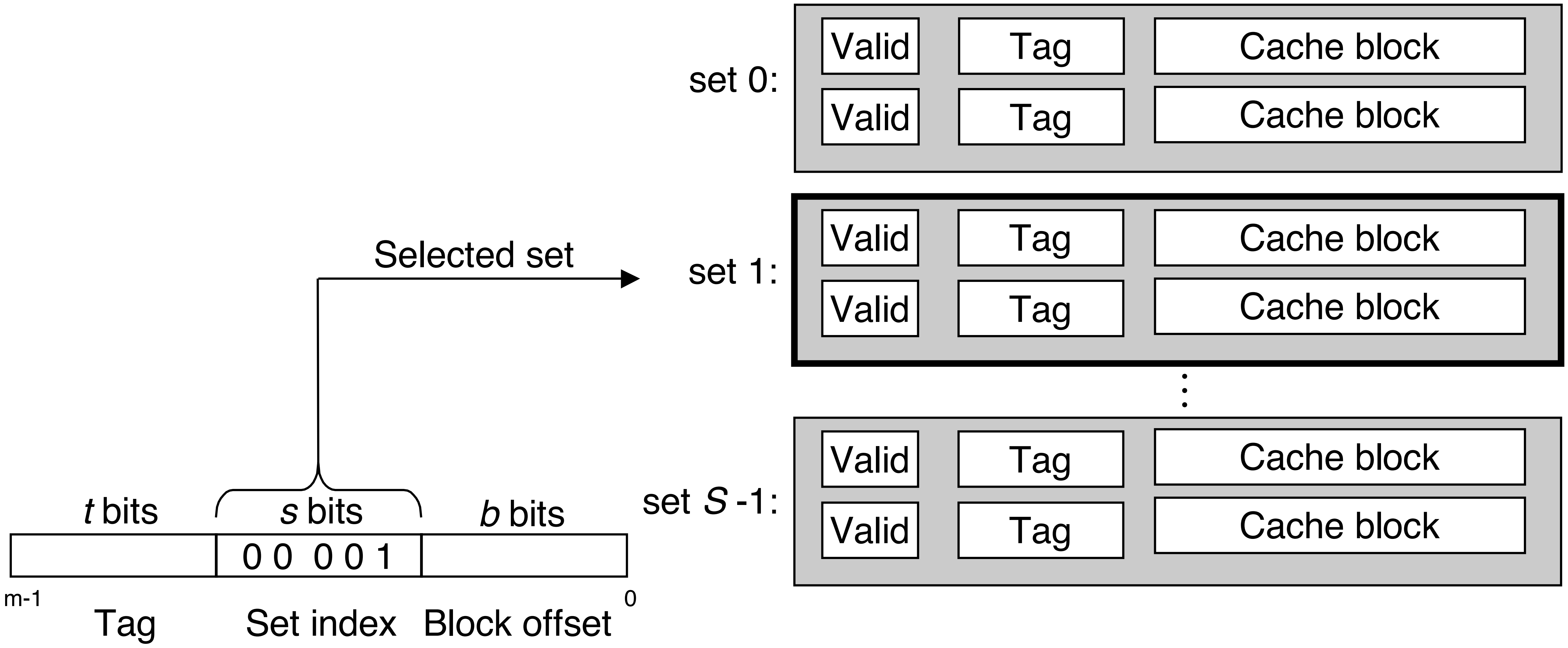
Valid	Tag	0	1	···	*B*–1
Valid	Tag	0	1	···	*B*–1

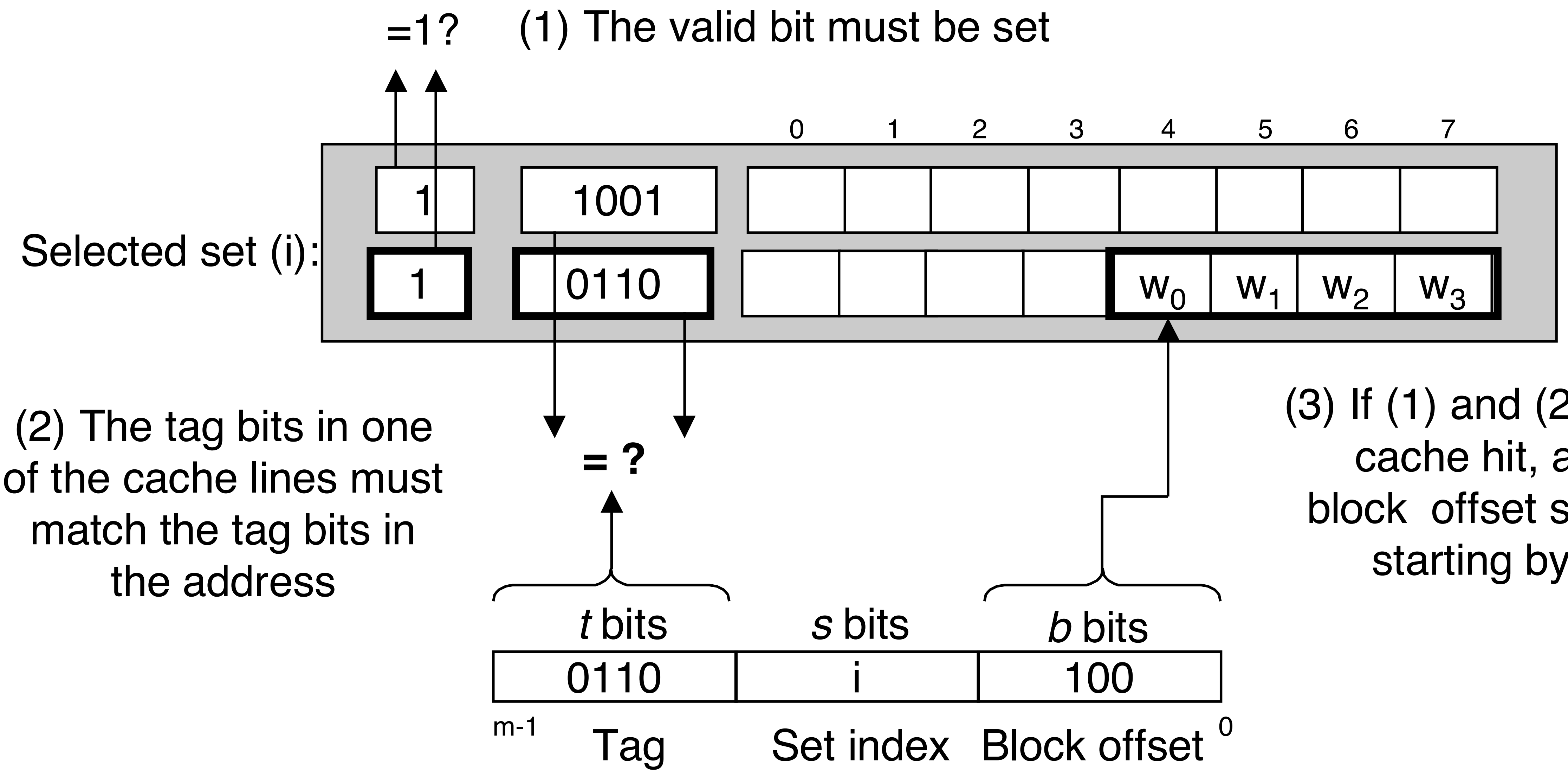
Set *S* -1:

Valid	Tag	0	1	···	*B*–1
Valid	Tag	0	1	···	*B*–1

$S = 2^s$ sets

Cache size: $C = B \times E \times S$ data bytes


set 0:
Valid
Tag
Cache block
Valid
Tag
Cache block
Selected set
set 1:
Valid
Tag
Cache block
Valid
Tag
Cache block
set S -1:
Valid
Tag
Cache block
Valid
Tag
Cache block
t bits
s bits
b bits
0 0 0 0 1
m-1
0
Tag
Set index
Block offset



(1) The valid bit must be set

(2) The tag bits in one of the cache lines must match the tag bits in the address

(3) If (1) and (2), then cache hit, and block offset selects starting byte

nomenclature:

- *n-way set associative* cache = *n* lines per set (each line containing 1 block)
- *direct mapped* cache: 1-way set associative
- *fully associative* cache: *n* = total # lines

Cache						
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	973	0	05	E2	6C	05
	C3B	1	0C	8E	FB	50
	89B	0	58	E0	EB	05
	64A	0	16	0C	F8	3E
1	929	0	B2	52	B9	2E
	C3A	1	95	07	51	3F
	B7B	0	DA	AC	B9	8E
	99A	1	9E	E3	20	03
2	5C0	0	C2	B1	FB	7C
	CEC	1	C8	2B	3E	D6
	B15	1	E0	05	FB	E8
	772	1	BE	D4	C7	79
3	745	1	92	74	C8	CB
	992	1	3C	76	25	89
	06C	1	66	41	2E	99
	FAB	1	C0	4D	08	88

Hits/Misses? Data returned if hit?

1. `0xCEC9`
2. `0xC3BC`

So far, only considered *read* requests;

What happens on a *write* request?

- don’t really need data *from* memory
- but if cache & memory out of sync, may need to eventually reconcile them

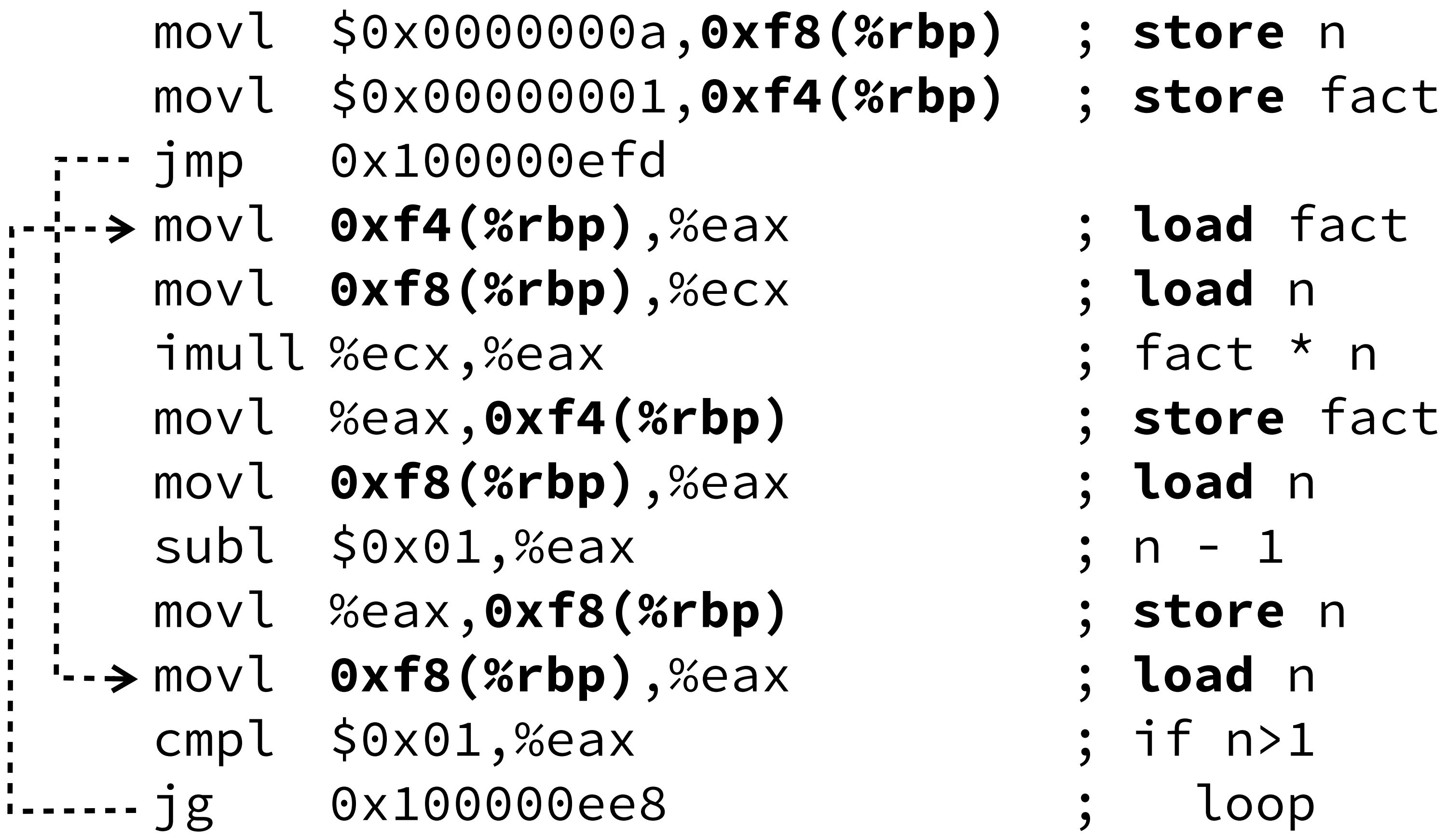
write hit	write-through	update memory & cache
	write-back	update cache only (requires “dirty bit”)
write miss	write-around	update memory only
	write-allocate	allocate space in cache for data, then write-hit

logical pairing:

1. write-through + write-around
2. write-back + write-allocate

With *write-back* policy, eviction (on future read/write) may require data-to-be-evicted to be written back to memory first.

```
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}
```



```
movl  $0x0000000a,0xf8(%rbp)  ; store n
movl  $0x00000001,0xf4(%rbp)  ; store fact
jmp   0x100000efd
movl  0xf4(%rbp),%eax         ; load fact
movl  0xf8(%rbp),%ecx         ; load n
imull %ecx,%eax               ; fact * n
movl  %eax,0xf4(%rbp)         ; store fact
movl  0xf8(%rbp),%eax         ; load n
subl  $0x01,%eax              ; n - 1
movl  %eax,0xf8(%rbp)         ; store n
movl  0xf8(%rbp),%eax         ; load n
cmpl  $0x01,%eax              ; if n>1
jg    0x100000ee8             ;   loop
```

Given: 2-way set assoc cache, 4-byte blocks.

# DRAM accesses with hit policies (1) vs. (2)?

(1) write-through + write-around

```
     movl  $0x0000000a,0xf8(%rbp)  ; write (around) to memory
     movl  $0x00000001,0xf4(%rbp)  ; write (around) to memory
     jmp   0x100000efd
     movl  0xf4(%rbp),%eax         ; read from memory → cache / cache
     movl  0xf8(%rbp),%ecx         ; read from memory → cache / cache
     imull %ecx,%eax
     movl  %eax,0xf4(%rbp)         ; write through (cache & memory)
     movl  0xf8(%rbp),%eax         ; read from cache
     subl  $0x01,%eax
     movl  %eax,0xf8(%rbp)         ; write through (cache & memory)
     movl  0xf8(%rbp),%eax         ; read from cache
     cmpl  $0x01,%eax
     jg    0x100000ee8
```

2 + 4 [first iteration]
+ 2 × # subsequent iterations

(1) write-back + write-allocate

```
movl  $0x0000000a,0xf8(%rbp)  ; allocate cache line
movl  $0x00000001,0xf4(%rbp)  ; allocate cache line
jmp   0x100000efd
movl  0xf4(%rbp),%eax         ; read from cache
movl  0xf8(%rbp),%ecx         ; read from cache
imull %ecx,%eax
movl  %eax,0xf4(%rbp)         ; update cache
movl  0xf8(%rbp),%eax         ; read from cache
subl  $0x01,%eax
movl  %eax,0xf8(%rbp)         ; update cache
movl  0xf8(%rbp),%eax         ; read from cache
cmpl  $0x01,%eax
jg    0x100000ee8
```

0 memory accesses! (but flush later)

i.e., write-back & write-allocate allow the cache to *absorb* multiple writes to memory why would you ever want write-through / write-around?

- to minimize cache complexity
- if *miss penalty* is not significant

cache metrics:

- *hit time*: time to detect hit and return requested data
- *miss penalty*: time to detect miss, retrieve data, update cache, and return data

cache metrics:

- *hit time* mostly depends on cache complexity (e.g., size & associativity)

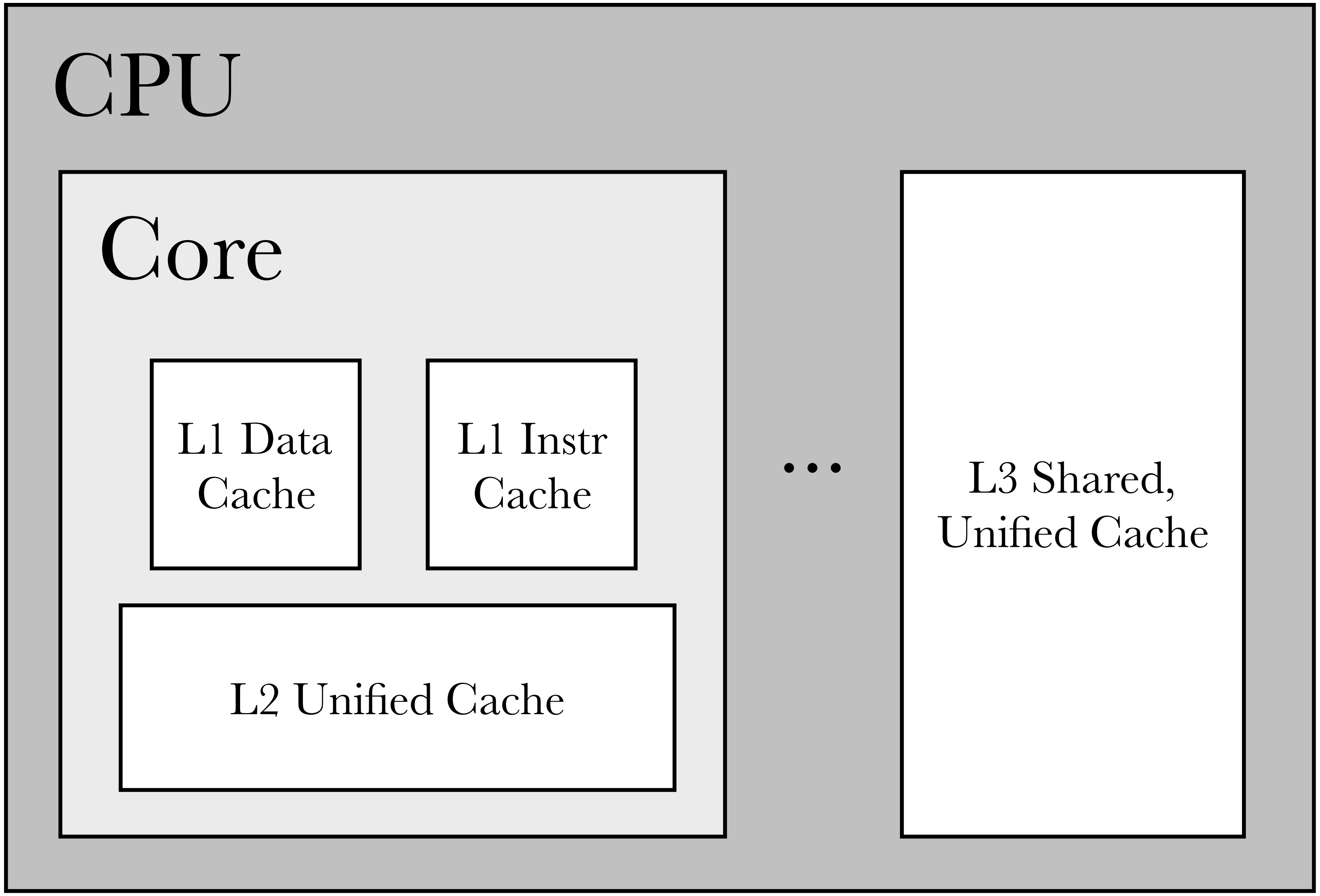
- *miss penalty* mostly depends on latency of lower level in memory hierarchy catch:

- best hit time favors simple design (e.g., small, low associativity)
- but simple caches = high miss rate; unacceptable if miss penalty is high!

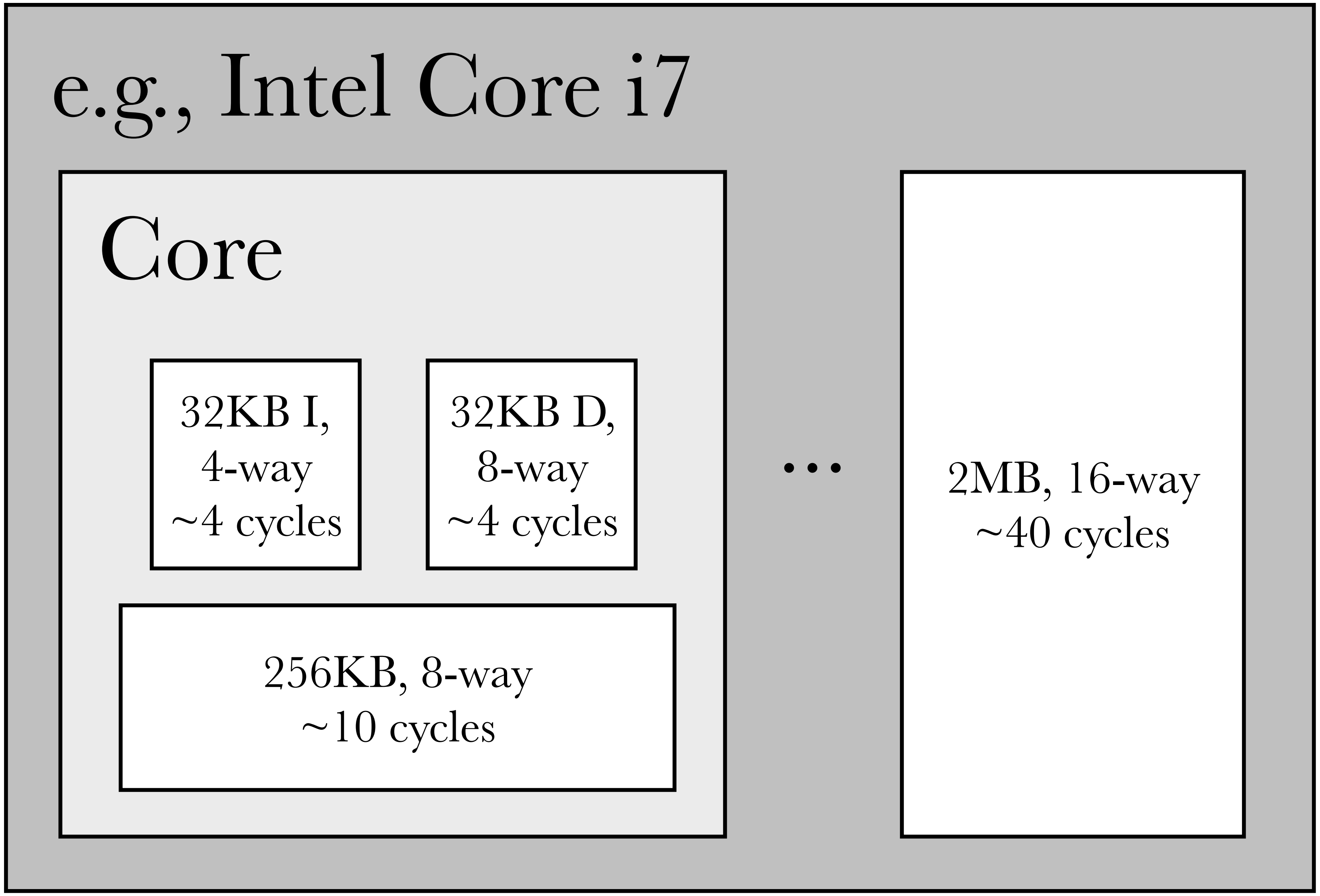
solution: use *multiple levels* of caching

closer to CPU: focus on optimizing hit time, possibly at expense of hit rate

closer to DRAM: focus on optimizing hit rate, possibly at expense of hit time



multi-level cache



multi-level cache

… but what does any of this have to do with systems programming?!?

§Cache-Friendly Code

In general, cache friendly code:

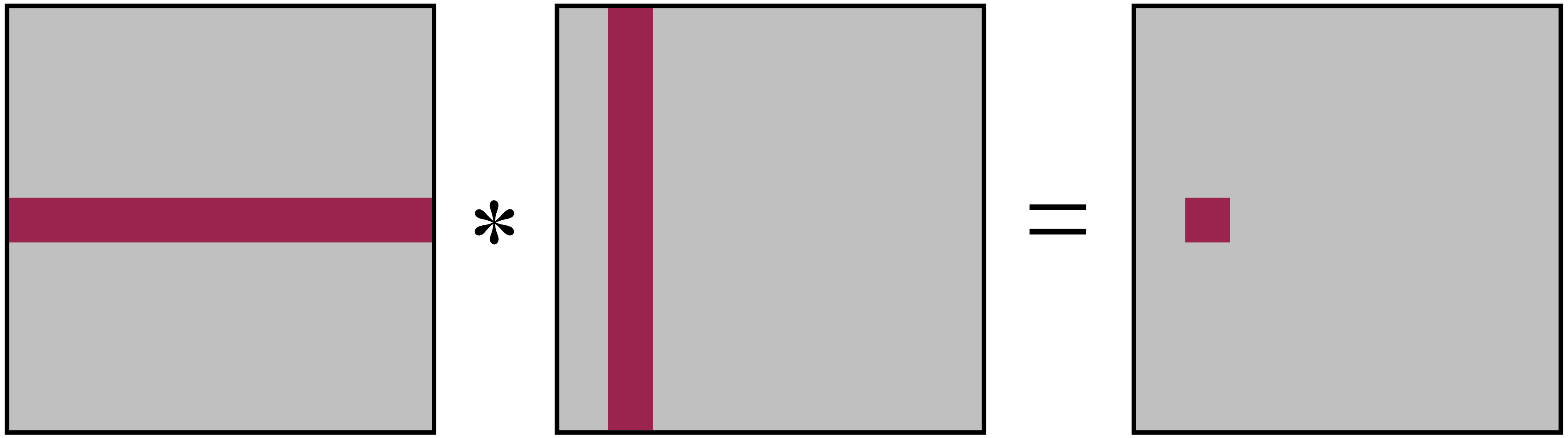
- exhibits *high locality* (temporal & spatial)
- maximizes cache *utilization*
- keeps *working set* size small
- avoids random memory access patterns

case study in software/cache interaction:

matrix multiplication

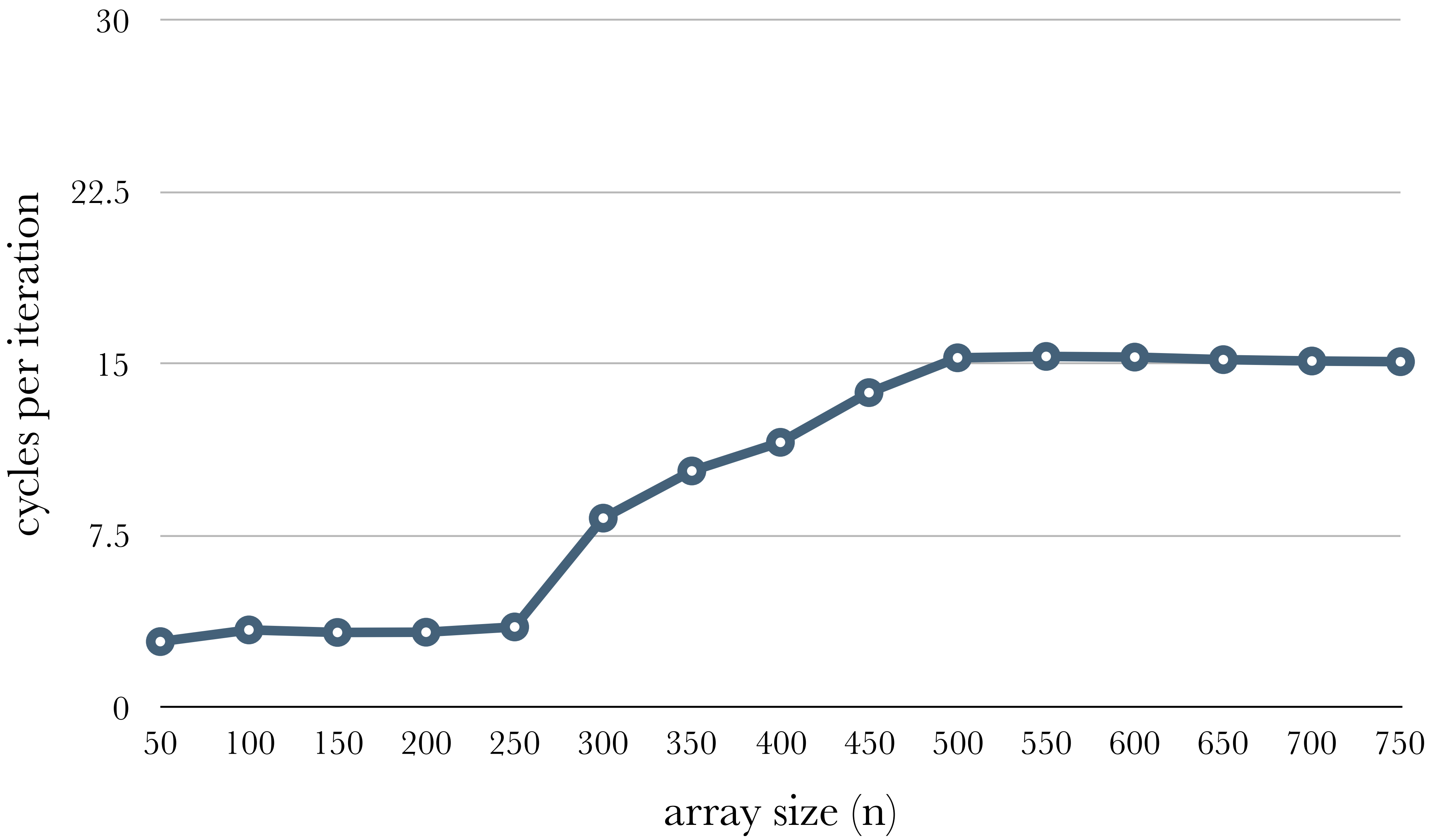
$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

$$\begin{aligned} c_{ij} &= \begin{pmatrix} a_{i1} & a_{i2} & a_{i3} \end{pmatrix} \cdot \begin{pmatrix} b_{1j} & b_{2j} & b_{3j} \end{pmatrix} \\ &= a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} \end{aligned}$$

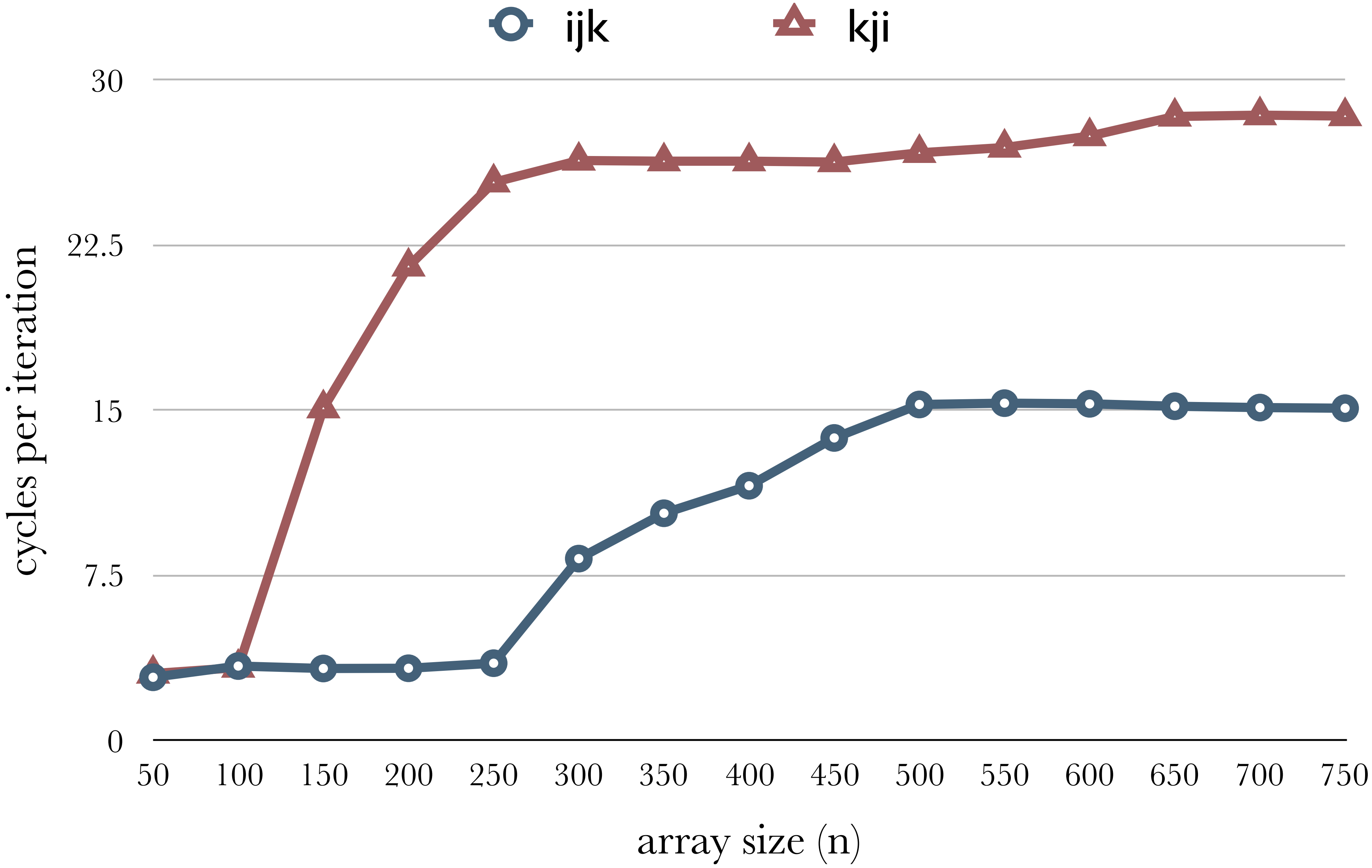


canonical implementation:

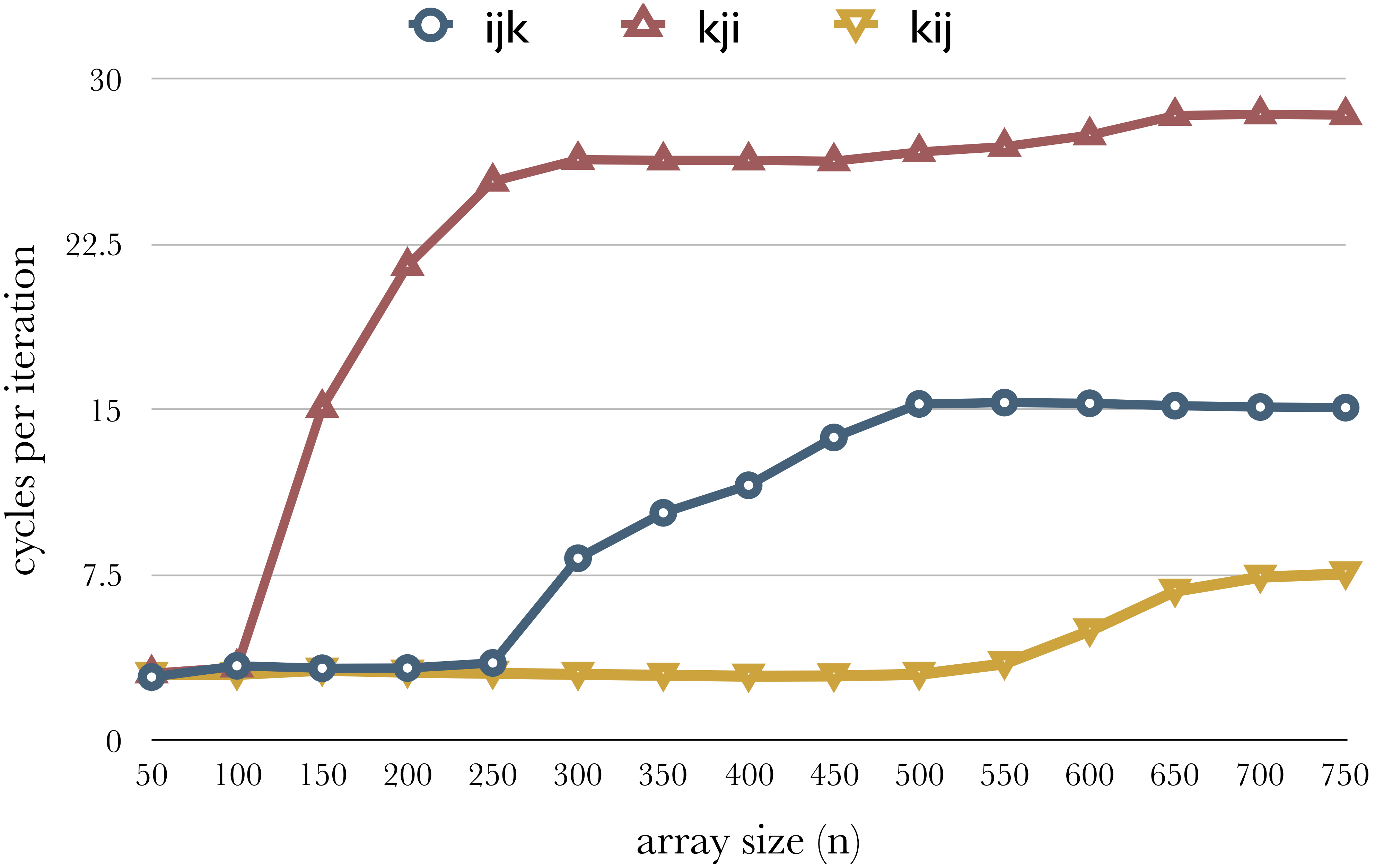
```
#define MAXN 1000
typedef double array[MAXN][MAXN];

/* multiply (compute the inner product of) two square matrices
 * A and B with dimensions n x n, placing the result in C  */
void matrix_mult(array A, array B, array C, int n) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < n; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```


30
22.5
15
7.5
0
cycles per iteration
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
array size (n)

```
void kji(array A, array B, array C, int n) {
    int i, j, k;
    double r;

    for (k = 0; k < n; k++) {
        for (j = 0; j < n; j++) {
            r = B[k][j];
            for (i = 0; i < n; i++)
                C[i][j] += A[i][k]*r;
        }
    }
}
```


ijk
kji
30
22.5
15
7.5
0
cycles per iteration
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
array size (n)

```
void kij(array A, array B, array C, int n) {
    int i, j, k;
    double r;

    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            r = A[i][k];
            for (j = 0; j < n; j++)
                C[i][j] += r*B[k][j];
        }
    }
}
```

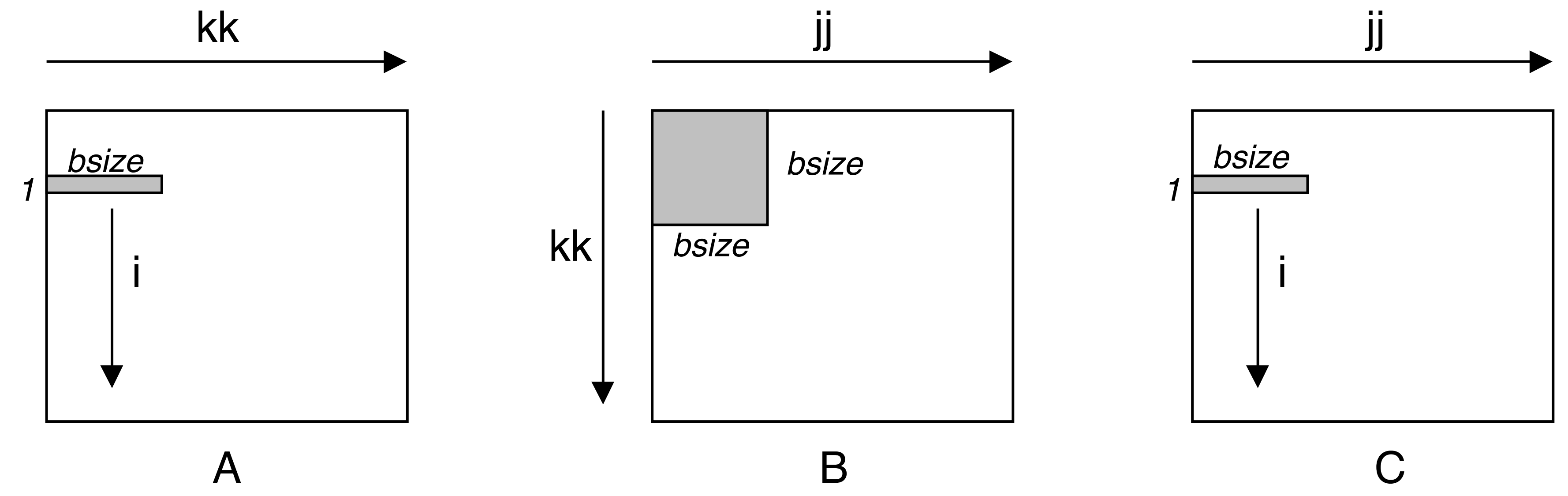

ijk
kji
kij
30
22.5
15
7.5
0
cycles per iteration
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
array size (n)

remaining problem: *working set size* grows beyond capacity of cache

smaller strides can help, to an extent (by leveraging spatial locality) idea for optimization: deal with matrices in smaller chunks at a time that will fit in the cache — “blocking”

```
/* "blocked" matrix multiplication, assuming n is evenly
 * divisible by bsize */
void bijk(array A, array B, array C, int n, int bsize) {
    int i, j, k, kk, jj;
    double sum;

    for (kk = 0; kk < n; kk += bsize) {
        for (jj = 0; jj < n; jj += bsize) {
            for (i = 0; i < n; i++) {
                for (j = jj; j < jj + bsize; j++) {
                    sum = C[i][j];
                    for (k = kk; k < kk + bsize; k++) {
                        sum += A[i][k]*B[k][j];
                    }
                    C[i][j] = sum;
                }
            }
        }
    }
}
```

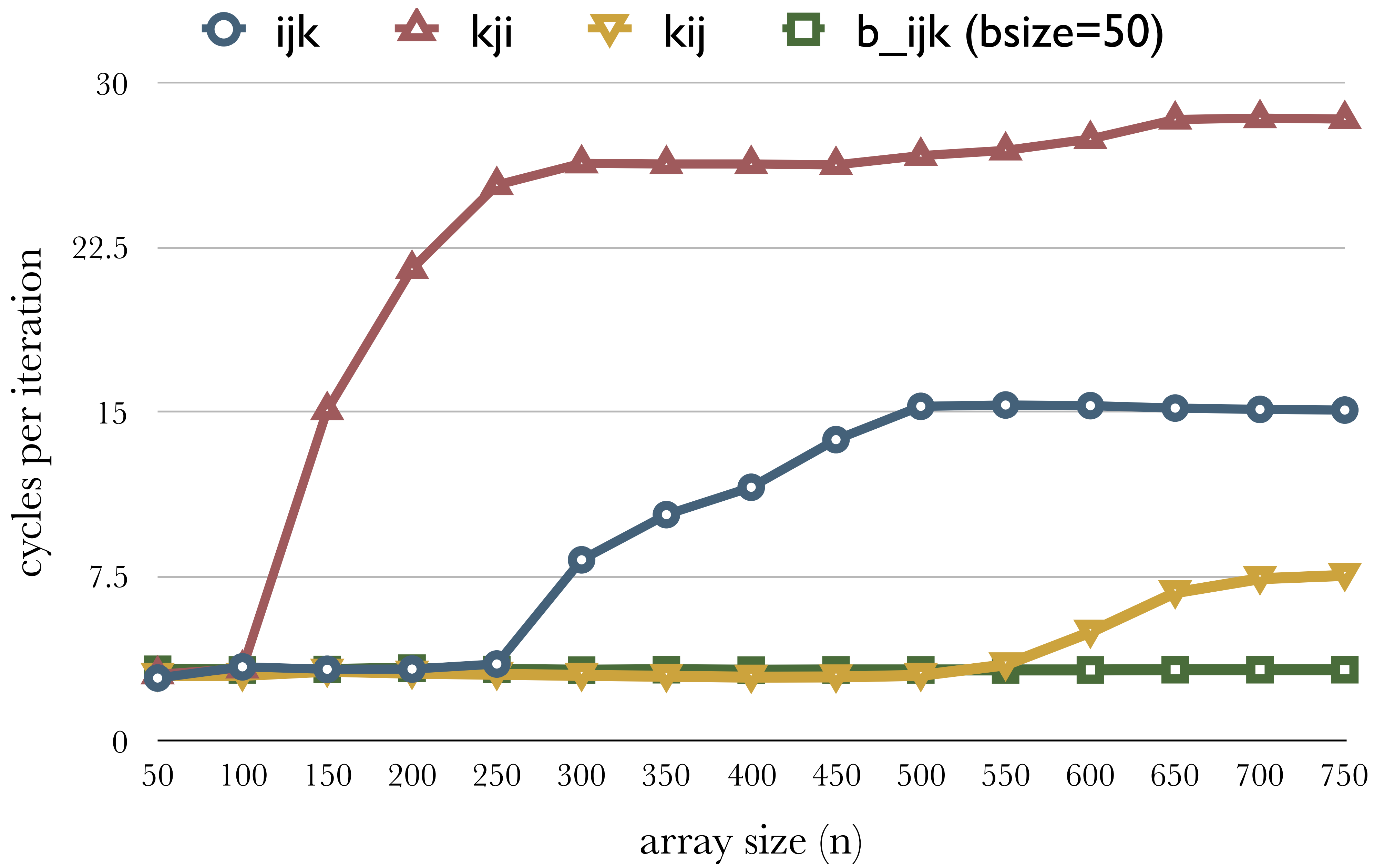
```
/* "blocked" matrix multiplication, assuming n is evenly
 * divisible by bsize */
void bijk(array A, array B, array C, int n, int bsize) {
    int i, j, k, kk, jj;
    double sum;

    for (kk = 0; kk < n; kk += bsize) {
        for (jj = 0; jj < n; jj += bsize) {
            for (i = 0; i < n; i++) {
                for (j = jj; j < jj + bsize; j++) {
                    sum = C[i][j];
                    for (k = kk; k < kk + bsize; k++) {
                        sum += A[i][k]*B[k][j];
                    }
                    C[i][j] = sum;
```



Use *1 x bsize* row sliver *bsize* times

Use *bsize x bsize* block *n* times in succession

Update successive elements of *1 x bsize* row sliver


ijk
kji
kij
b_ijk (bsize=50)
30
22.5
15
7.5
0
cycles per iteration
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
array size (n)

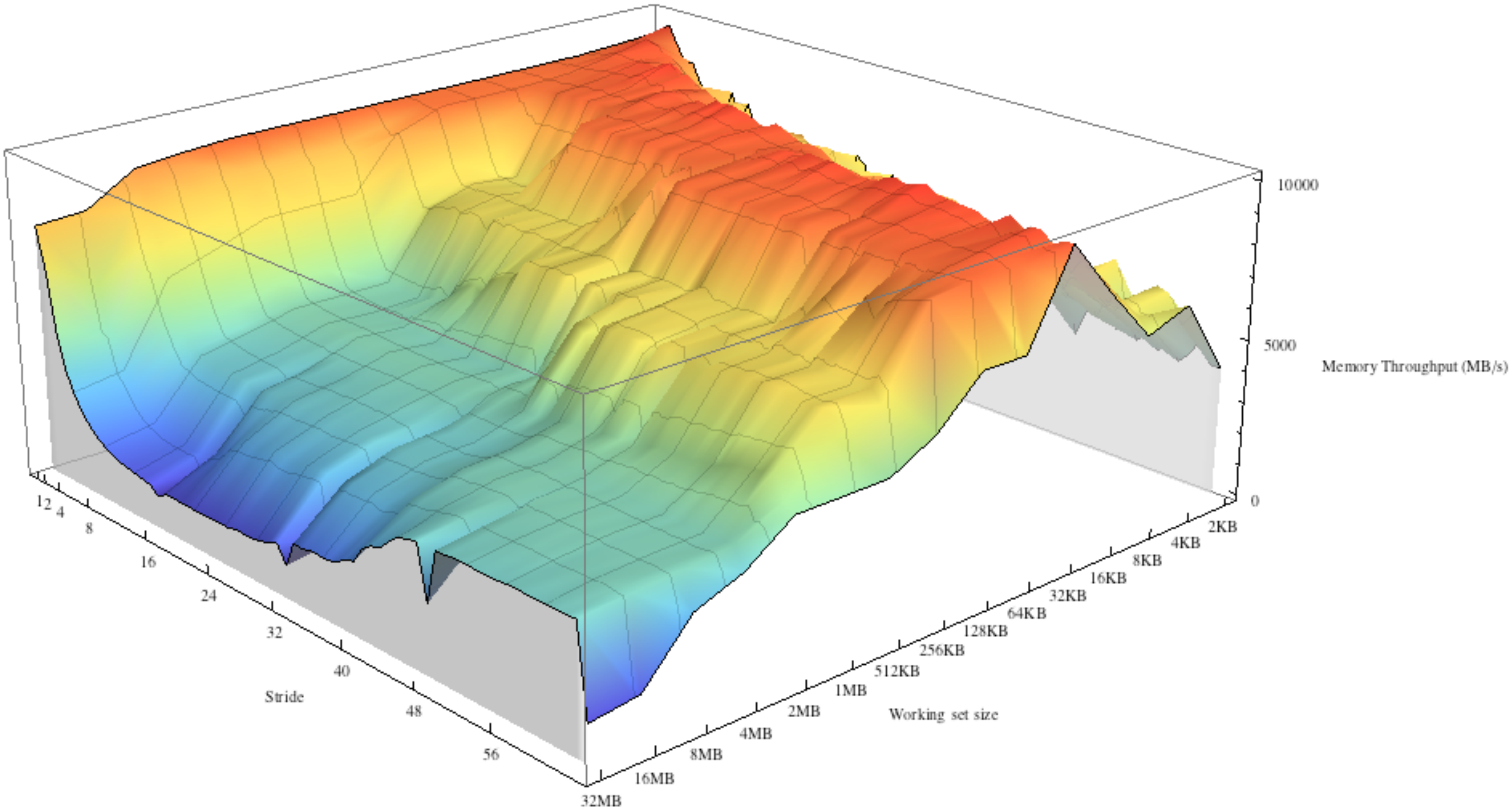
```
/* Quite a bit uglier without making previous assumption! */
void bijk(array A, array B, array C, int n, int bsize) {
    int i, j, k, kk, jj;
    double sum;
    int en = bsize * (n/bsize); /* Amount that fits evenly into blocks */

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            C[i][j] = 0.0;

    for (kk = 0; kk < en; kk += bsize) {
        for (jj = 0; jj < en; jj += bsize) {
            for (i = 0; i < n; i++) {
                for (j = jj; j < jj + bsize; j++) {
                    sum = C[i][j];
                    for (k = kk; k < kk + bsize; k++) {
                        sum += A[i][k]*B[k][j];
                    }
                    C[i][j] = sum;
                }
            }
        }
        /* Now finish off rest of j values */
        for (i = 0; i < n; i++) {
            for (j = en; j < n; j++) {
                sum = C[i][j];
                for (k = kk; k < kk + bsize; k++) {
                    sum += A[i][k]*B[k][j];
                }
                C[i][j] = sum;
            }
        }
    }
    /* Now finish remaining k values */
    for (jj = 0; jj < en; jj += bsize) {
        for (i = 0; i < n; i++) {
            for (j = jj; j < jj + bsize; j++) {
                sum = C[i][j];
                for (k = en; k < n; k++) {
                    sum += A[i][k]*B[k][j];
                }
                C[i][j] = sum;
            }
        }
    }

    /* Now finish off rest of j values */
    for (i = 0; i < n; i++) {
        for (j = en; j < n; j++) {
            sum = C[i][j];
            for (k = en; k < n; k++) {
                sum += A[i][k]*B[k][j];
            }
            C[i][j] = sum;
        }
    }
} /* end of bijk */
```

See CS:APP MEM:BLOCKING *“Web Aside” for more details*

Another nice demo of software-cache interaction: the *memory mountain* demo

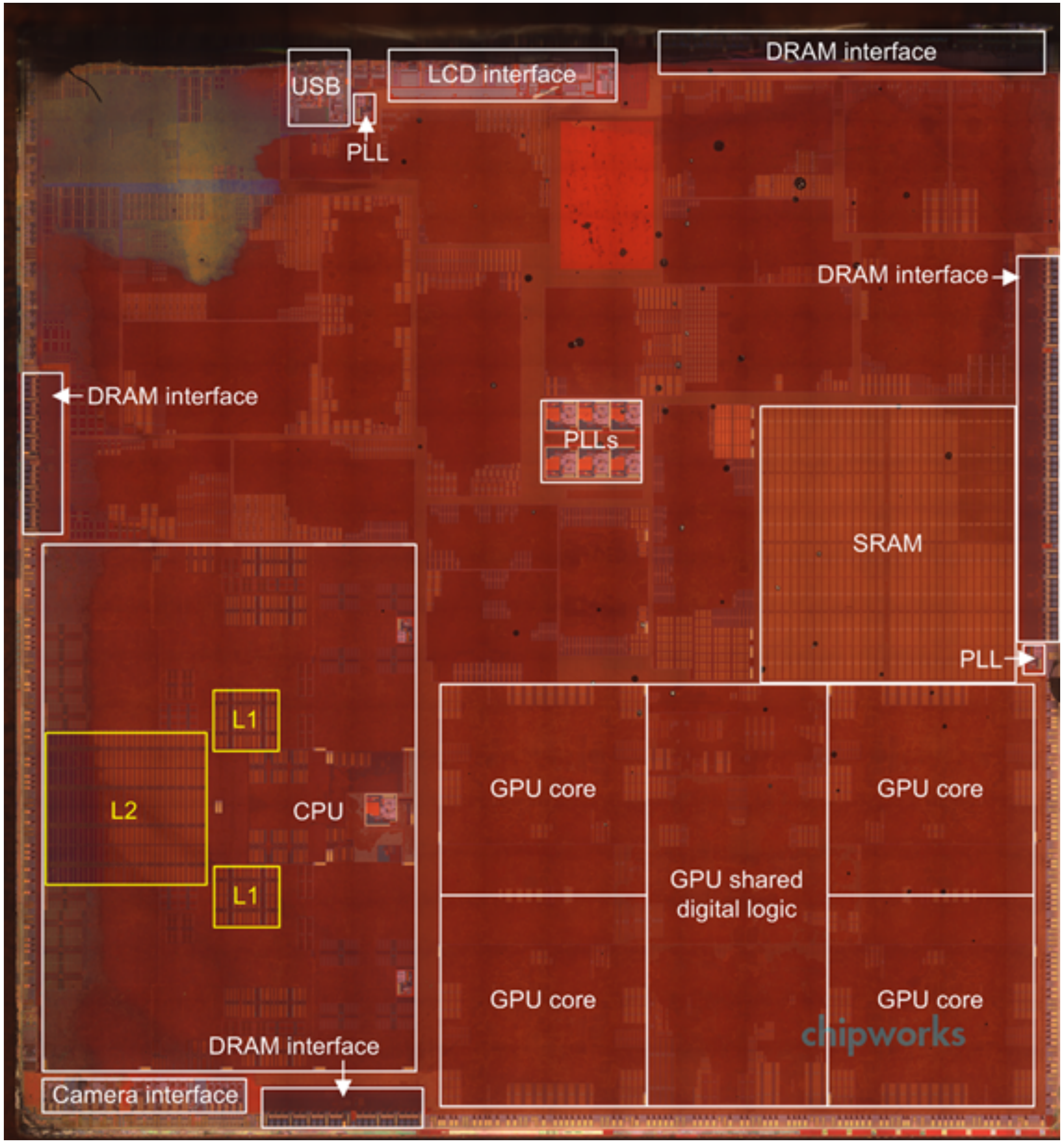
```
/*
 * test - Iterate over first "elems" elements of array "data"
 *        with stride of "stride".
 */
void test(int elems, int stride) {
    int i;
    double result = 0.0;
    volatile double sink;

    for (i = 0; i < elems; i += stride) {
        result += data[i];
    }
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* run - Run test(elems, stride) and return read throughput (MB/s).
 *       "size" is in bytes, "stride" is in array elements, and
 *       Mhz is CPU clock frequency in Mhz.
 */
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(double);

    test(elems, stride);                     /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

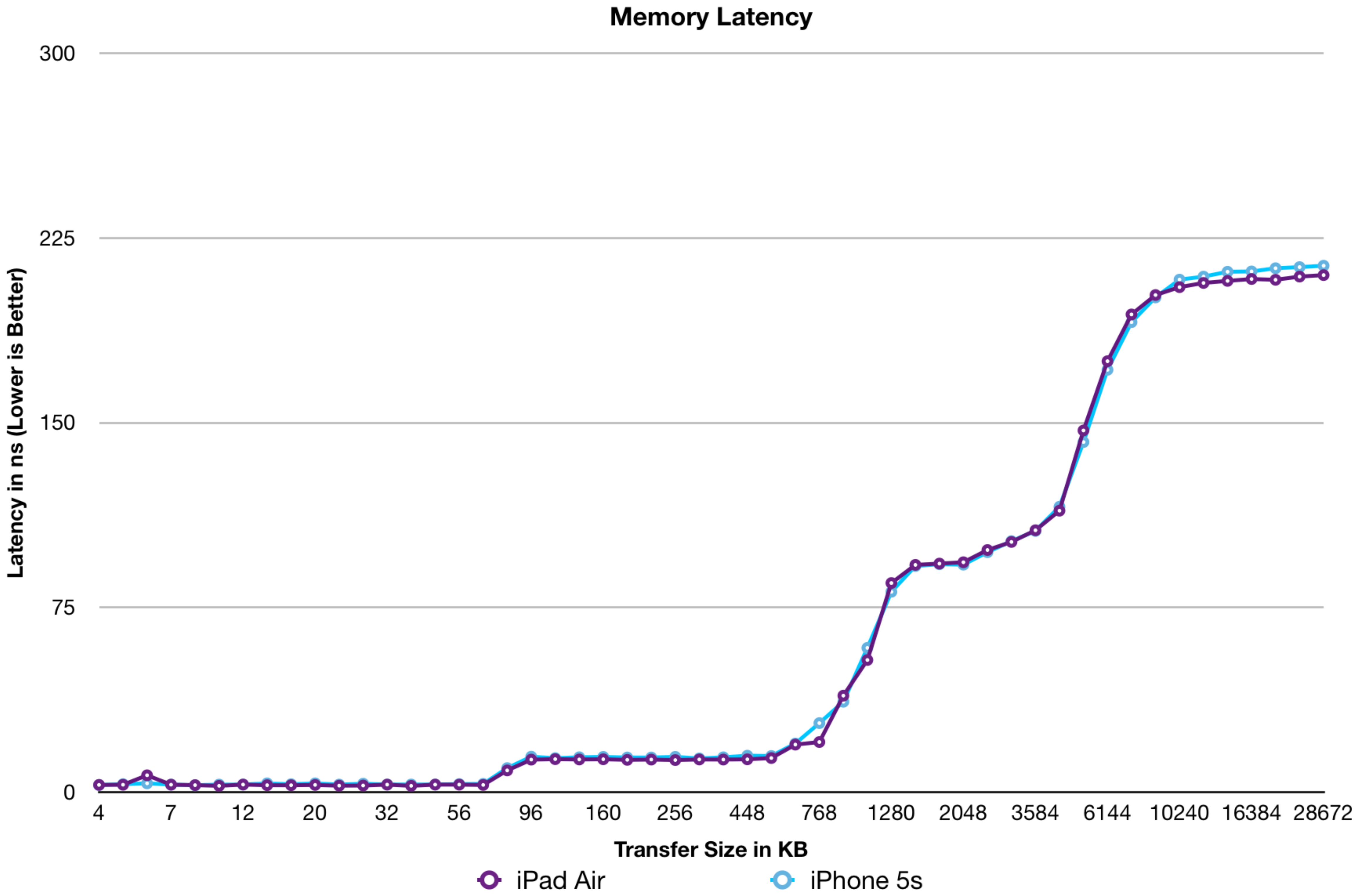
```
#define MINBYTES (1 << 11)  /* Working set size ranges from 2 KB */
#define MAXBYTES (1 << 25)  /* ... up to 64 MB */
#define MAXSTRIDE 64        /* Strides range from 1 to 64 elems */
#define MAXELEMS MAXBYTES/sizeof(double)

double data[MAXELEMS];      /* The global array we'll be traversing */

int main() {
    int size;        /* Working set size (in bytes) */
    int stride;      /* Stride (in array elements) */
    double Mhz;      /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data */
    Mhz = mhz(0);              /* Estimate the clock frequency */

    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++) {
            printf("%.1f\t", run(size, stride, Mhz));

        }
    }
}
```

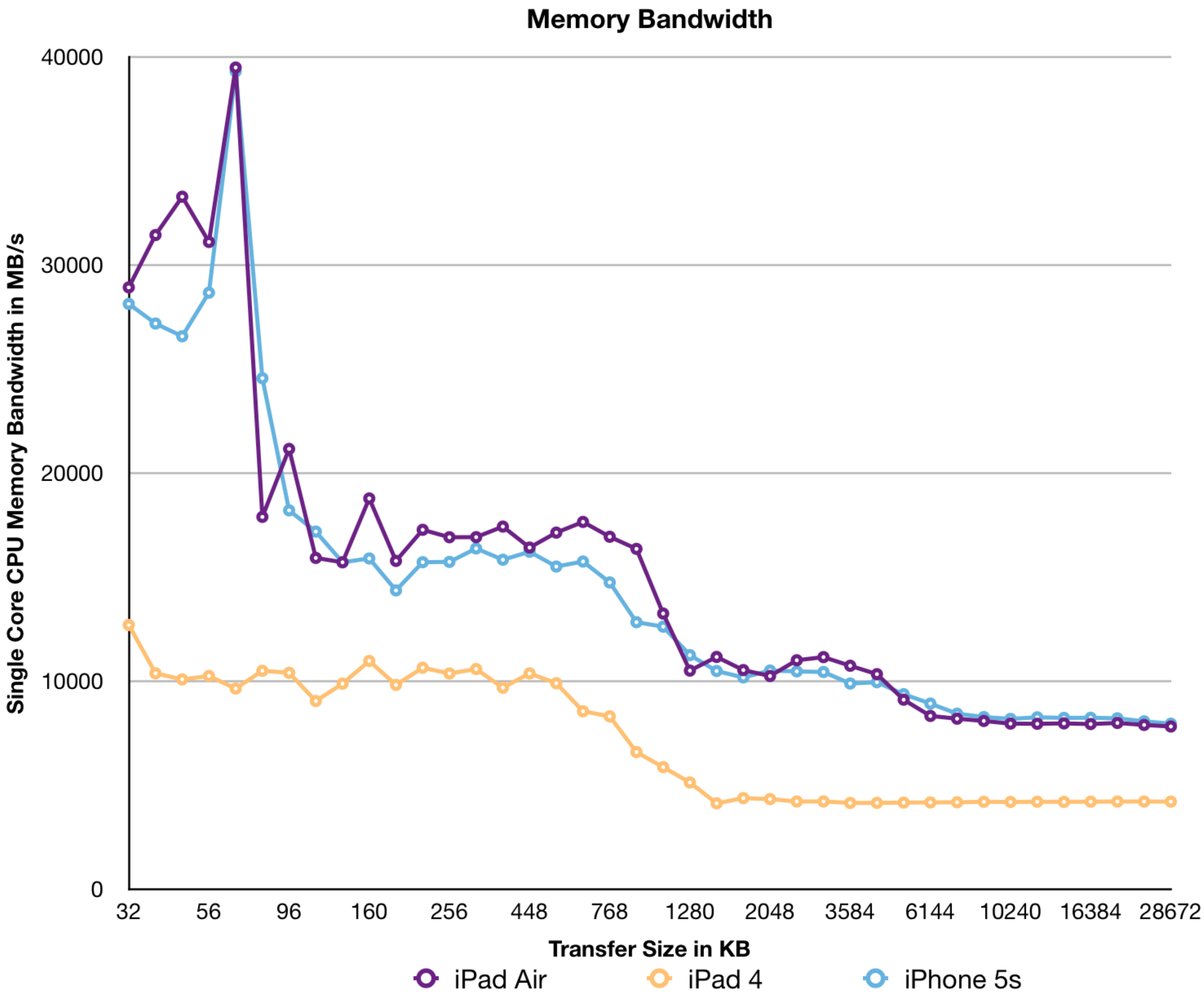

10000
5000
0
Memory Throughput (MB/s)
1
2
4
8
16
24
32
40
48
56
Stride
2KB
4KB
8KB
16KB
32KB
64KB
128KB
256KB
512KB
1MB
2MB
4MB
8MB
16MB
32MB
Working set size

recently: AnandTech's Apple A7 analysis

http://www.anandtech.com/show/7460/apple-ipad-air-review/2


USB
LCD interface
DRAM interface
PLL
DRAM interface
DRAM interface
PLLs
SRAM
PLL
L1
L2
CPU
L1
GPU core
GPU shared digital logic
GPU core
GPU core
GPU core
chipworks
DRAM interface
Camera interface


Memory Latency
Latency in ns (Lower is Better)
300
225
150
75
0
4
7
12
20
32
56
96
160
256
448
768
1280
2048
3584
6144
10240
16384
28672
Transfer Size in KB
iPad Air
iPhone 5s


Memory Bandwidth
Single Core CPU Memory Bandwidth in MB/s
40000
30000
20000
10000
0
32
56
96
160
256
448
768
1280
2048
3584
6144
10240
16384
28672
Transfer Size in KB
iPad Air
iPad 4
iPhone 5s

Demo: cachegrind

```
ssh fourier ; cd classes/cs351/repos/examples/mem

less matrixmul.c
valgrind --tool=cachegrind ./a.out 0 1
valgrind --tool=cachegrind ./a.out 1 1
valgrind --tool=cachegrind ./a.out 2 1
```