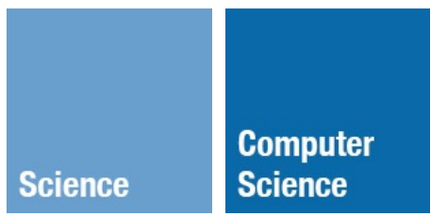
Memory Hierarchy & Caching



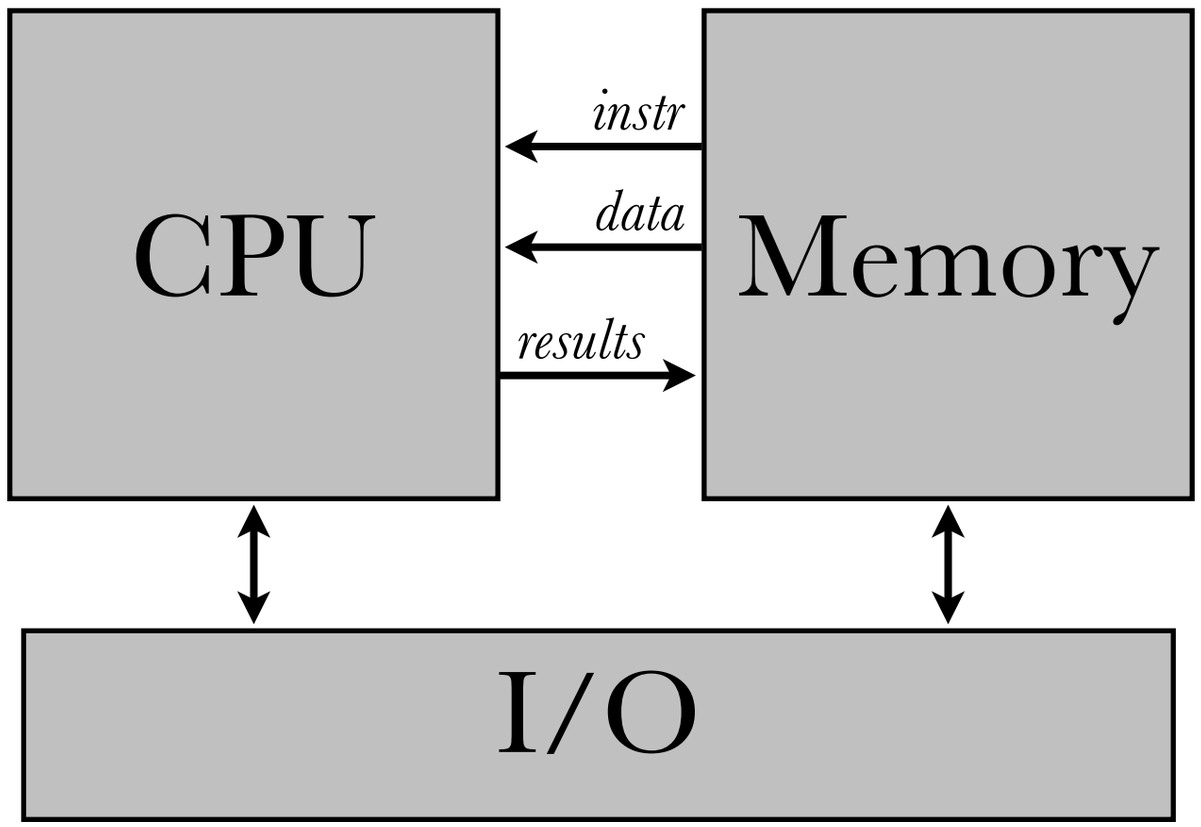
CS 351: Systems Programming
Melanie Cornelius

Slides and course content obtained with permission from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

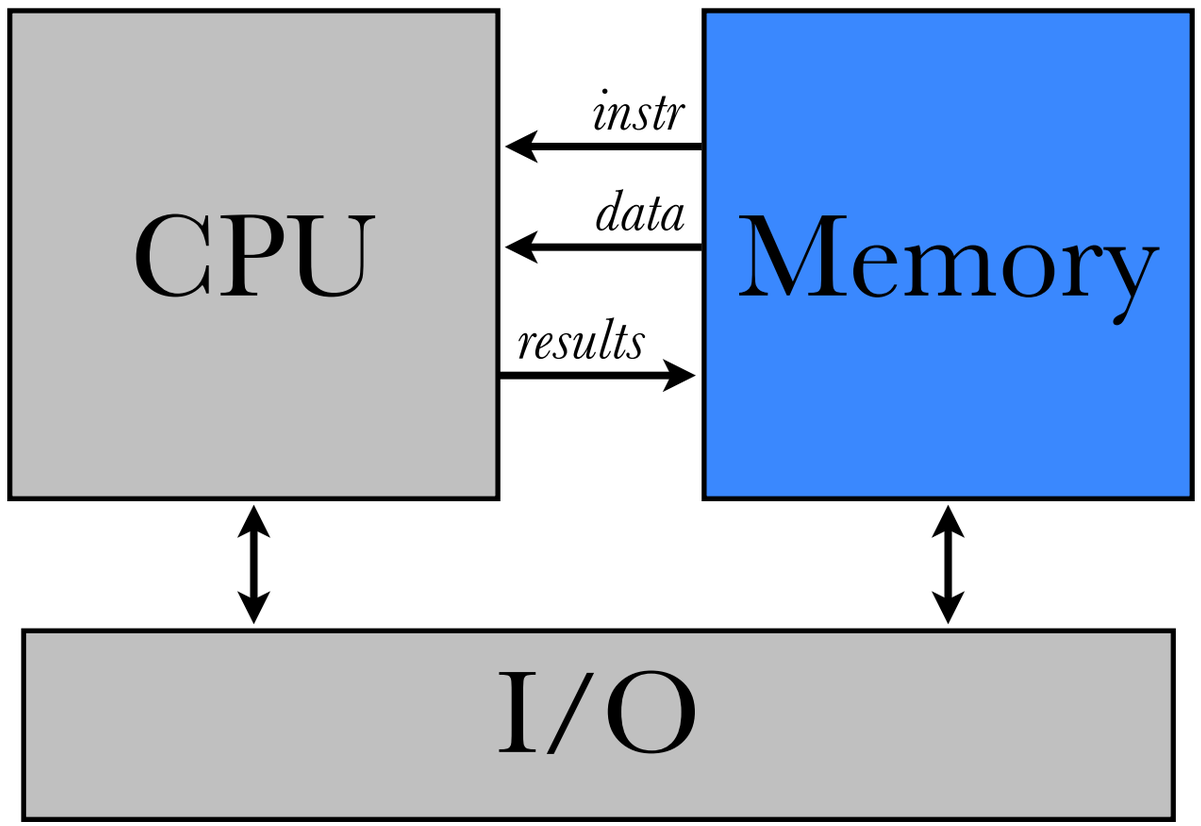
Why skip from process mgmt to memory?!

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

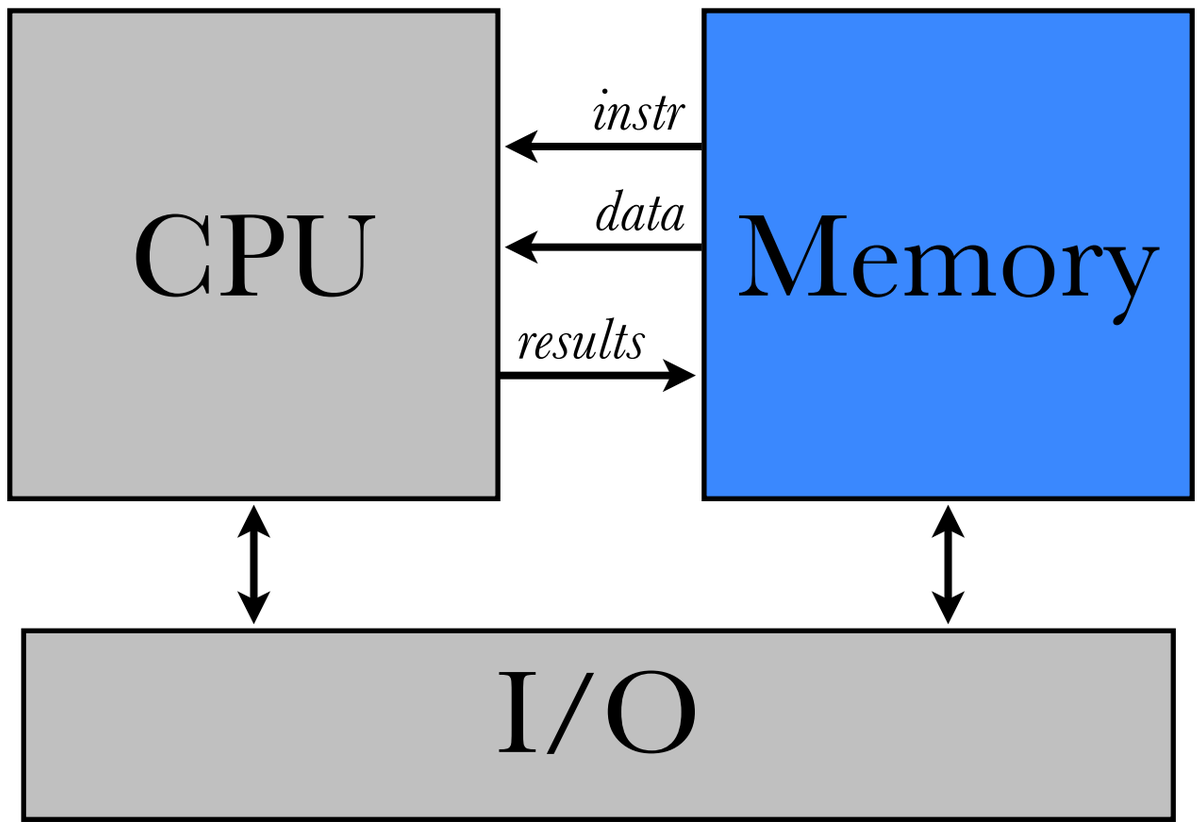


recall the *Von Neumann architecture*

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

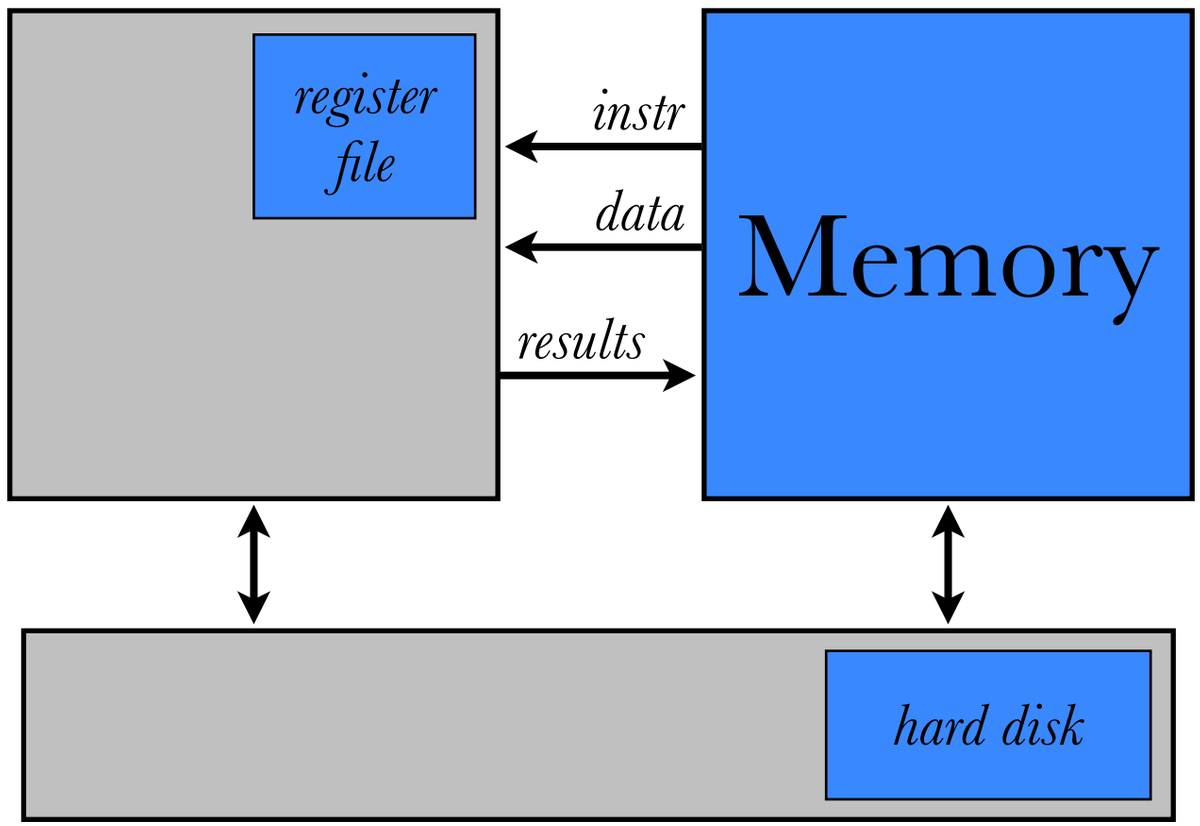


"memory" is an *idealized* storage device that holds our programs (instructions) and data (operands)



colloquially: "RAM", *random access memory*

~ big array of byte-accessible data

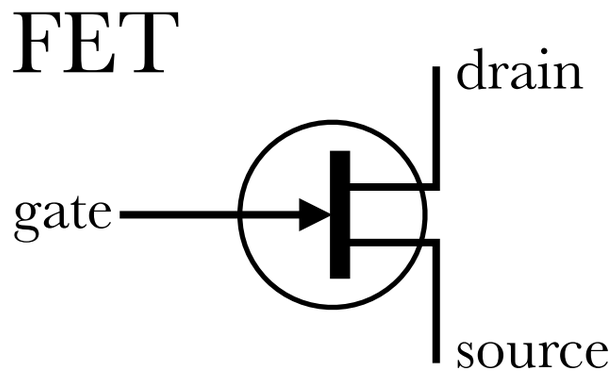


in reality, “memory” is a combination of storage systems with very different access characteristics

common types of “memory”:

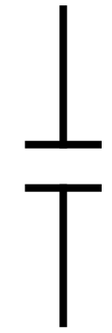
SRAM, DRAM, NVRAM, HDD

SRAM



- **S**tatic **R**andom **A**ccess **M**emory
  - Built entirely with transistors (MOSFETs)
    - 6+ transistors per bit — not very dense, and quite expensive!
  - Data stable as long as power applied

DRAM

capacitor 

- **D**ynamic **R**andom **A**ccess **M**emory
  - 1 capacitor + 1 transistor per bit
    - Requires periodic “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 $\approx 1\times10^9$ 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!

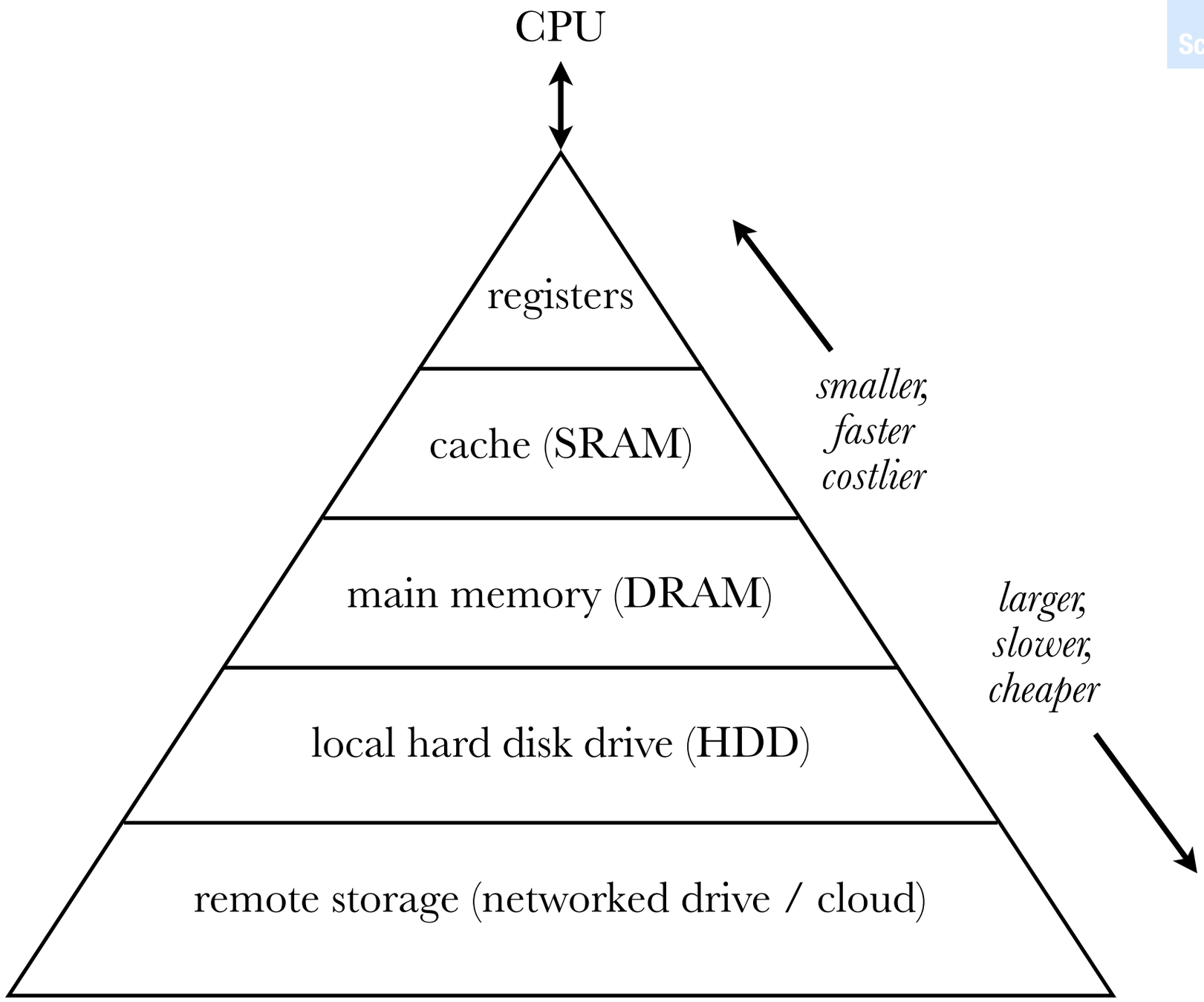
Relative Speeds

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 - 2 TB	~100 cycles	(ns)
SSD	≤ 8 TB	~10,000 cycles	(µs)
HDD	≤ 25 TB	~10,000,000 cycles	(ms)

human blink ≈ 350,000 µs

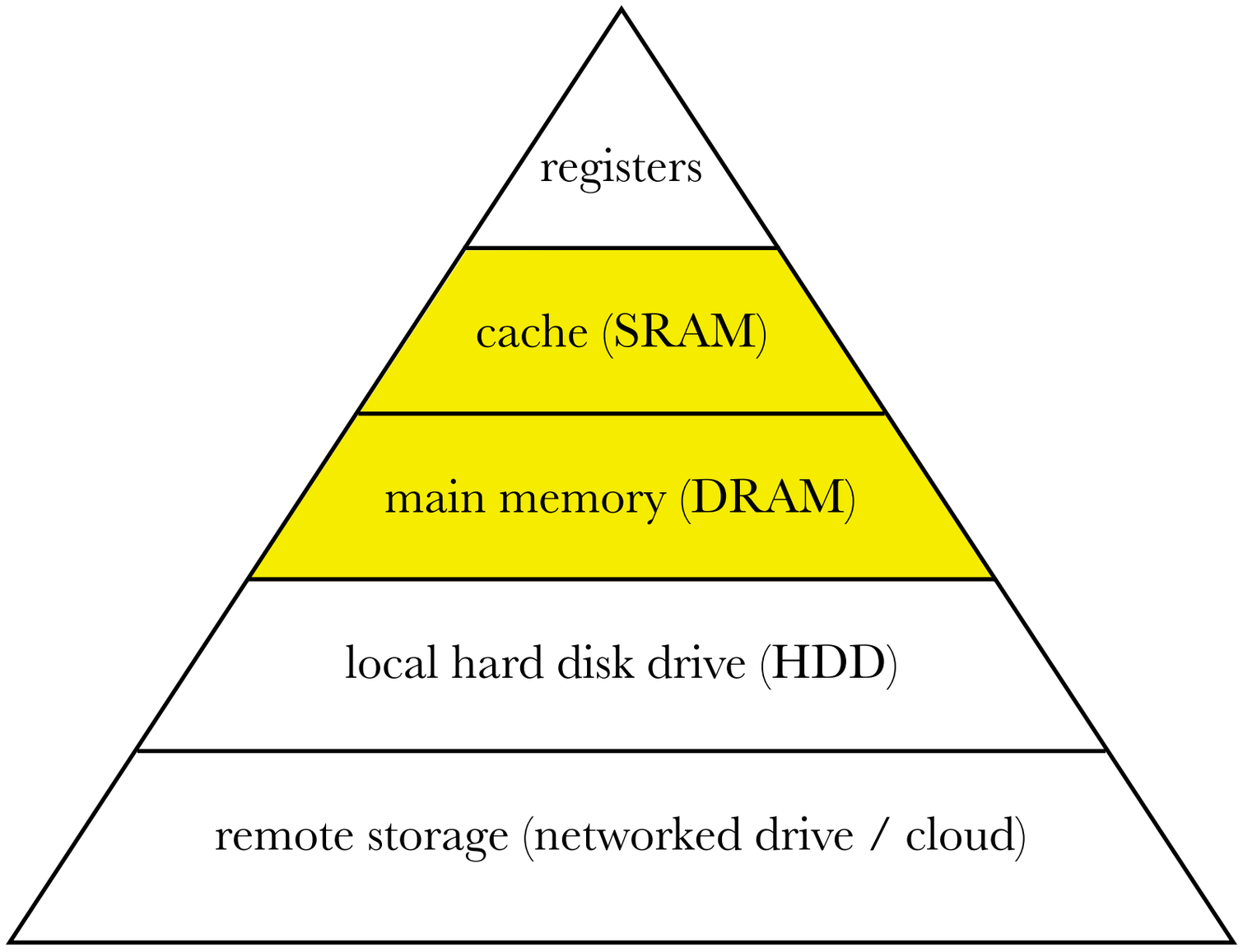
would like:

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



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



boundary 1: SRAM ⇔ DRAM

§Caching

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

cache |kaSH|

verb

store away in hiding or for future use.

cache |kaSH|

noun

• (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 ("miss"), fetch from DRAM

essential issues:

1. *what* data to cache
2. *where* to store cached data;
   i.e., how to *map* address $k \rightarrow$ 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:

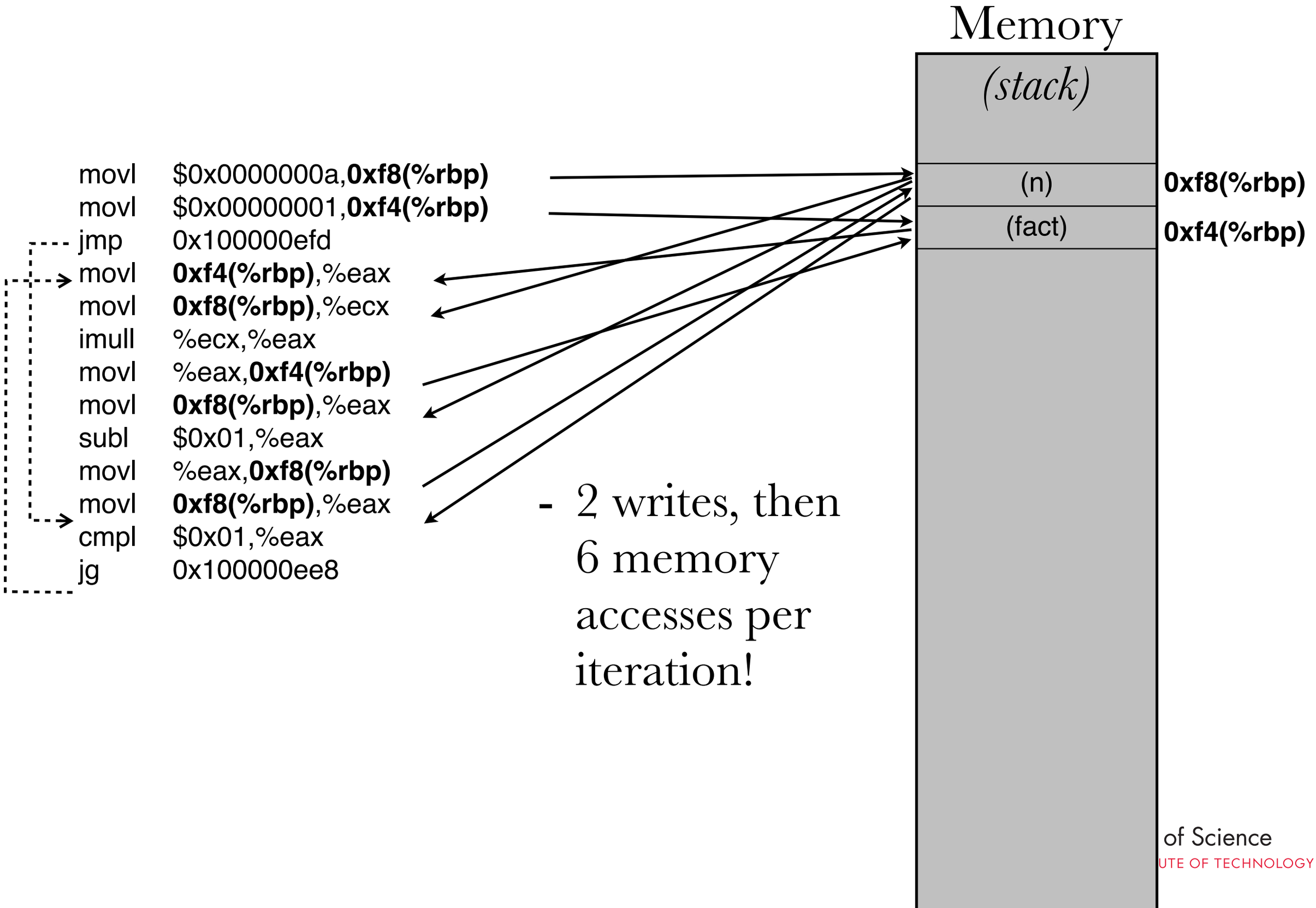
- if a datum was accessed recently, it's likely to be accessed again soon

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

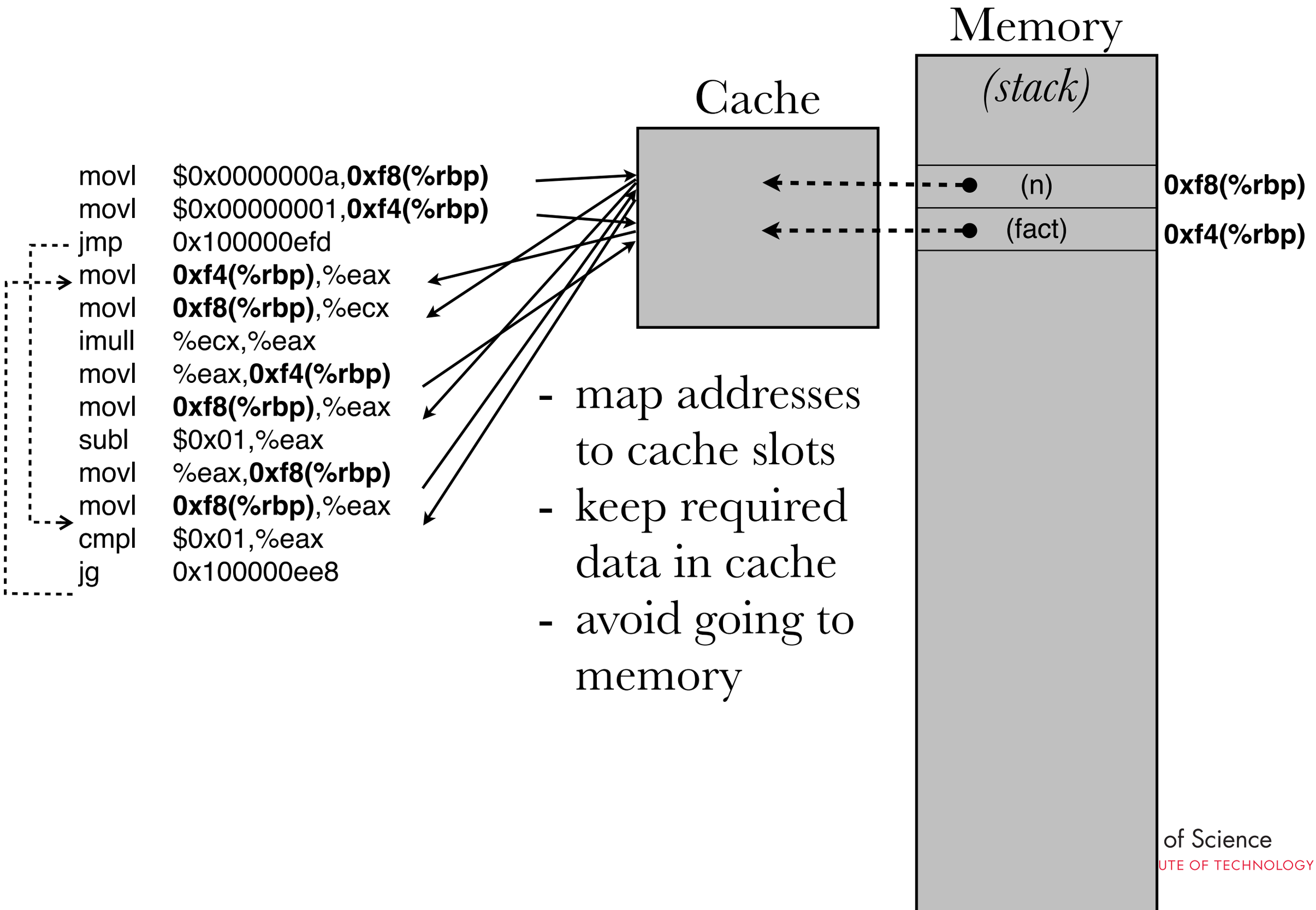
```
main() {
    int n = 10;
    int fact = 1;
    while (n>1) {
        fact = fact * n;
        n = n - 1;
    }
}
```

```
        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
```

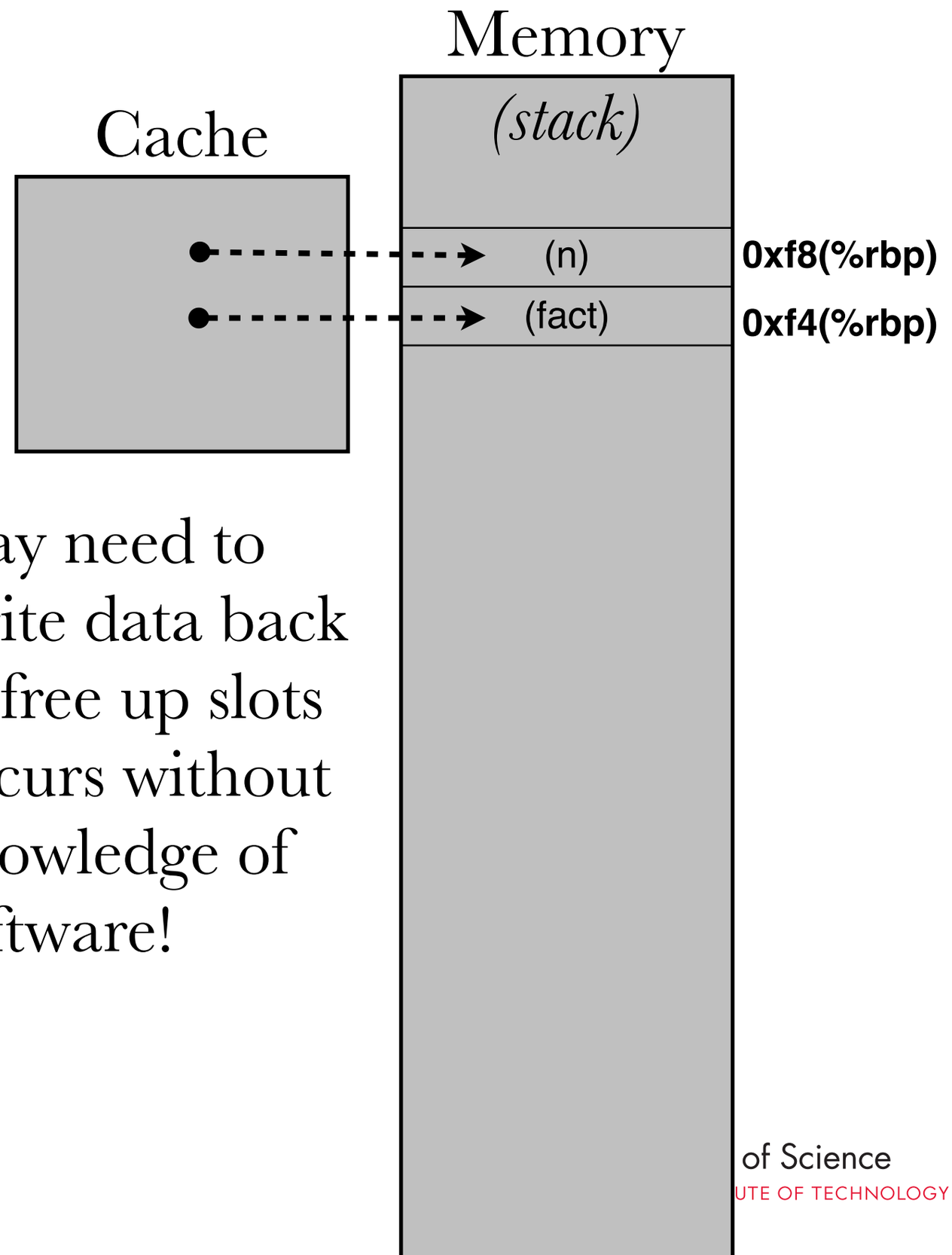
(memory references in bold)



- 2 writes, then 6 memory accesses per iteration!


Memory
(stack)
Cache
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
(n)
(fact)
0xf8(%rbp)
0xf4(%rbp)
- 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*!

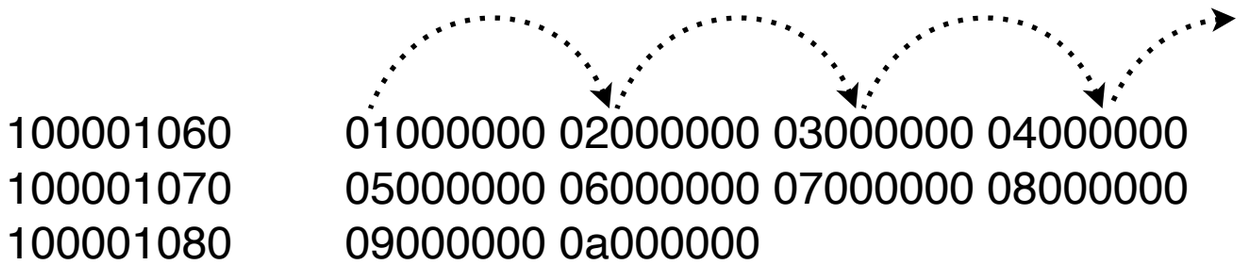
b. **spatial** (location-based) locality:

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

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

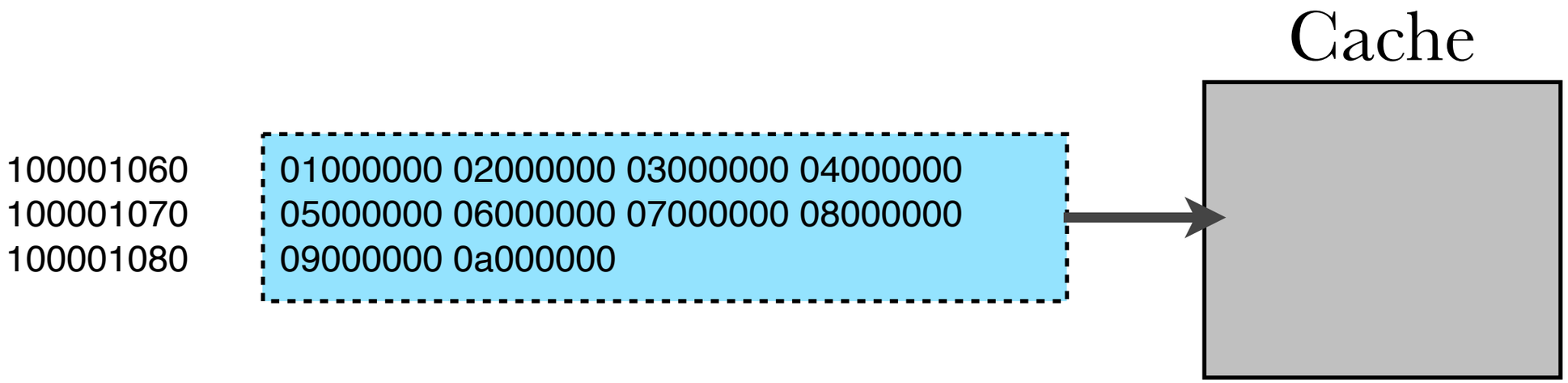
```
int arr[] = {1, 2, 3, 4, 5,
             6, 7, 8, 9, 10};

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

stride length = 1 **int** (4 bytes)




```
    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

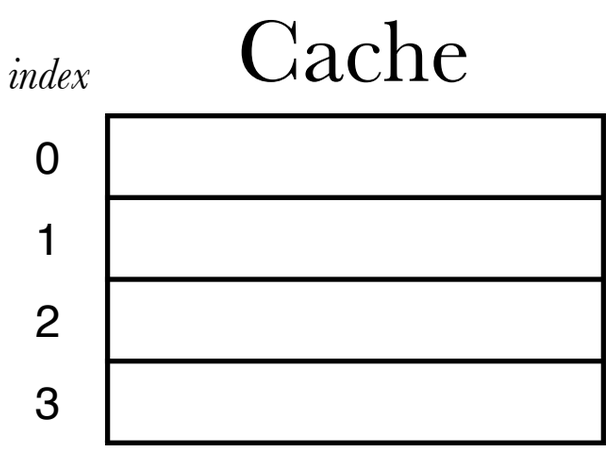


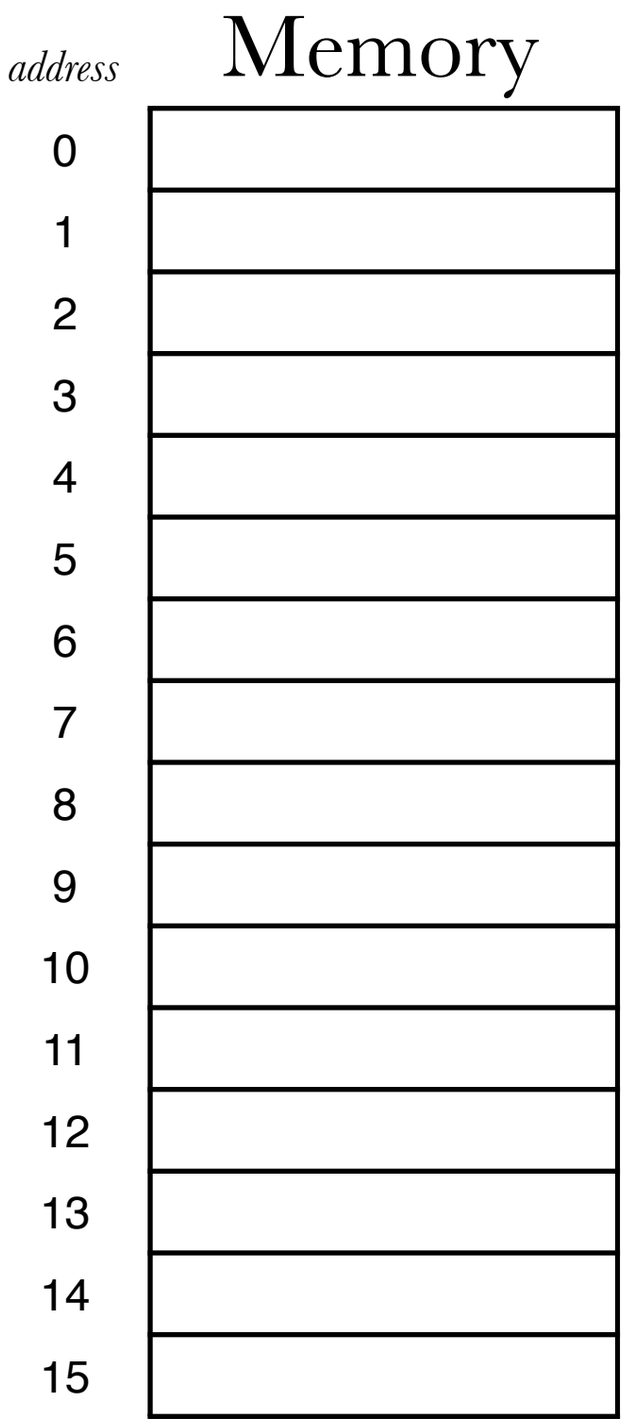
idea: transfer array from memory to cache on accessing *first item*, then only access cache!

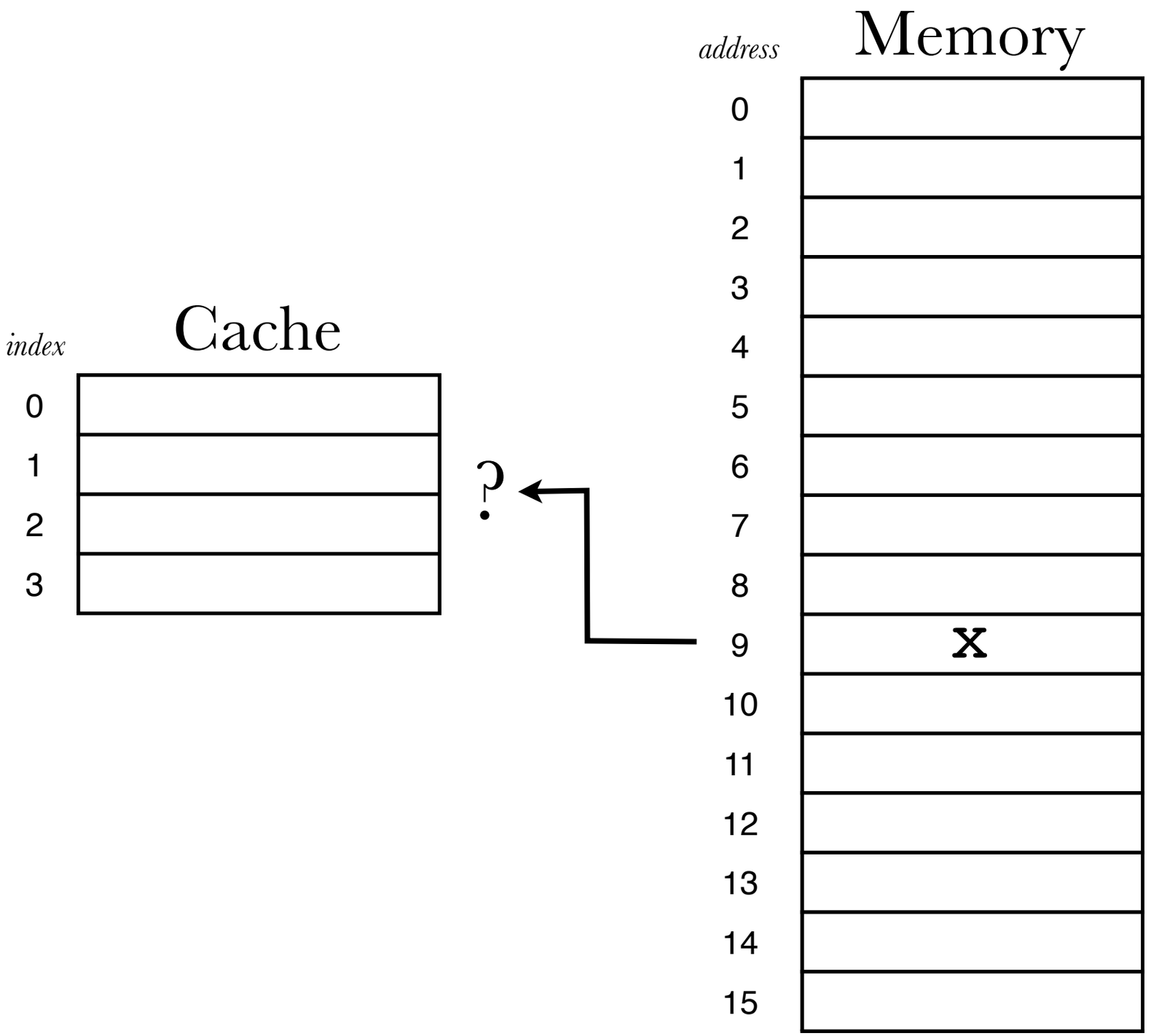
2. *where* to store cached data?

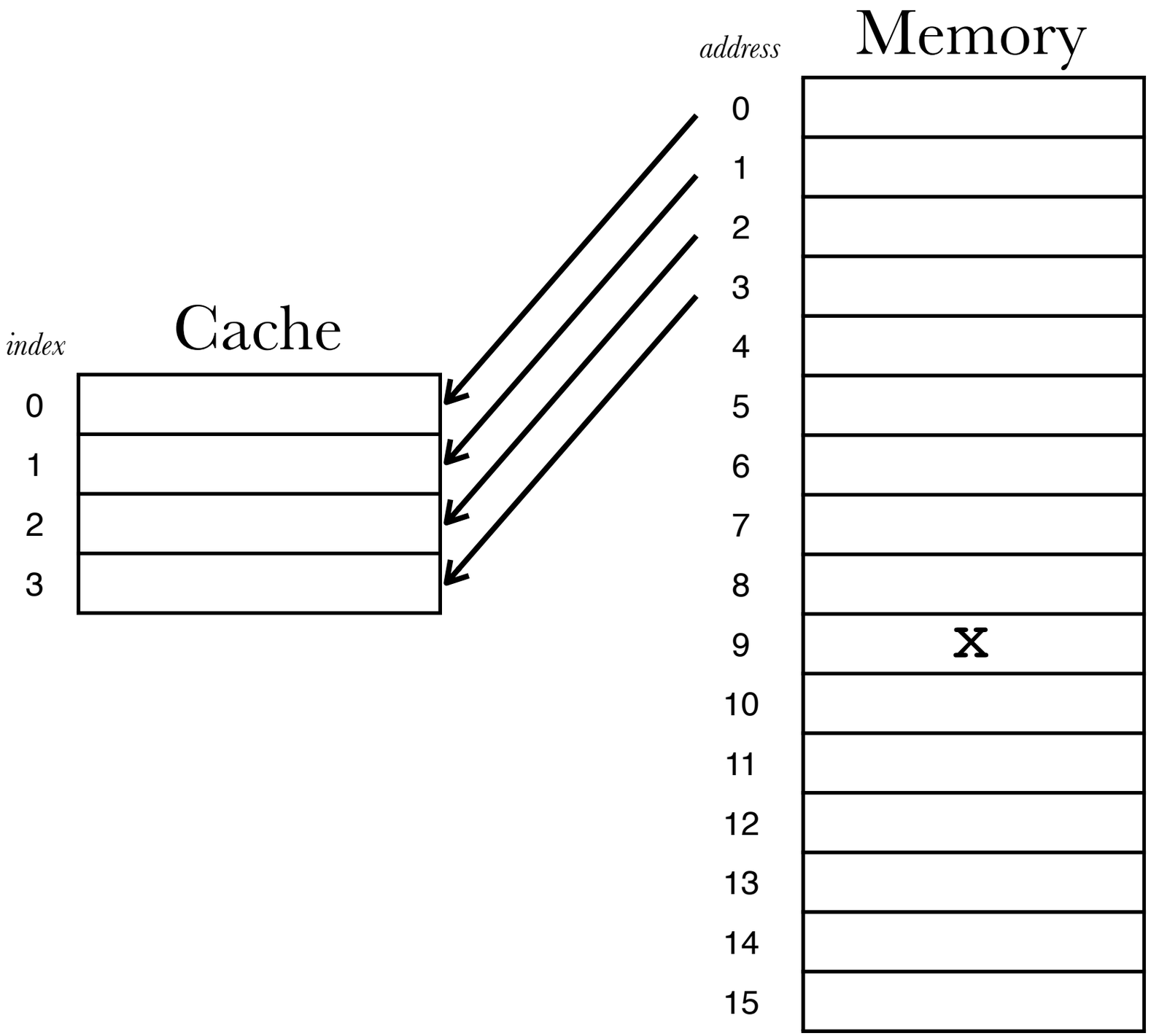
i.e., how to *map* address $k \rightarrow$ cache slot

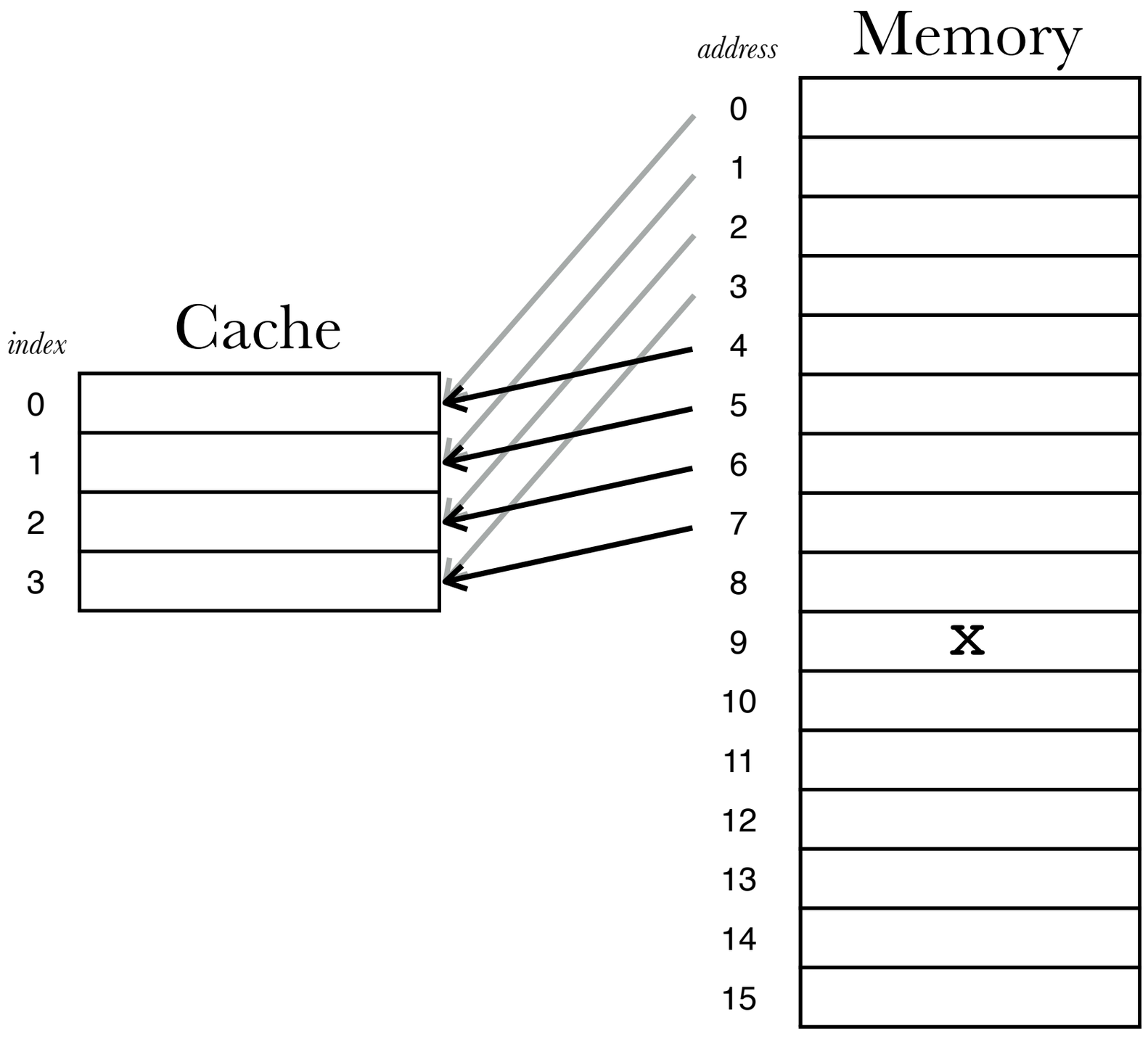
§Cache Organization

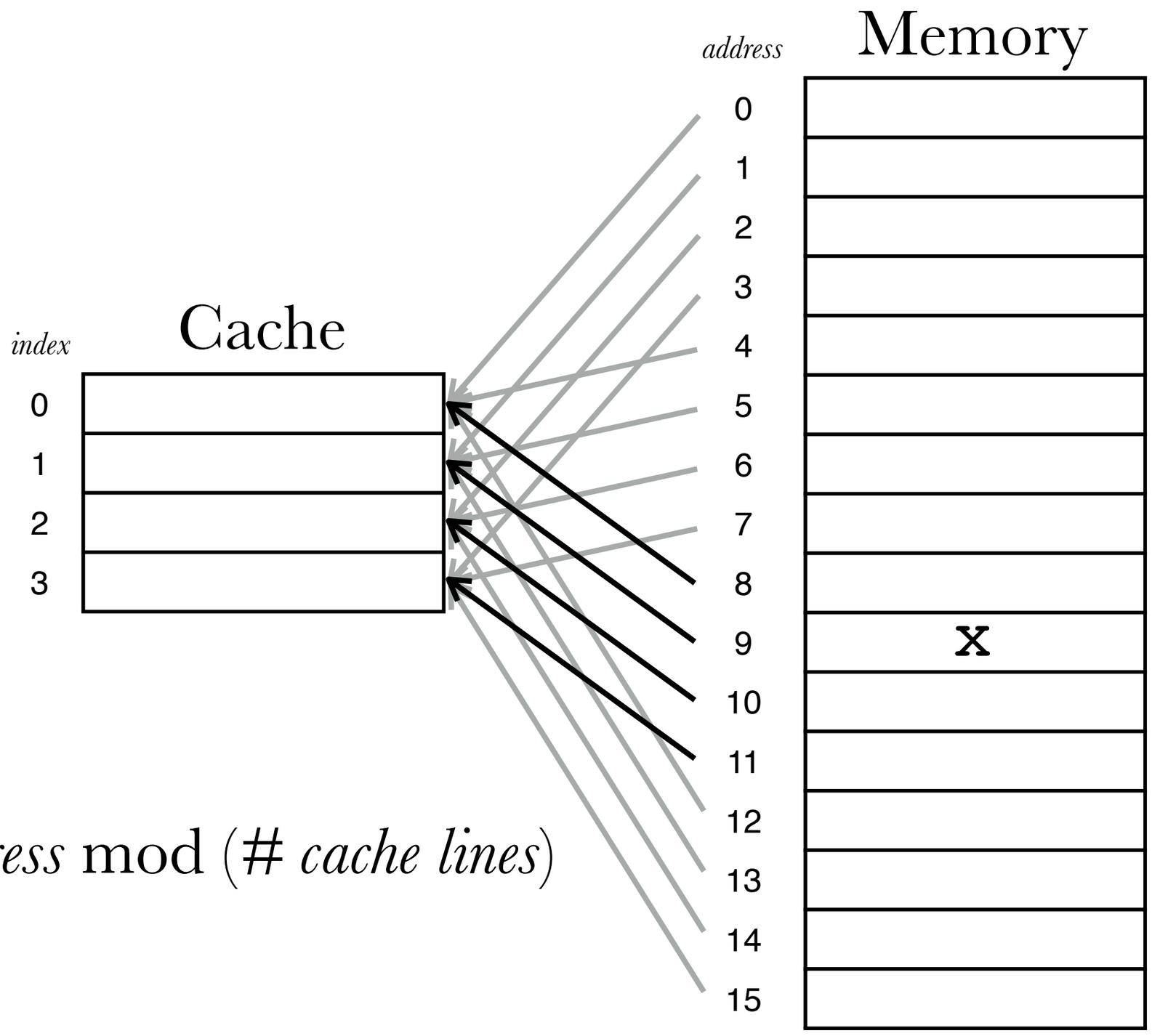

Cache
index
0
1
2
3


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

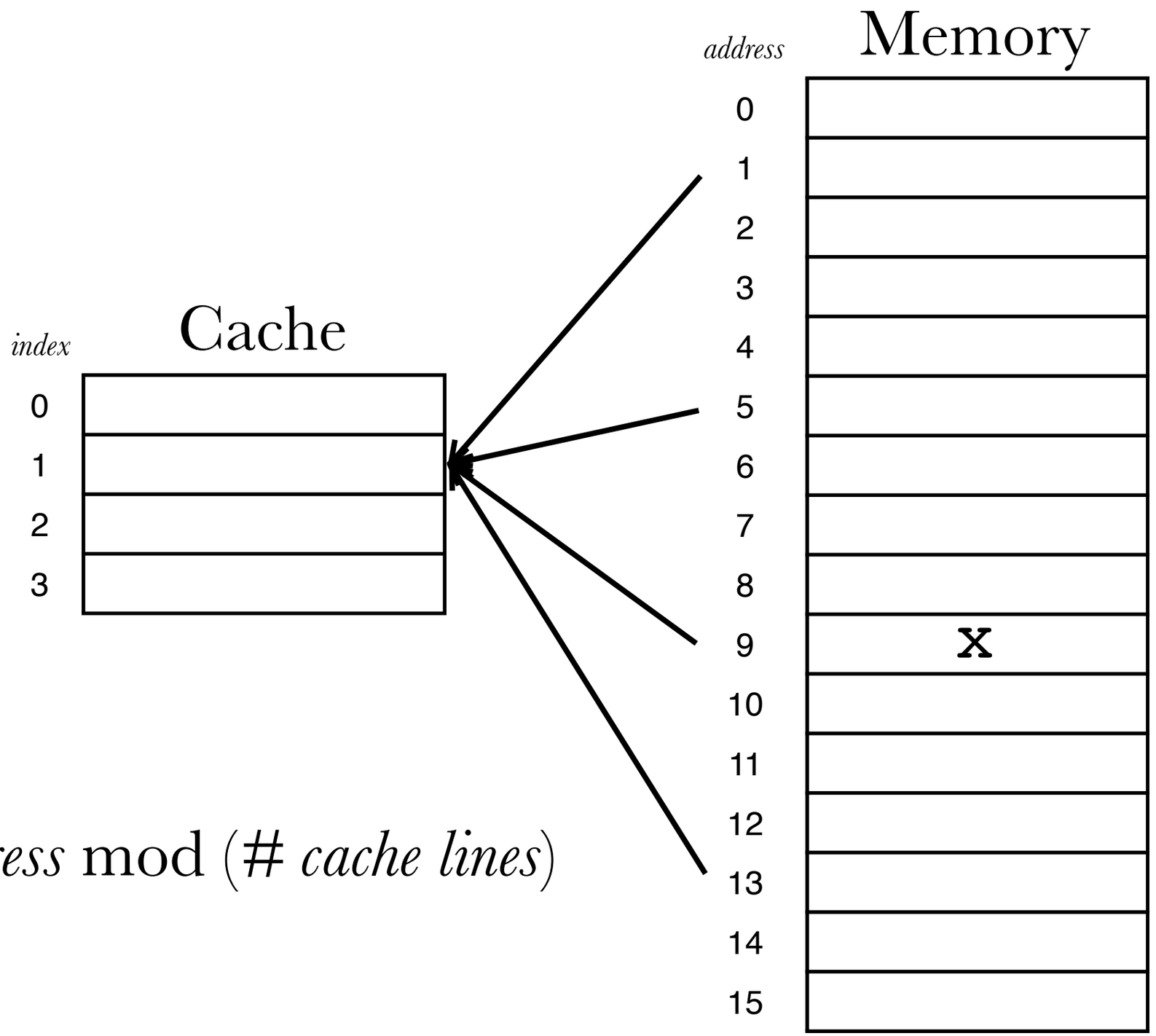

Cache
index
0
1
2
3
?
Memory
address
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
X


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

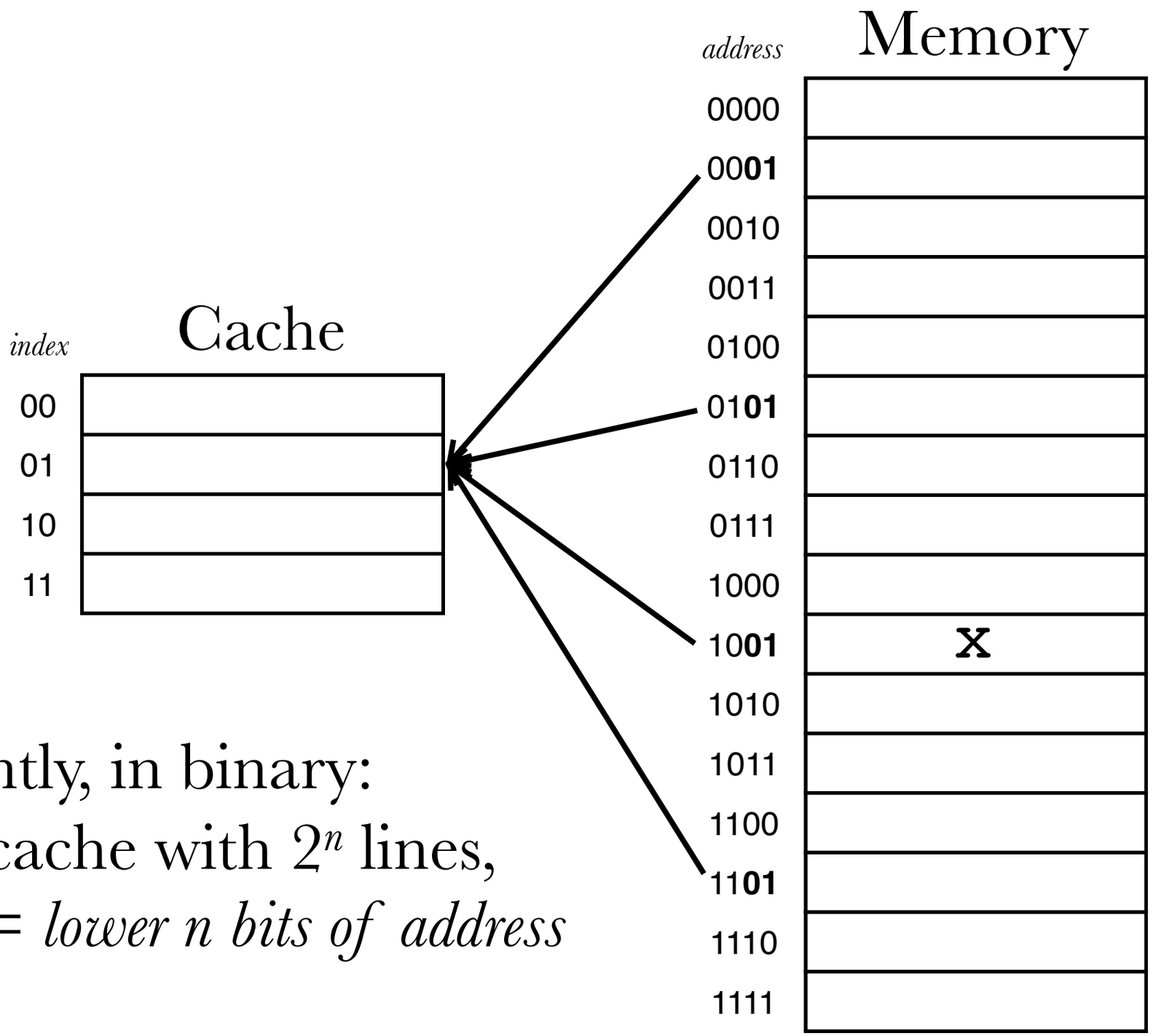

Memory
address
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
X
Cache
index
0
1
2
3



$$index = address \bmod (\#\ cache\ lines)$$



$$index = address \bmod (\#\ cache\ lines)$$

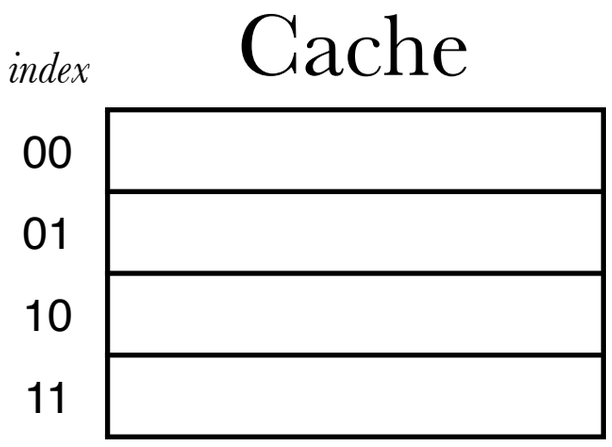


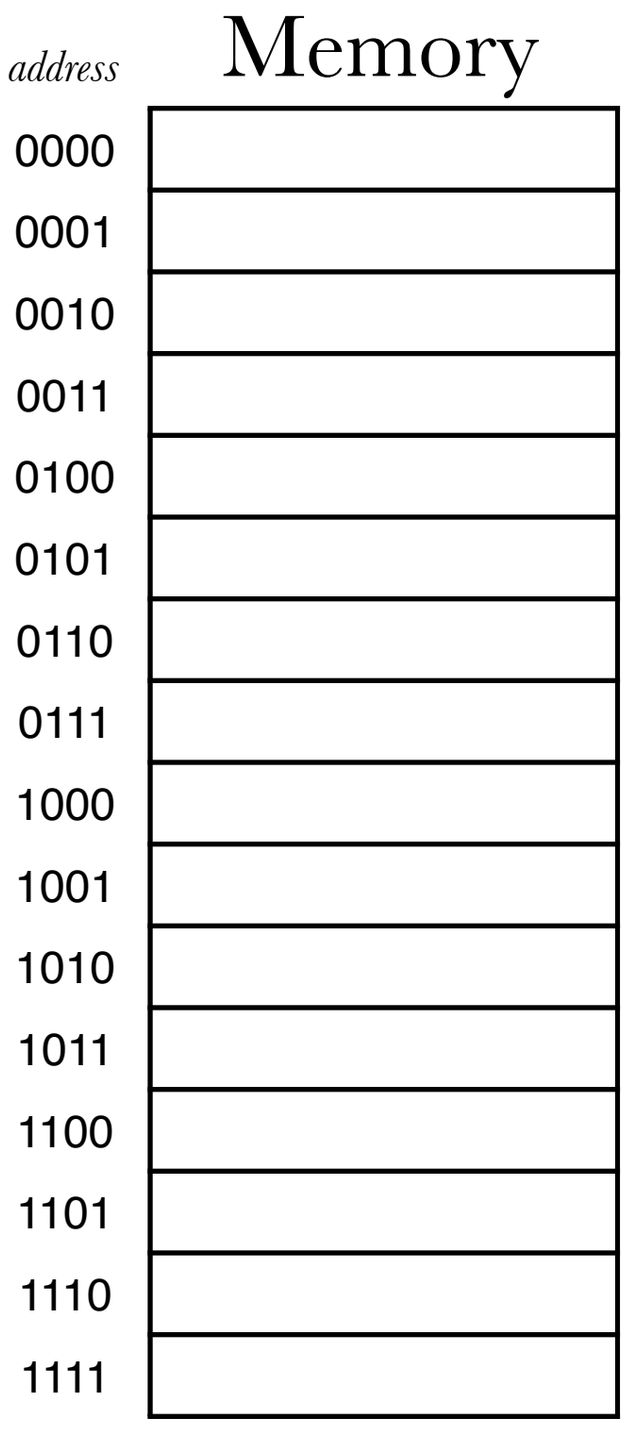
equivalently, in binary:

for a cache with 2^n lines,

index = lower n bits of address

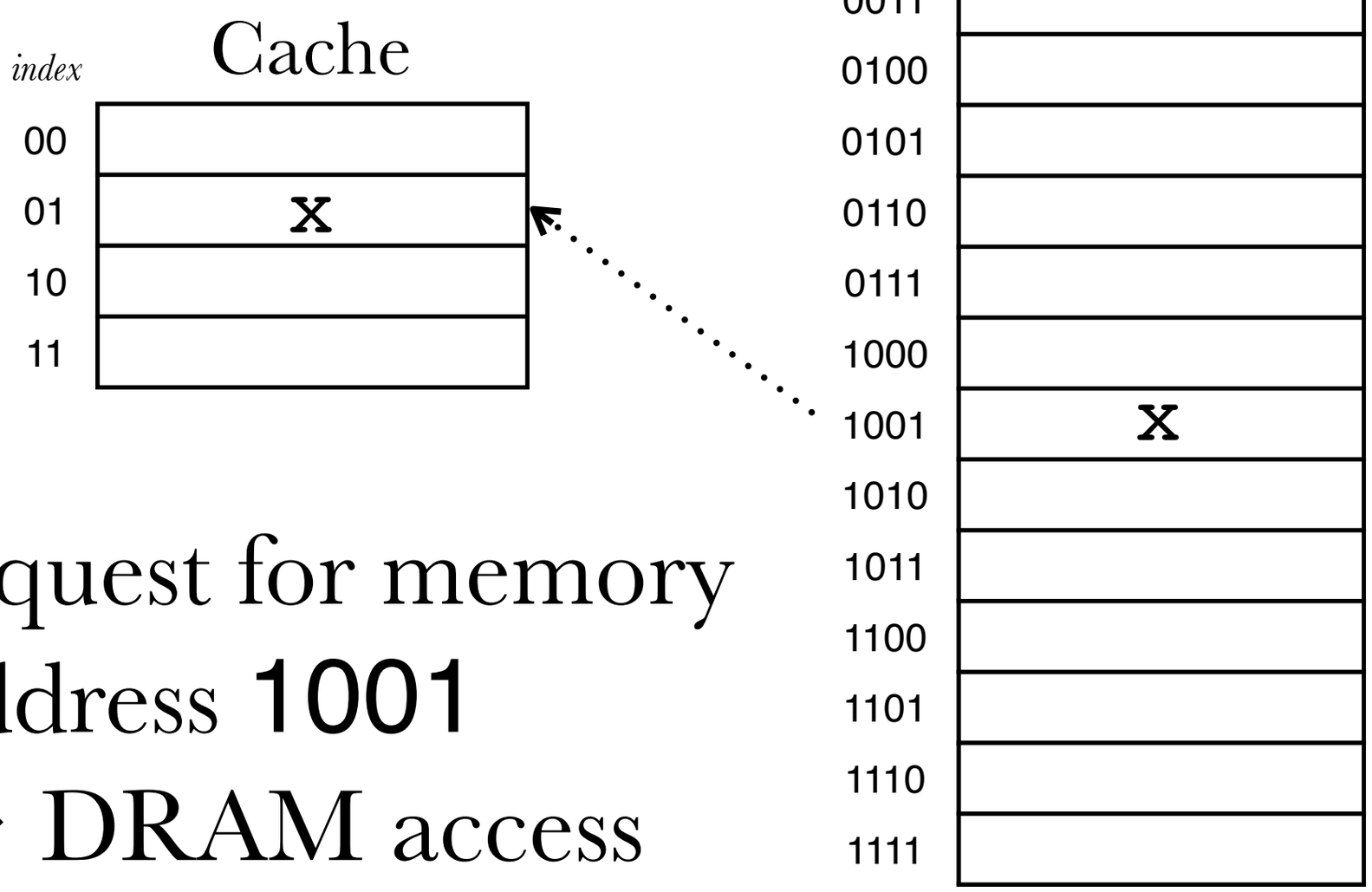
1) **direct** mapping





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

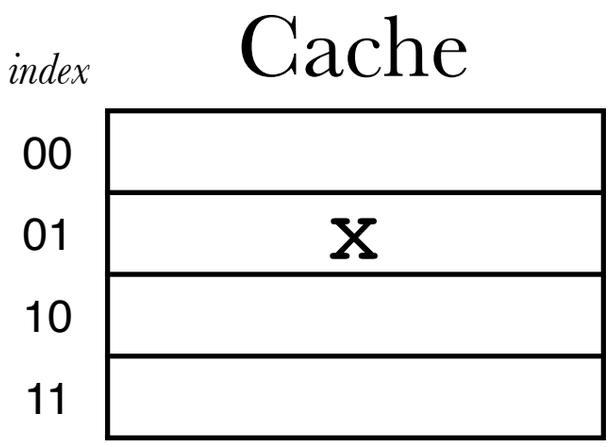
1) **direct** mapping

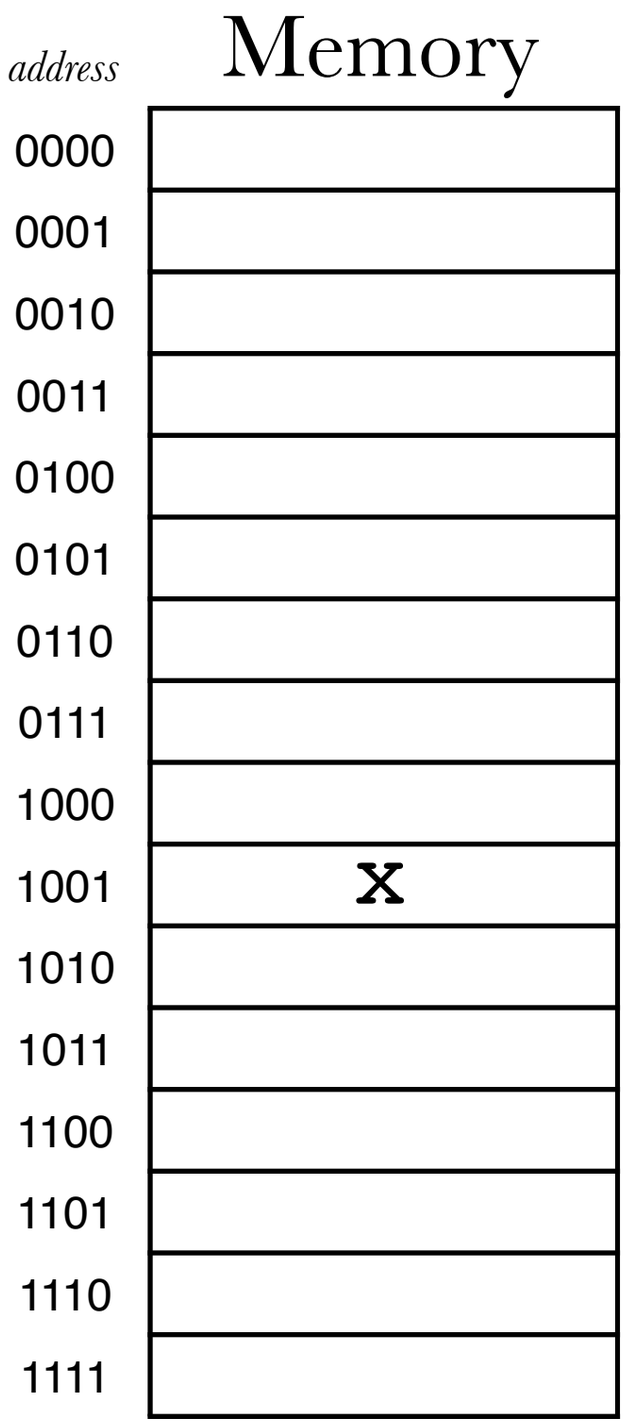


e.g., request for memory address **1001**

→ DRAM access

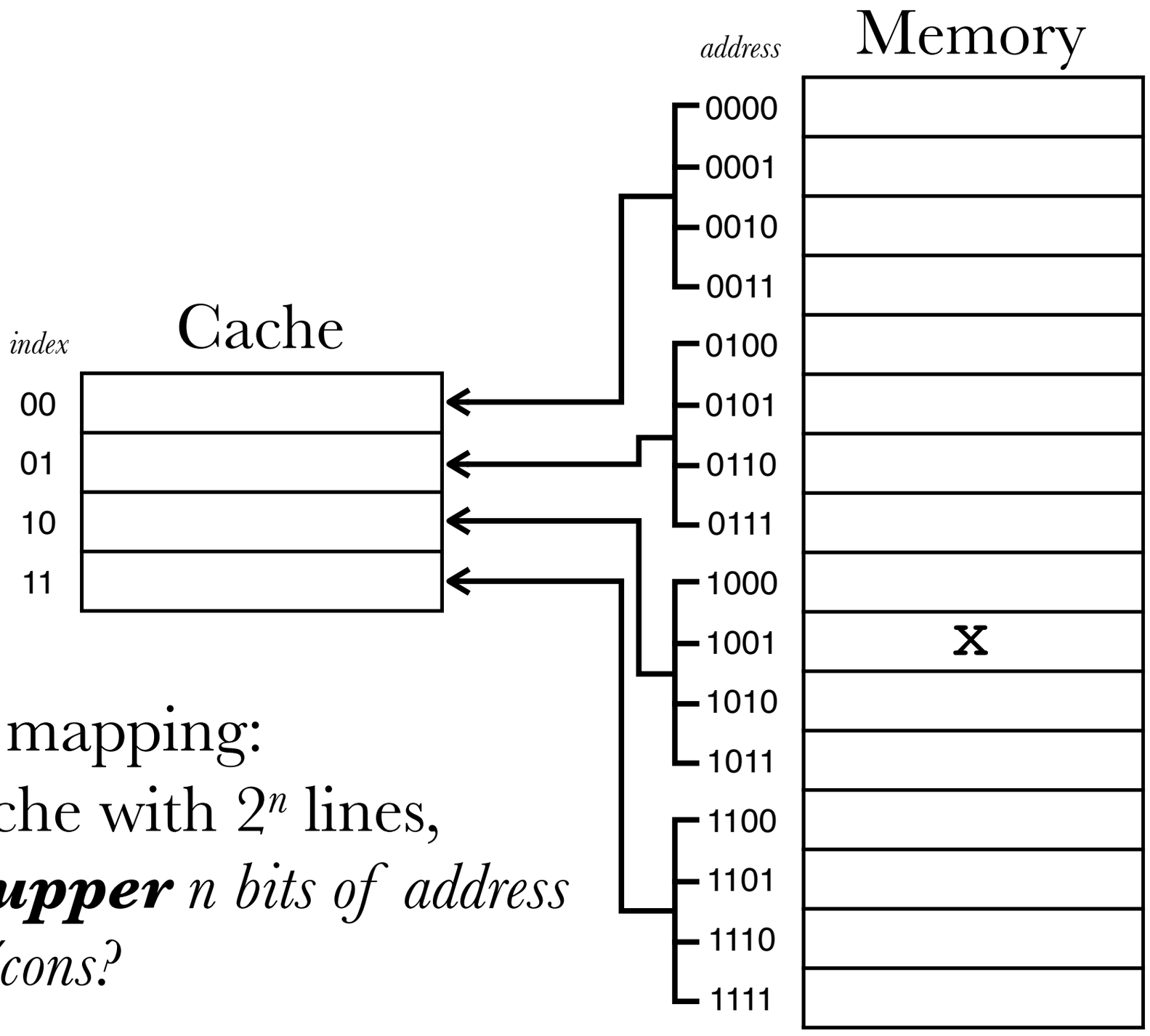
1) **direct** mapping





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

→ cache “hit”

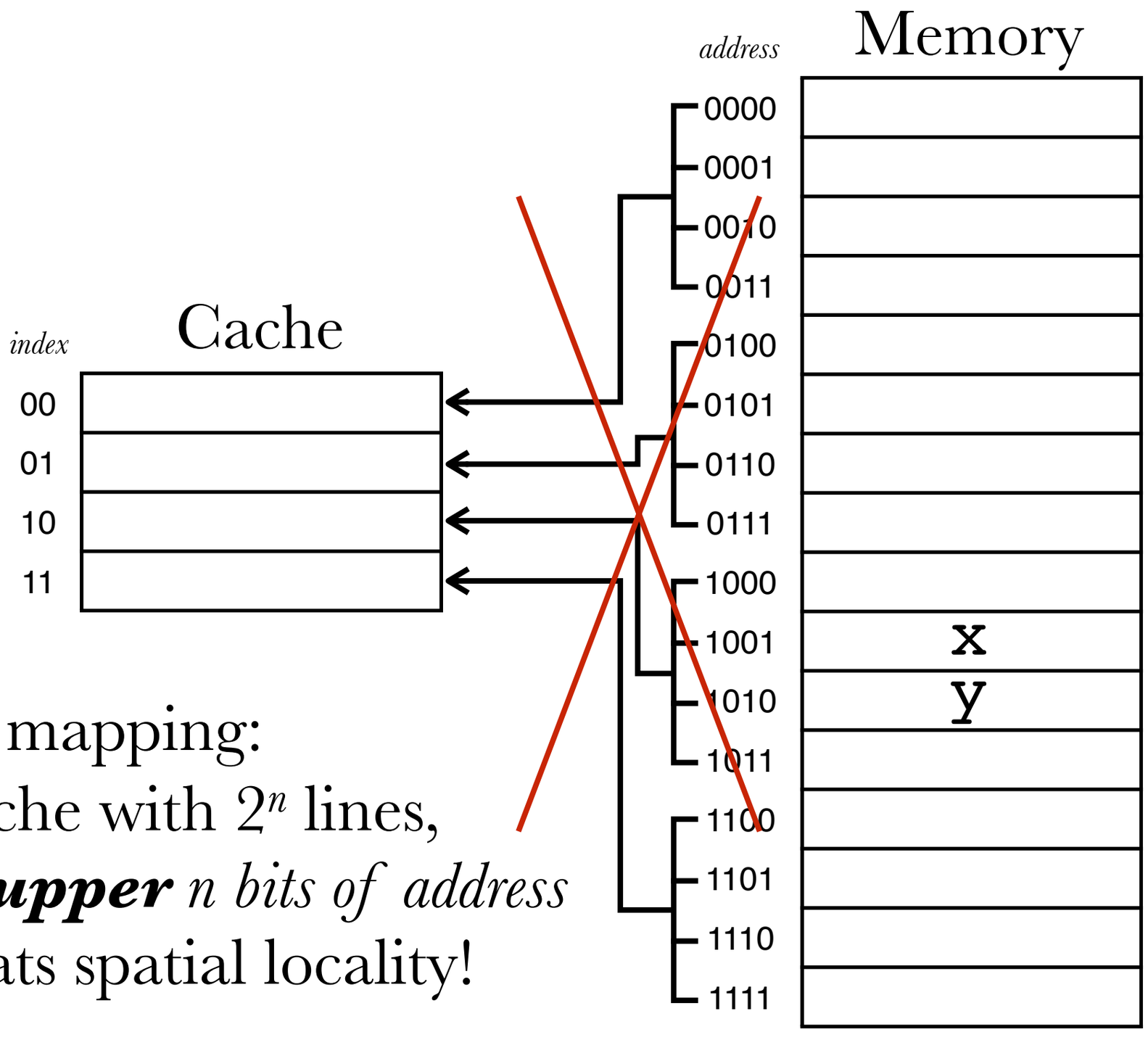


alternative mapping:

for a cache with 2^n lines,

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

— *pros/cons?*



vie for the same line ("cache collision")

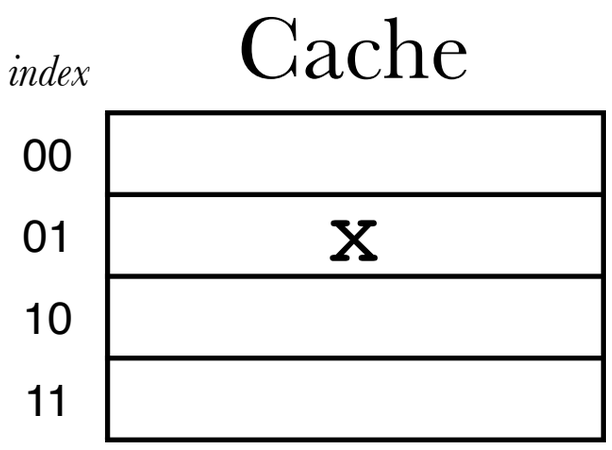
alternative mapping:

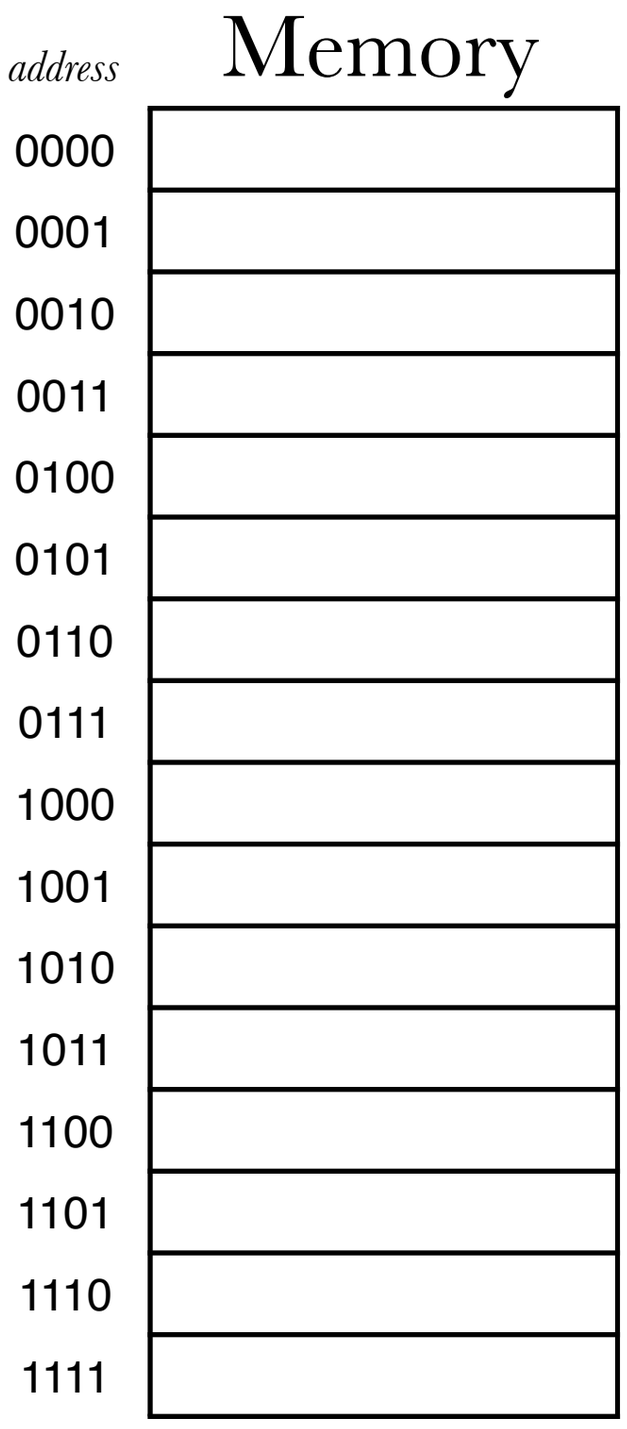
for a cache with 2^n lines,

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

— defeats spatial locality!

1) **direct** mapping





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

1) **direct** mapping

Cache

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

Memory

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

must add some fields

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

1) **direct** mapping

Cache

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

10|01

Memory

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

i.e., **x** “belongs to” address **1001**

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

Memory

address	
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	
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	
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?

Memory

address	
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**?

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

Memory

address	
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 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

Memory

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

what if new request arrives for **1000**?

- *evict* old mapping to make room for new

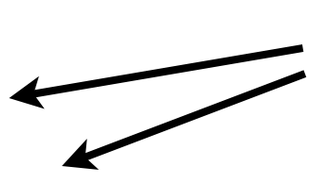
1) **direct** mapping

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

- motivated by temporal locality

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;
    }
}
```



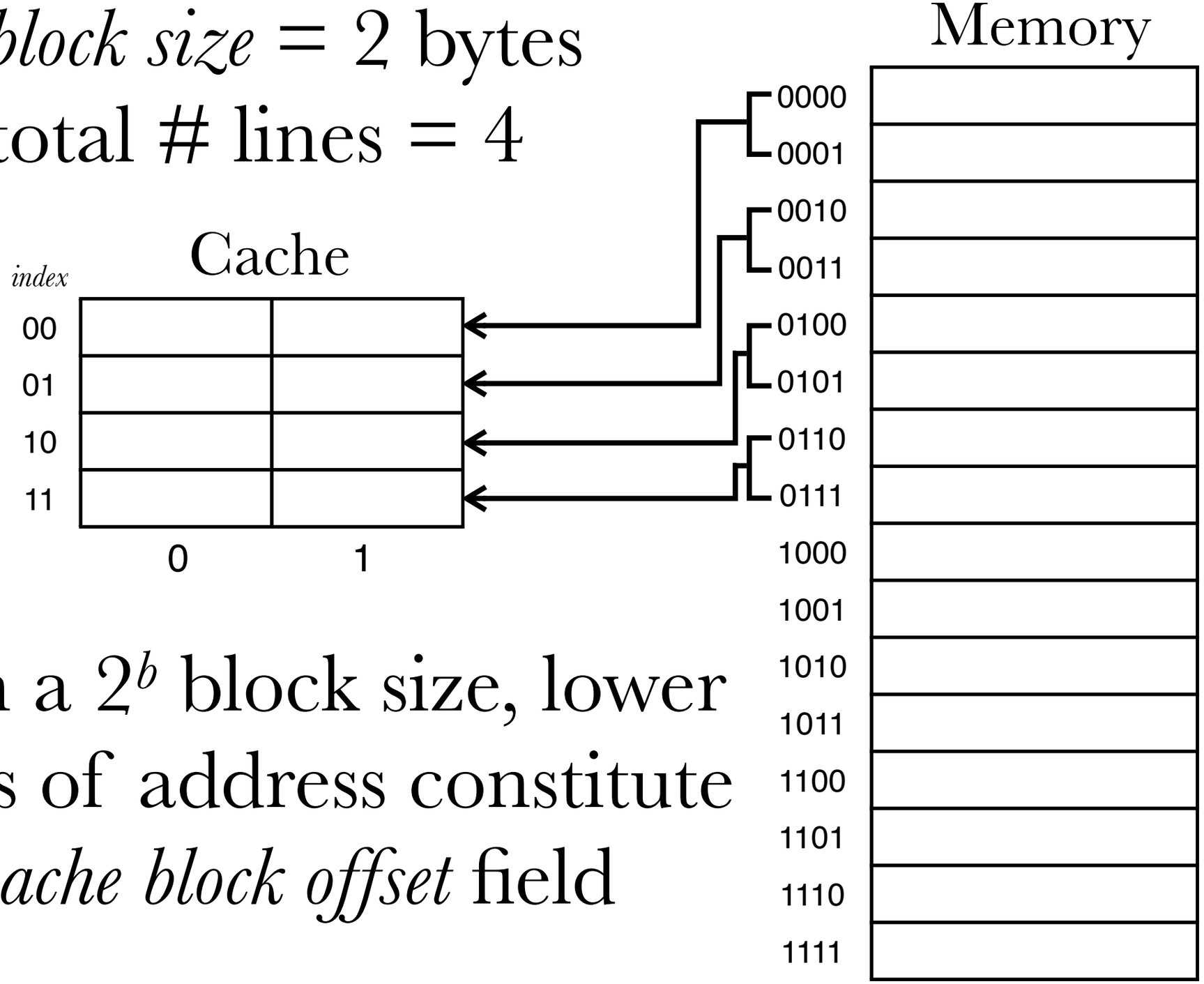
But we frequently deal with > 1 byte of data at a time (e.g., words)

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

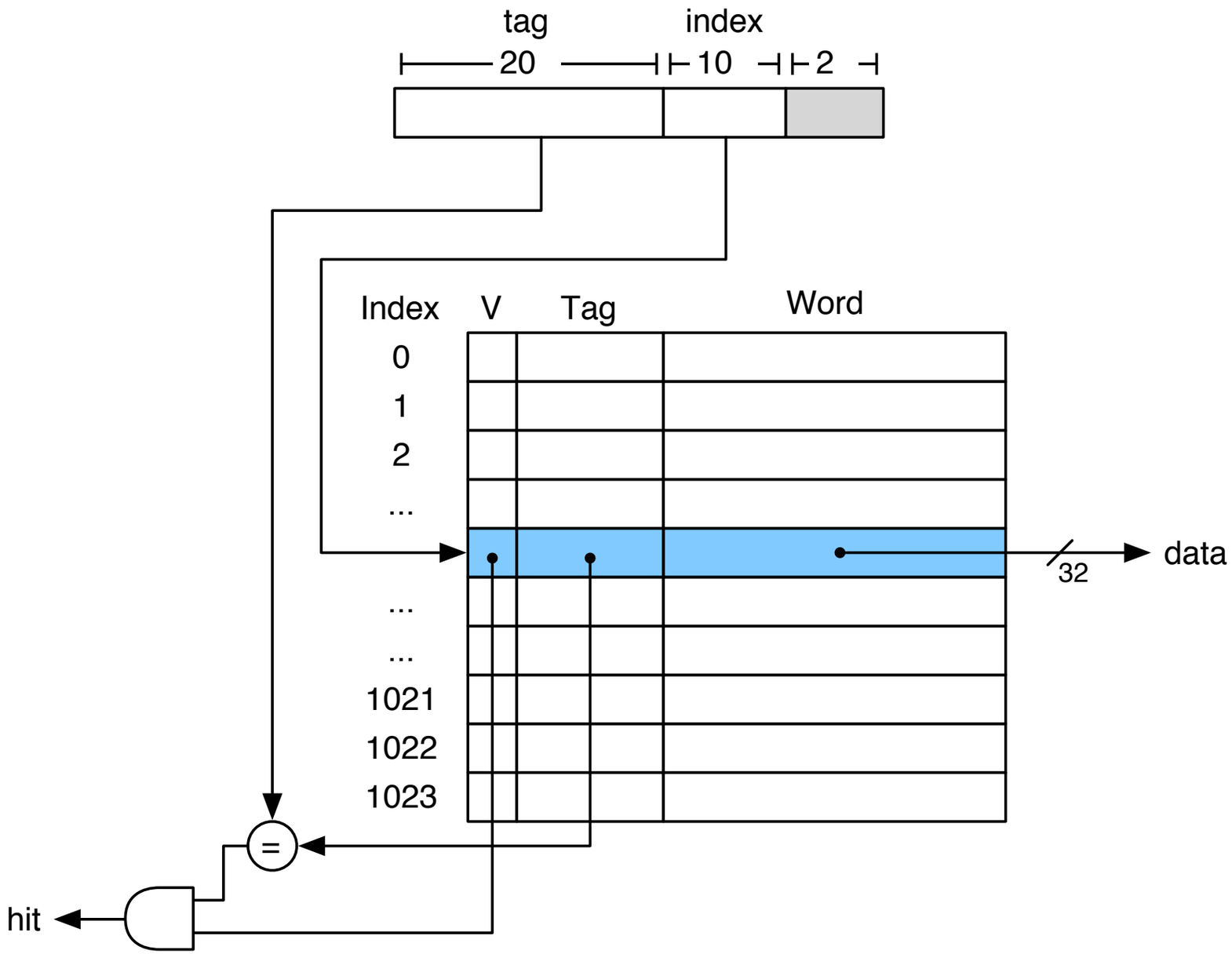
index

$\log_2$(# *lines*) bits wide

block offset

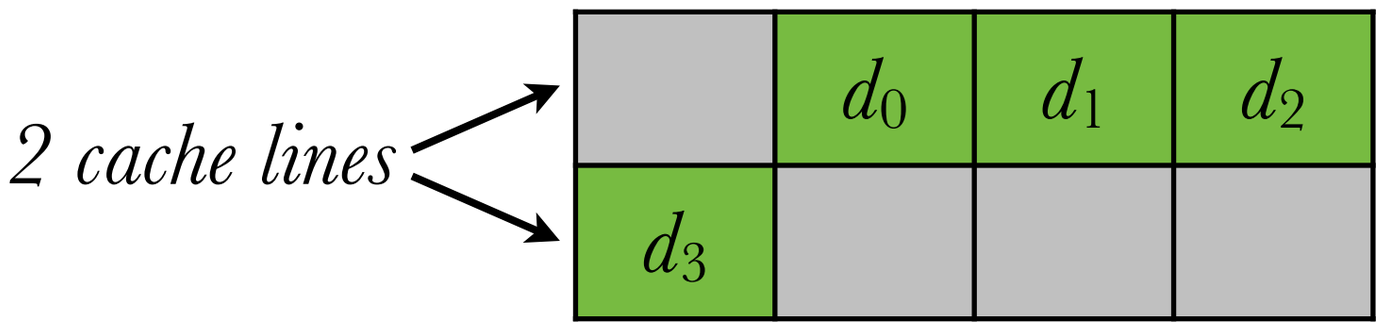
$\log_2$(*block size*) bits wide

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



To maximize cache utilization, multi-byte data may be *aligned* to start at addresses that are multiples of the block size.

Otherwise, data may unnecessarily straddle multiple blocks!



In practice, data are typically aligned to start at multiples of the datum size — "natural alignment"

E.g., 4-byte words start at addresses 0, 4, 8, 12, …

```
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)

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

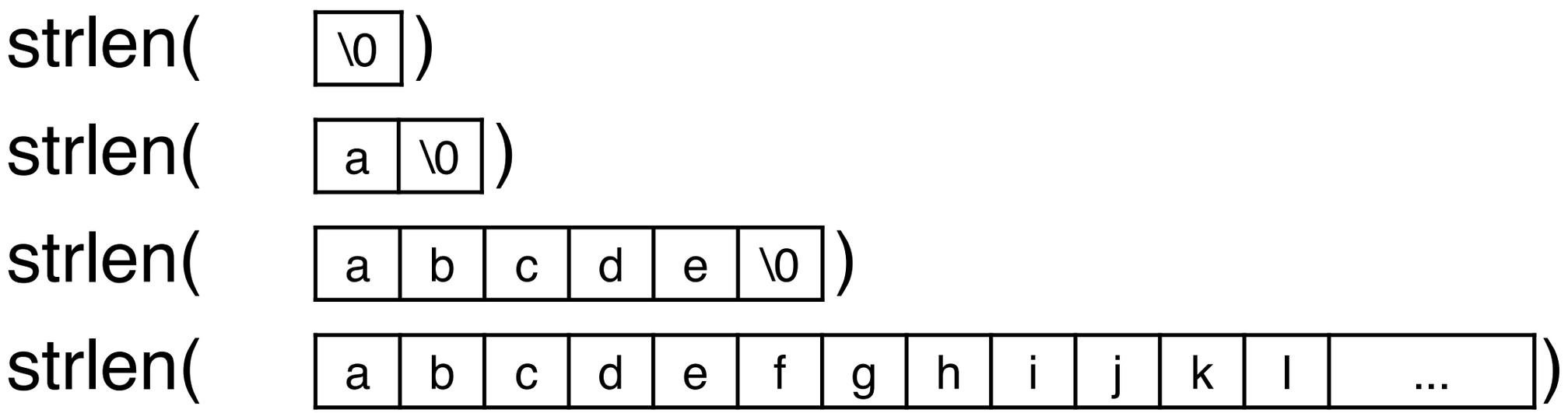
```
strlen:             ; buf in %rdi
pushq    %rbp
movq     %rsp,%rbp
mov    $0x0,%eax    ; result = 0
cmpb   $0x0,(%rdi)  ; if *buf == 0
je     0x10000500   ;   return 0
add    $0x1,%rdi    ; buf += 1
add    $0x1,%eax    ; result += 1
movzbl (%rdi),%edx  ; %edx = *buf
add    $0x1,%rdi    ; buf += 1
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

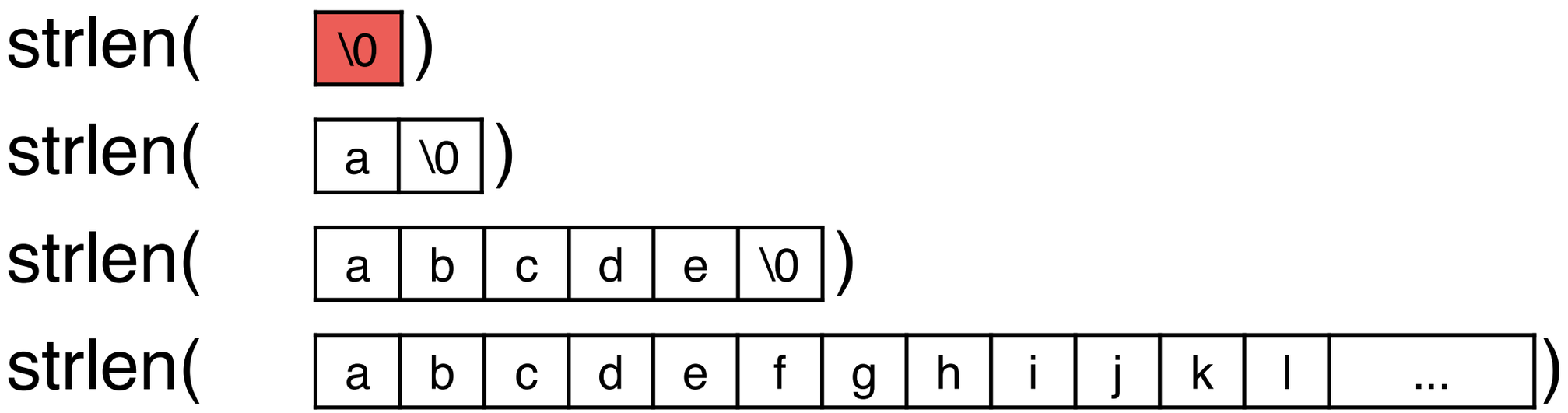
```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
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
```



```
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
```



```
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 ...)

```
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
```

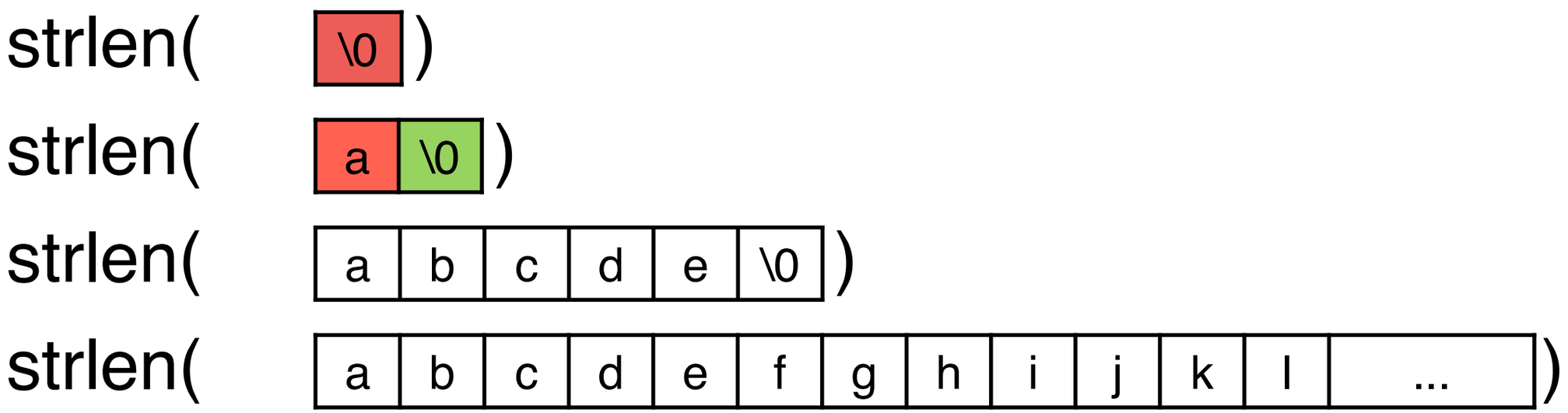
strlen(\0)

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

— simplifying assumption: first byte of buf is aligned

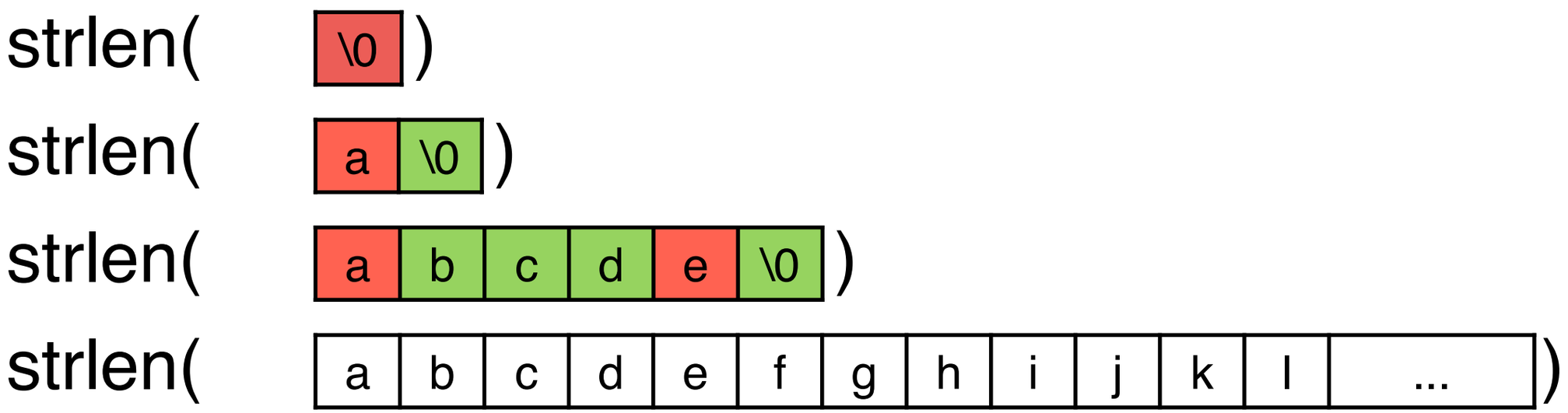
```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
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
```



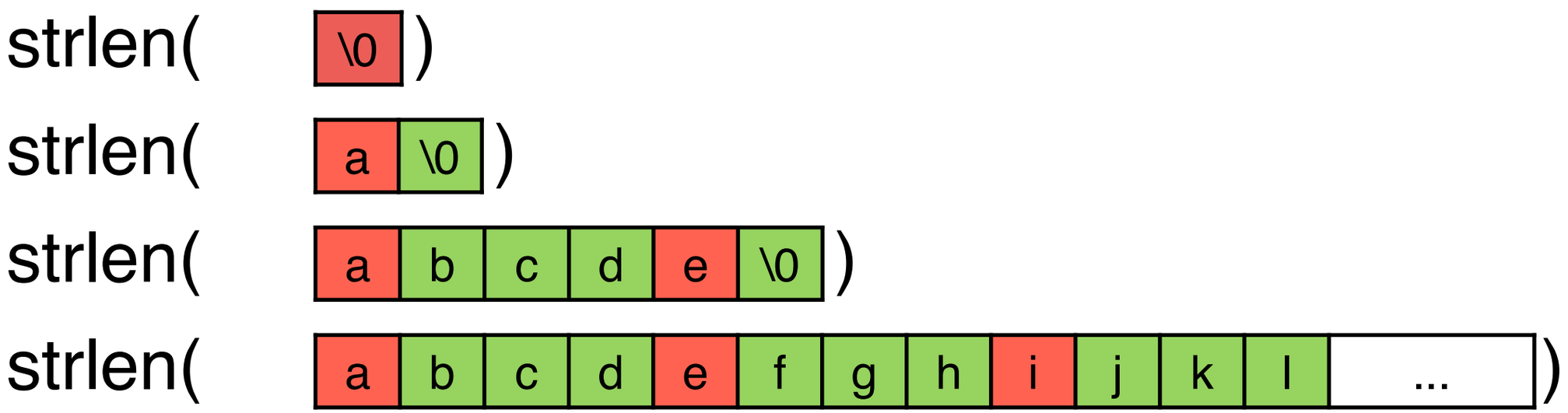
```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
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
```



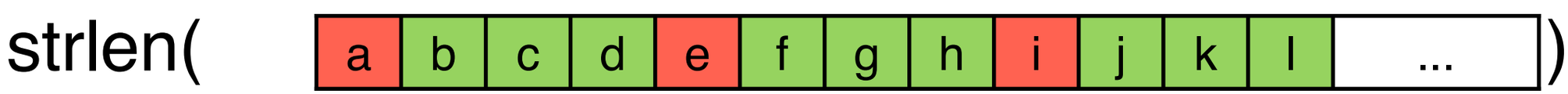
```
int strlen(char *buf) {
    int result = 0;
    while (*buf++)
        result++;
    return result;
}
```

```
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
```



```
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
```



In the long run, hit rate = ¾ = 75%

```
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)

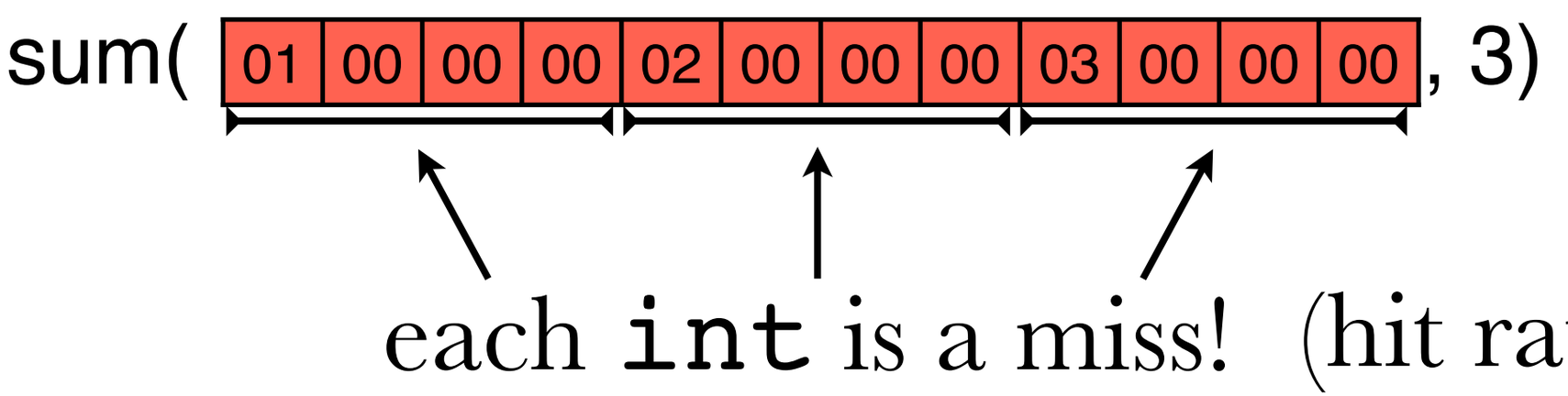
```
int sum(int *arr, int n) {
    int i, r = 0;
    for (i=0; i<n; i++)
        r += arr[i];
    return r;
}
```

```
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)

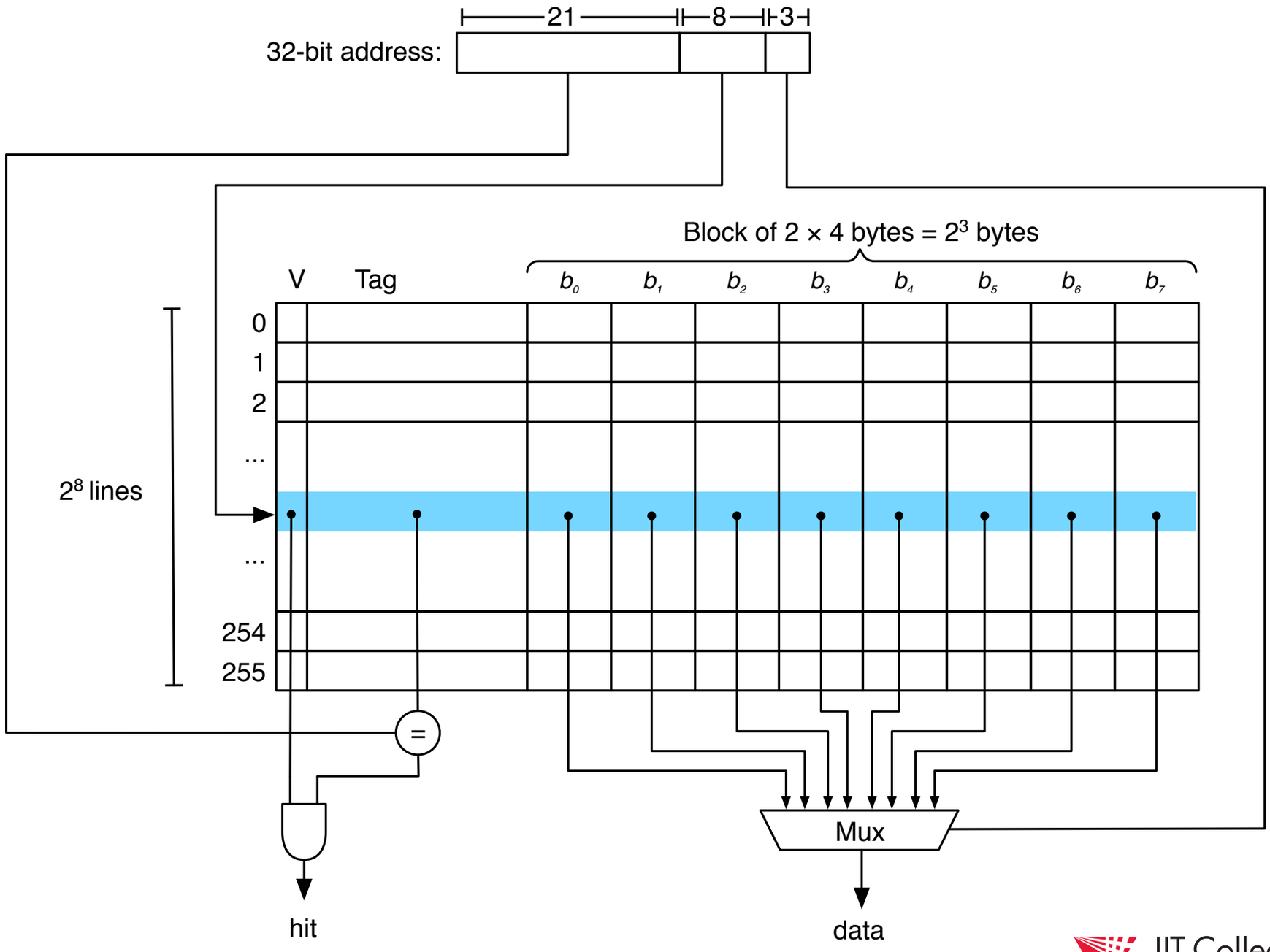
```
int sum(int *arr, int n) {
    int i, r = 0;
    for (i=0; i<n; i++)
        r += arr[i];
    return r;
}
```

```
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
```



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

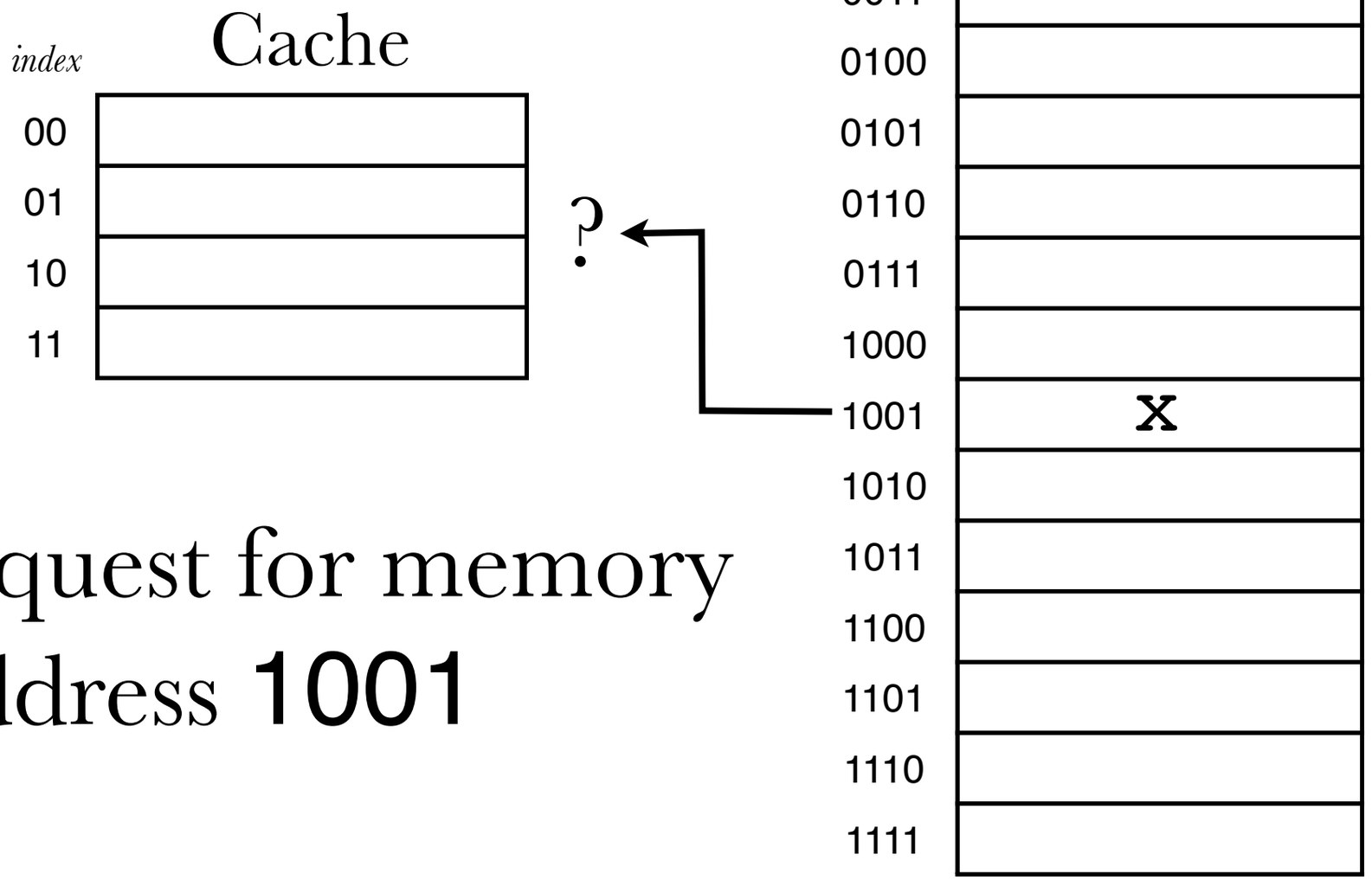
e.g., cache with 2^8 lines of 2 × 4 byte blocks



problem: when a *cache collision* occurs, we must evict the old (direct) mapping

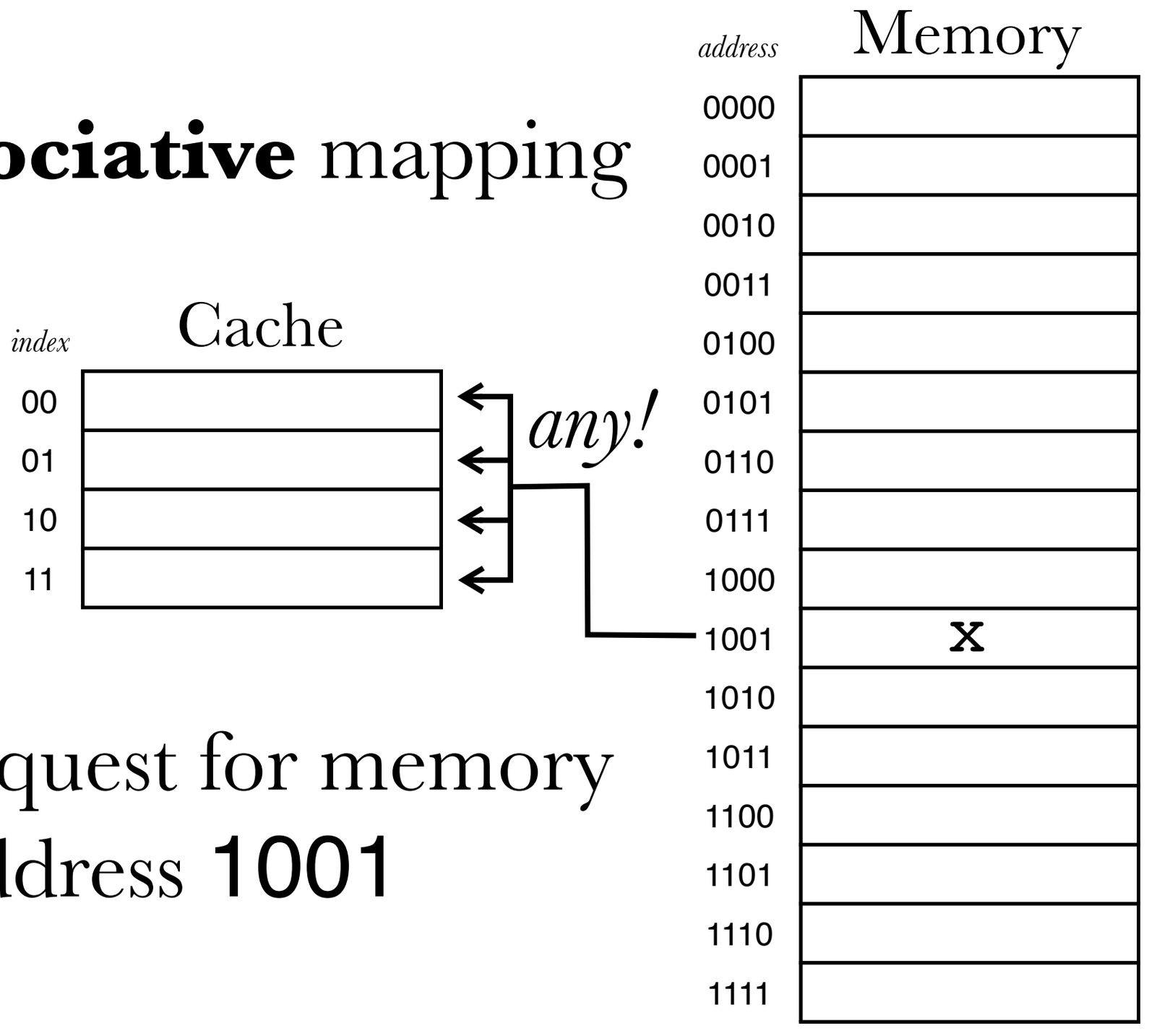
— no way to use a different cache slot

2) **associative** mapping



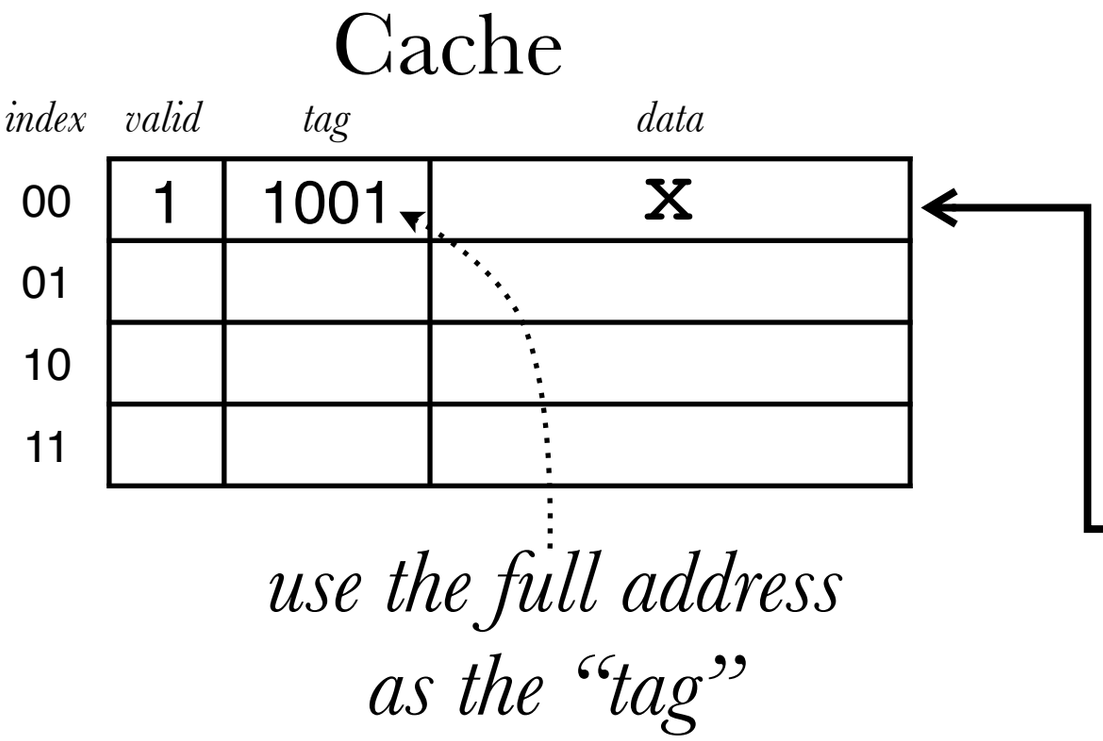
e.g., request for memory address **1001**

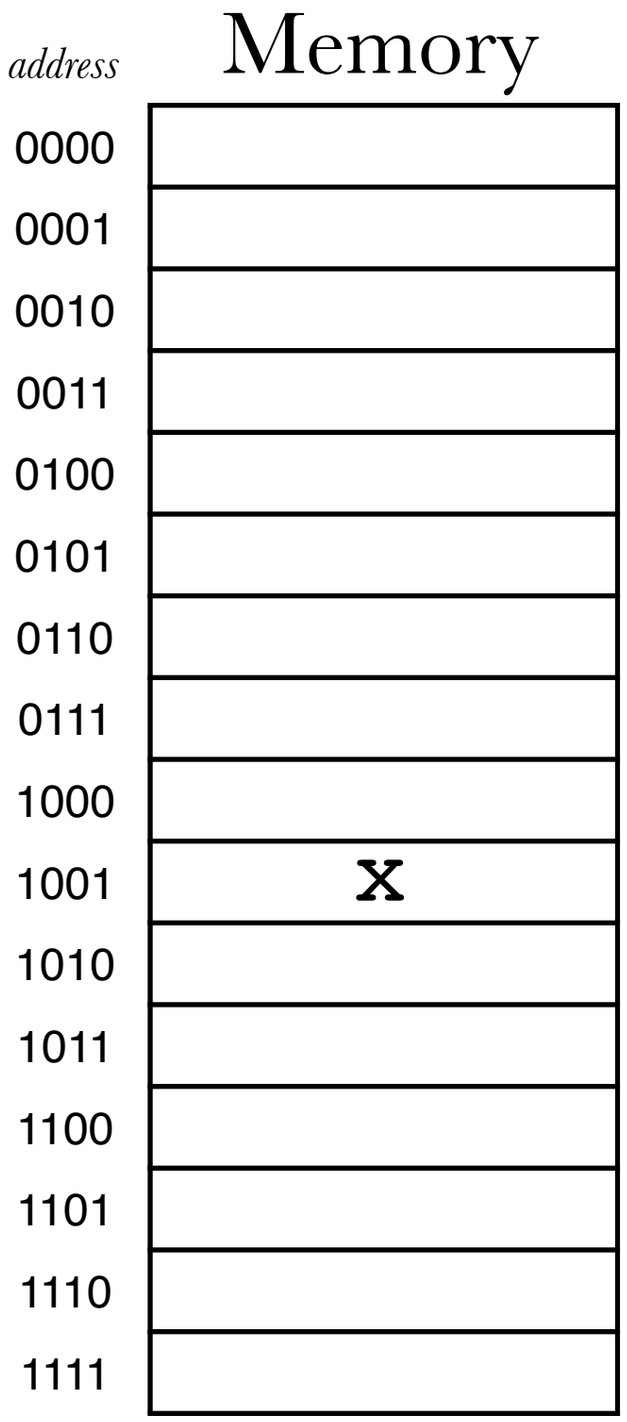
2) **associative** mapping



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

2) **associative** mapping





- effectively a hardware lookup table

2) **associative** mapping

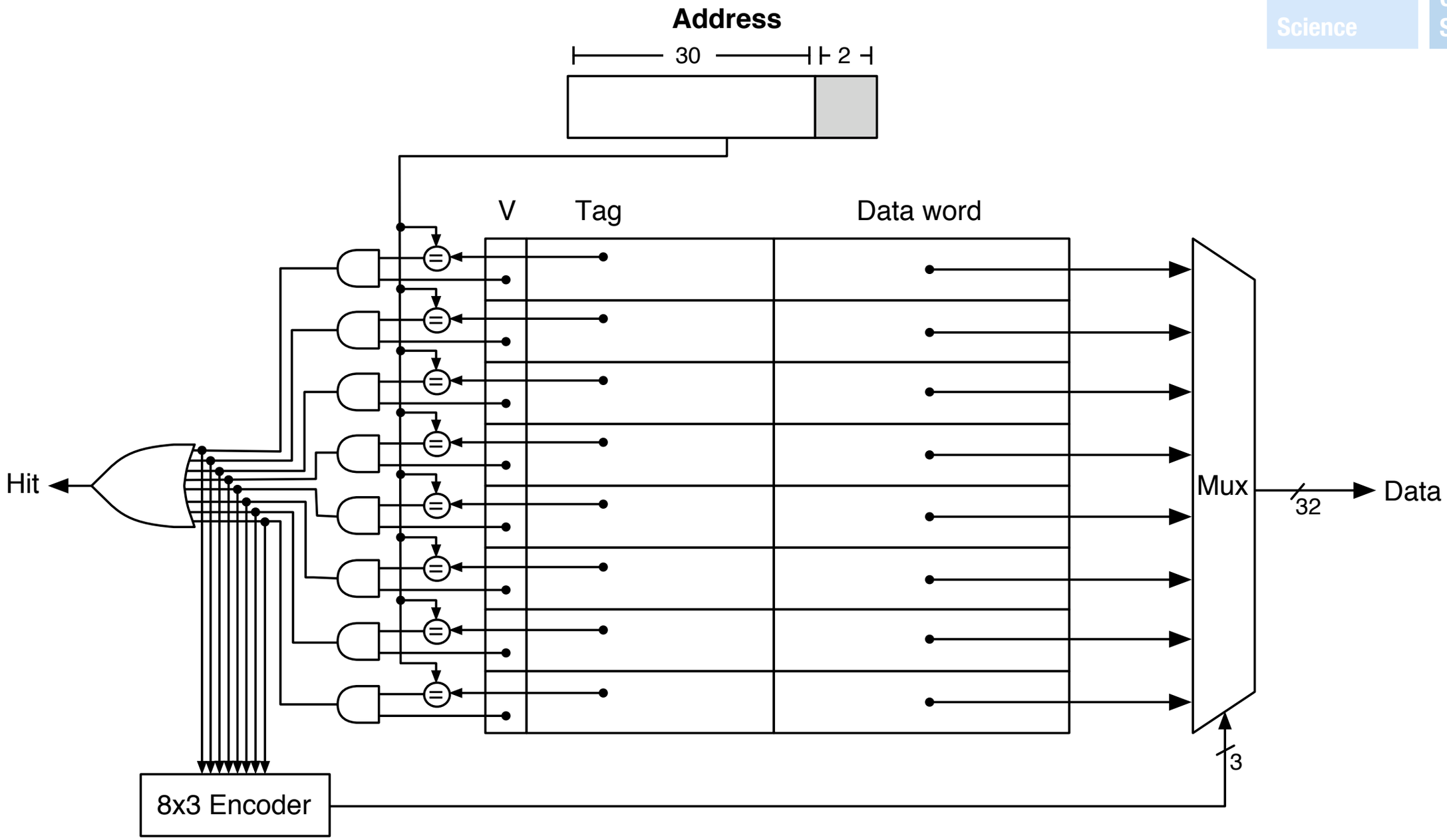
Cache

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	
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	
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	
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	
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	
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~~

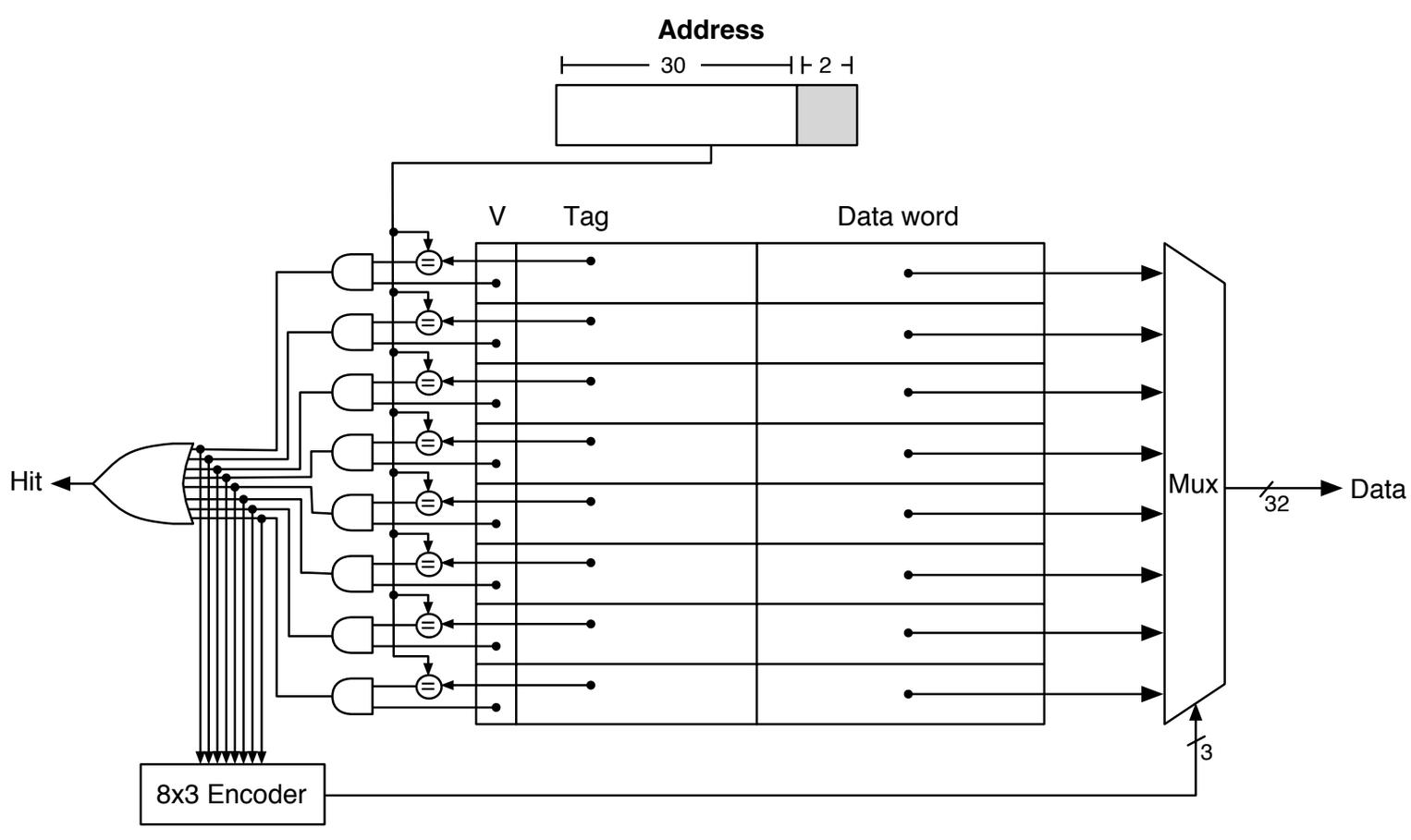
Memory

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

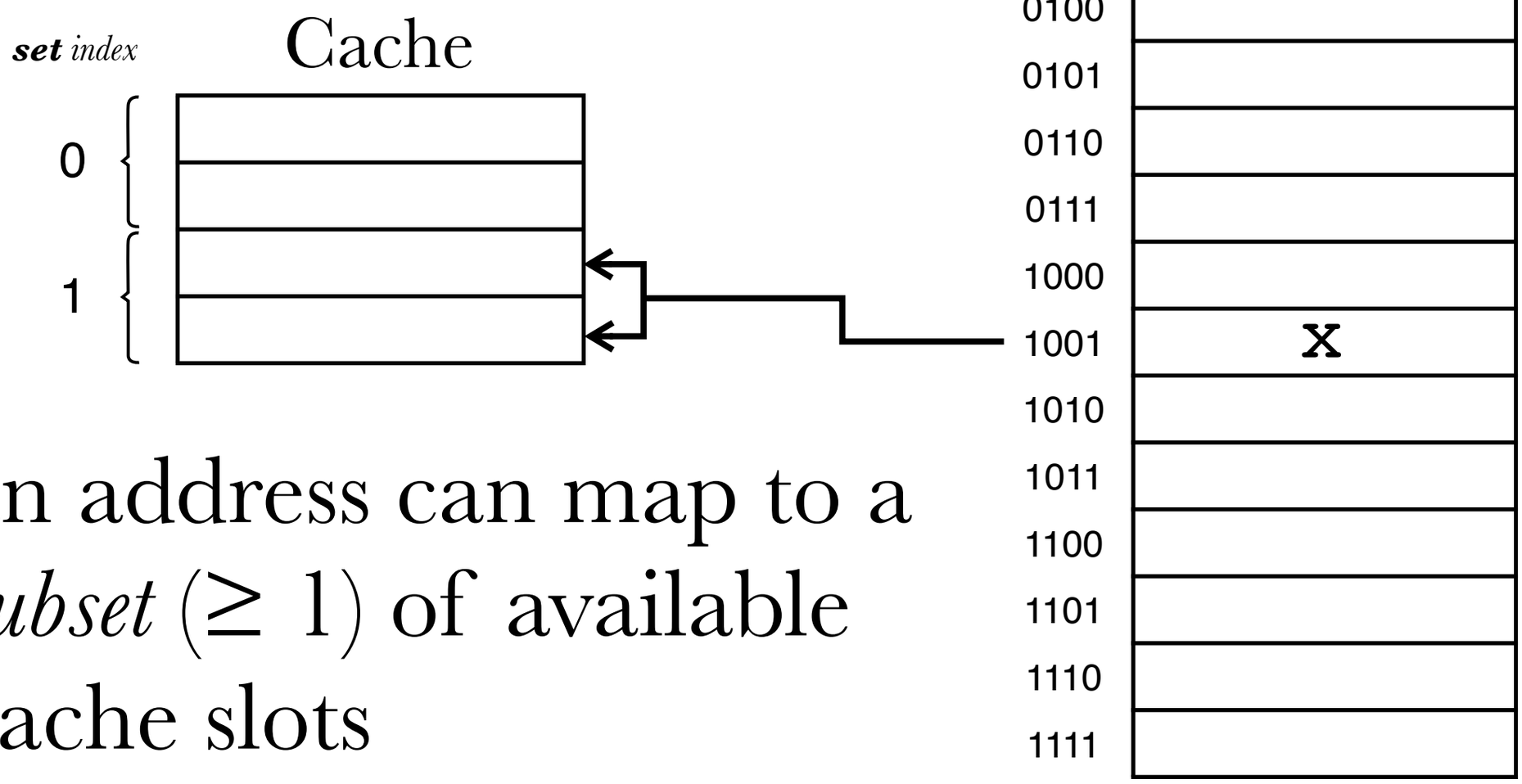
in practice, LRU is too complex (slow/ expensive) to implement in hardware

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

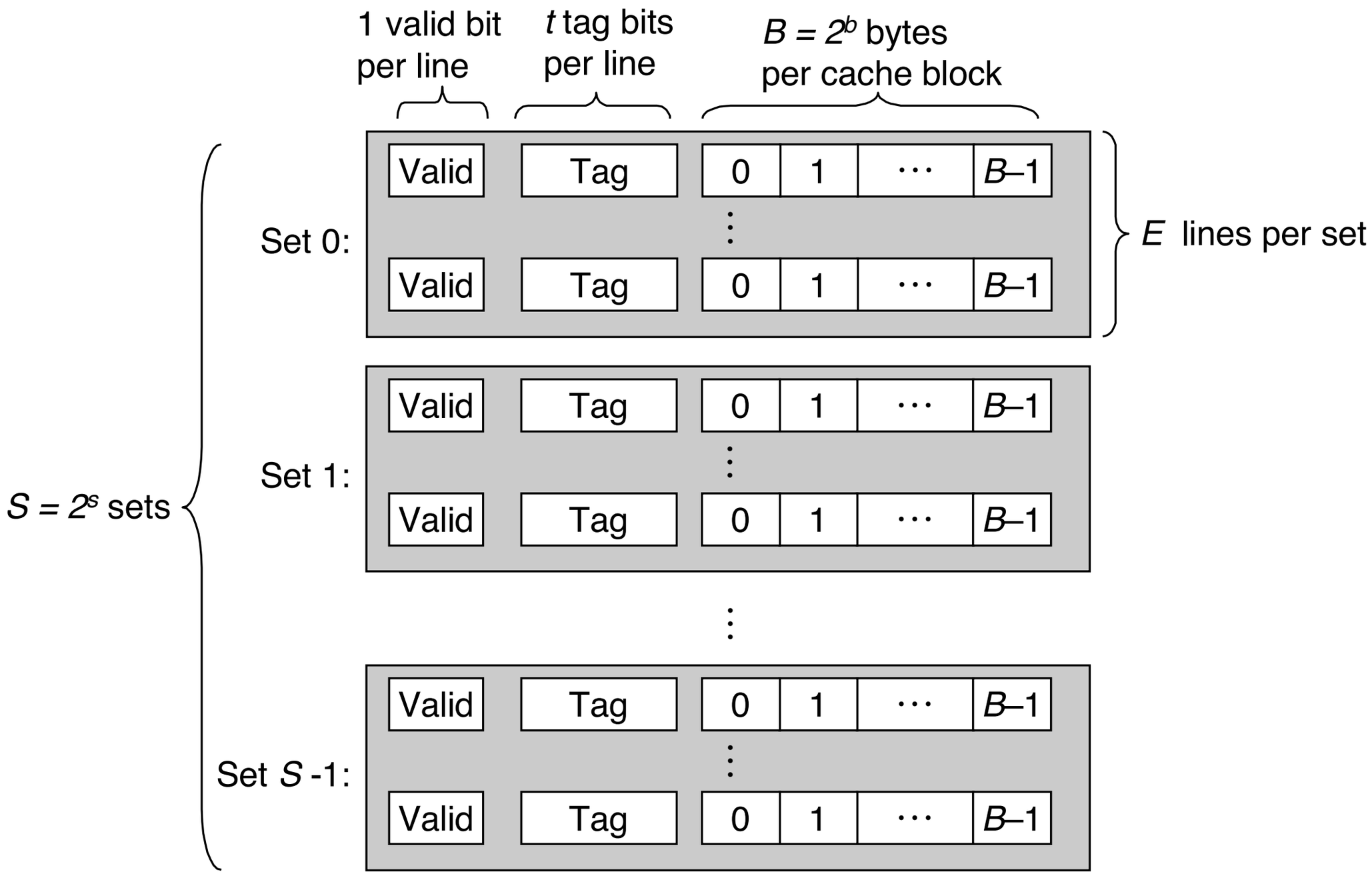
even with optimization, a *fully associative* cache with more than a few lines is prohibitively complex / expensive



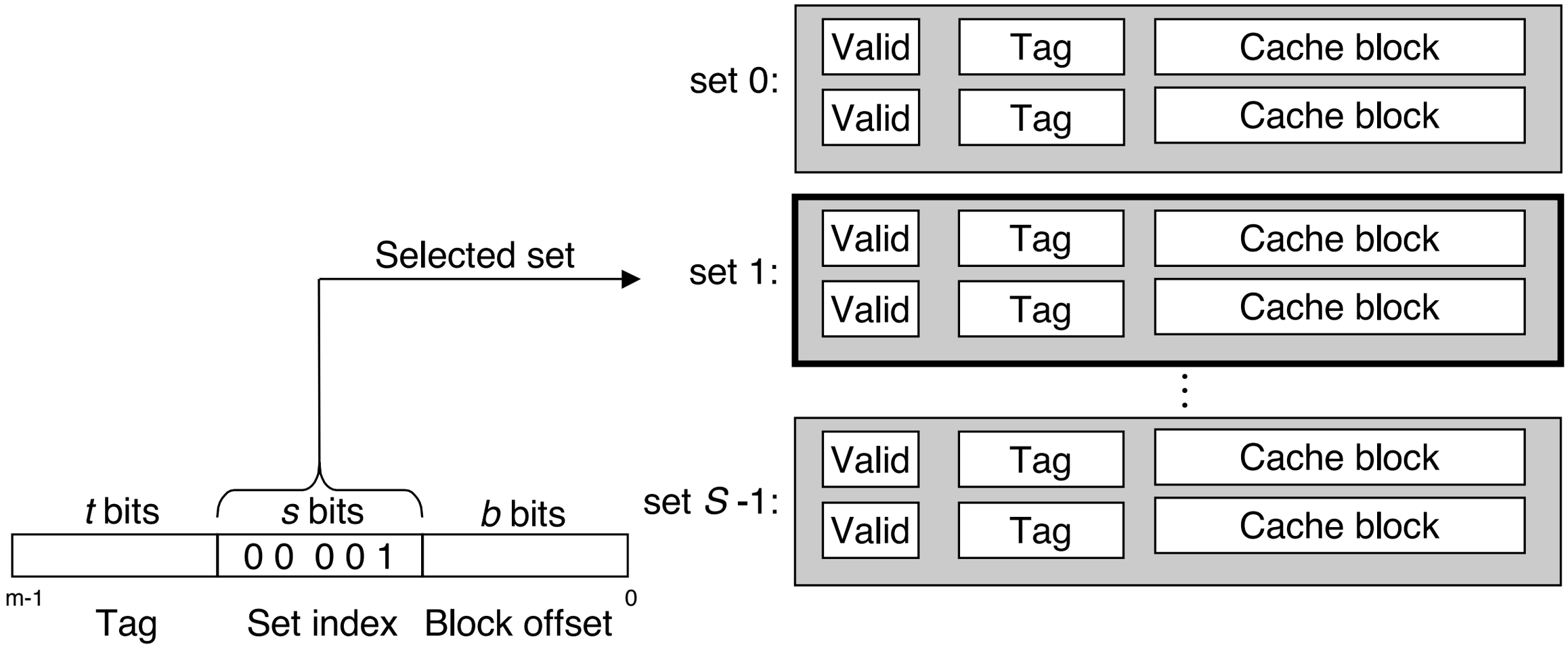
3) **set associative** mapping

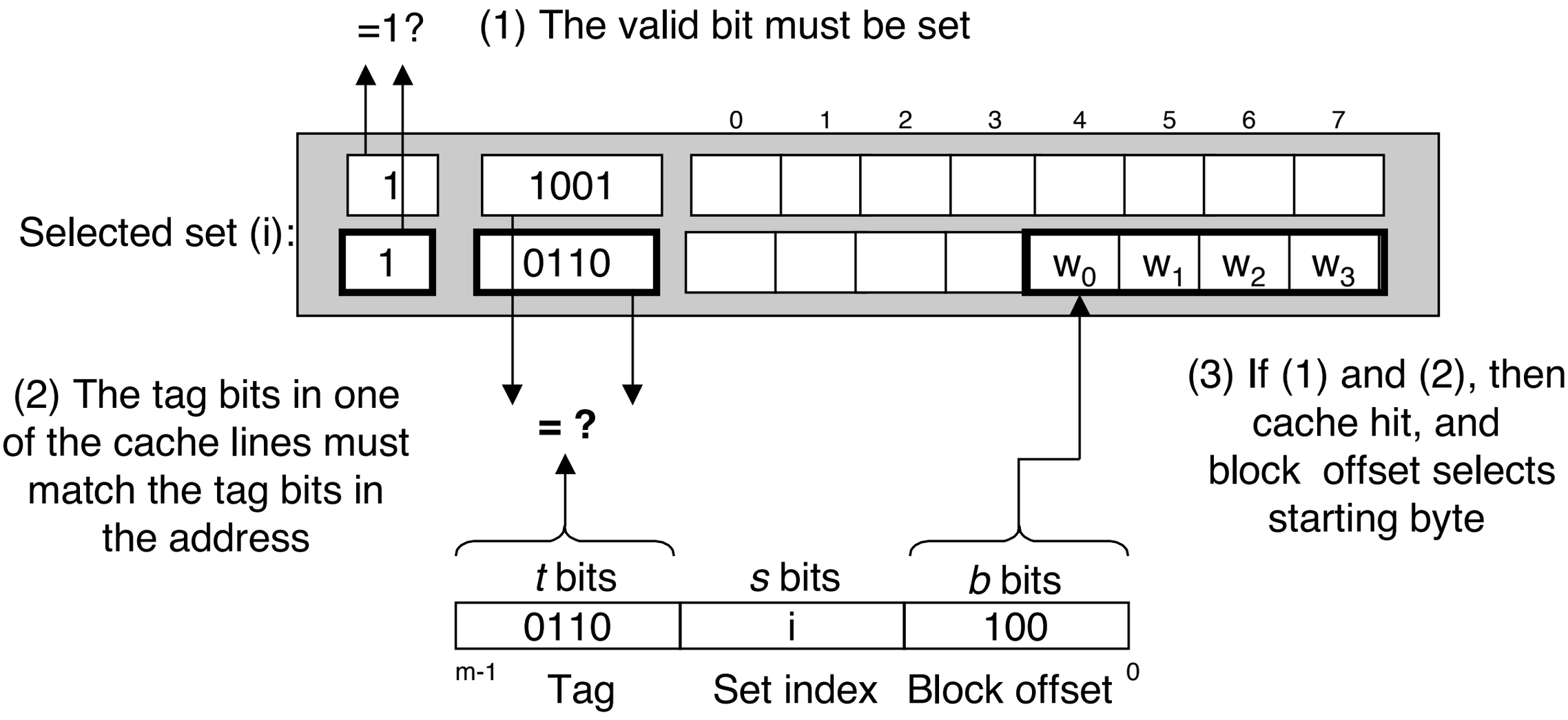


an address can map to a *subset* (≥ 1) of available cache slots



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?
(1) The valid bit must be set
0 1 2 3 4 5 6 7
Selected set (i):
1
1001
1
0110
w_0 w_1 w_2 w_3
= ?
(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
t bits
s bits
b bits
0110
i
100
m-1
Tag
Set index
Block offset
0

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

So far, only considered *read* requests;

What happens on a *write* request?

- **write-hit**: address has been cached
- **write-miss**: address has not been cached
  - in either case, no data needs to be fetched from memory — *new* data is provided by CPU

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 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
```

$$2 + 4\ [\text{first iteration}] + 2 \times \#\ \text{subsequent iterations}$$

(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

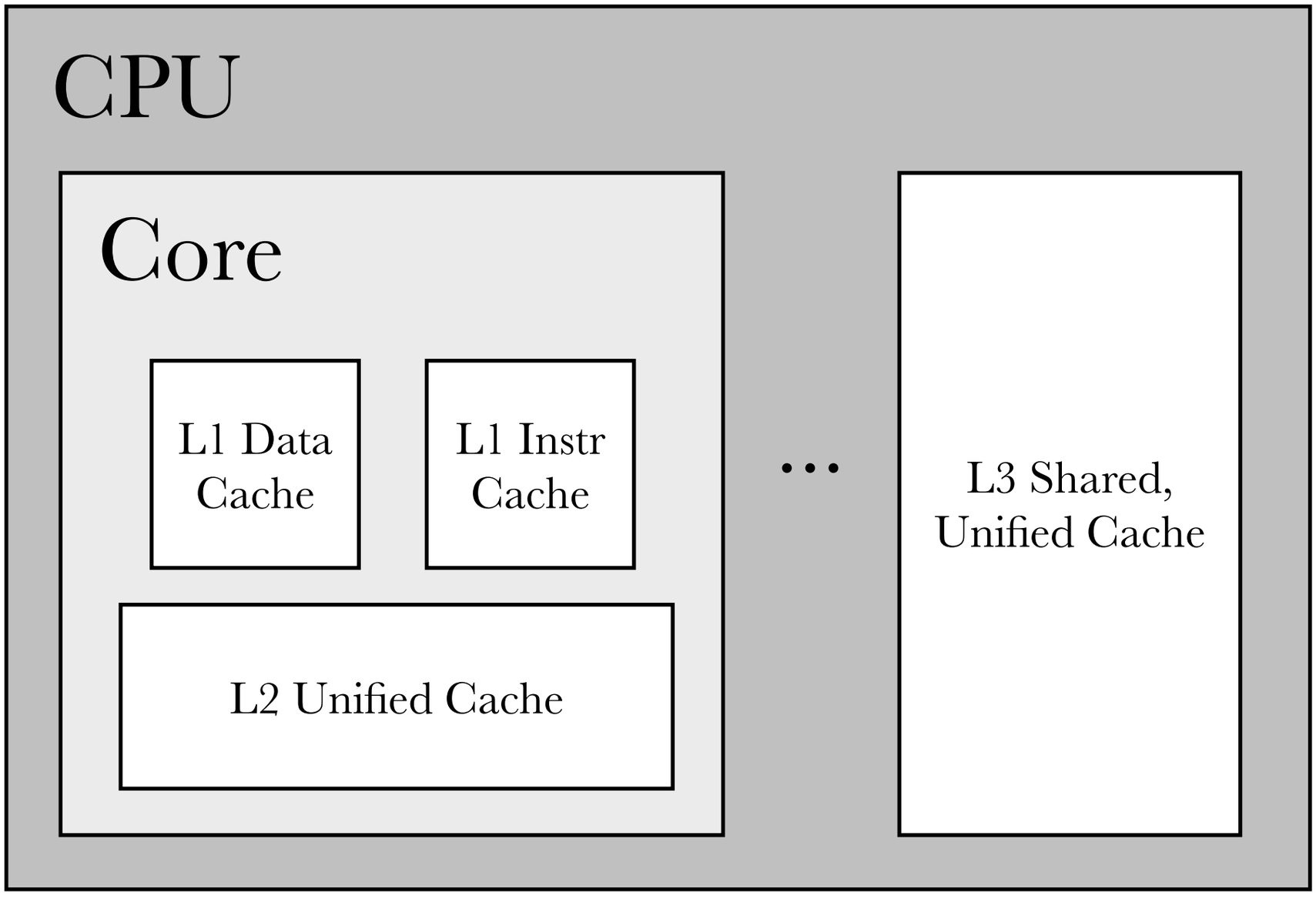
catch:

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

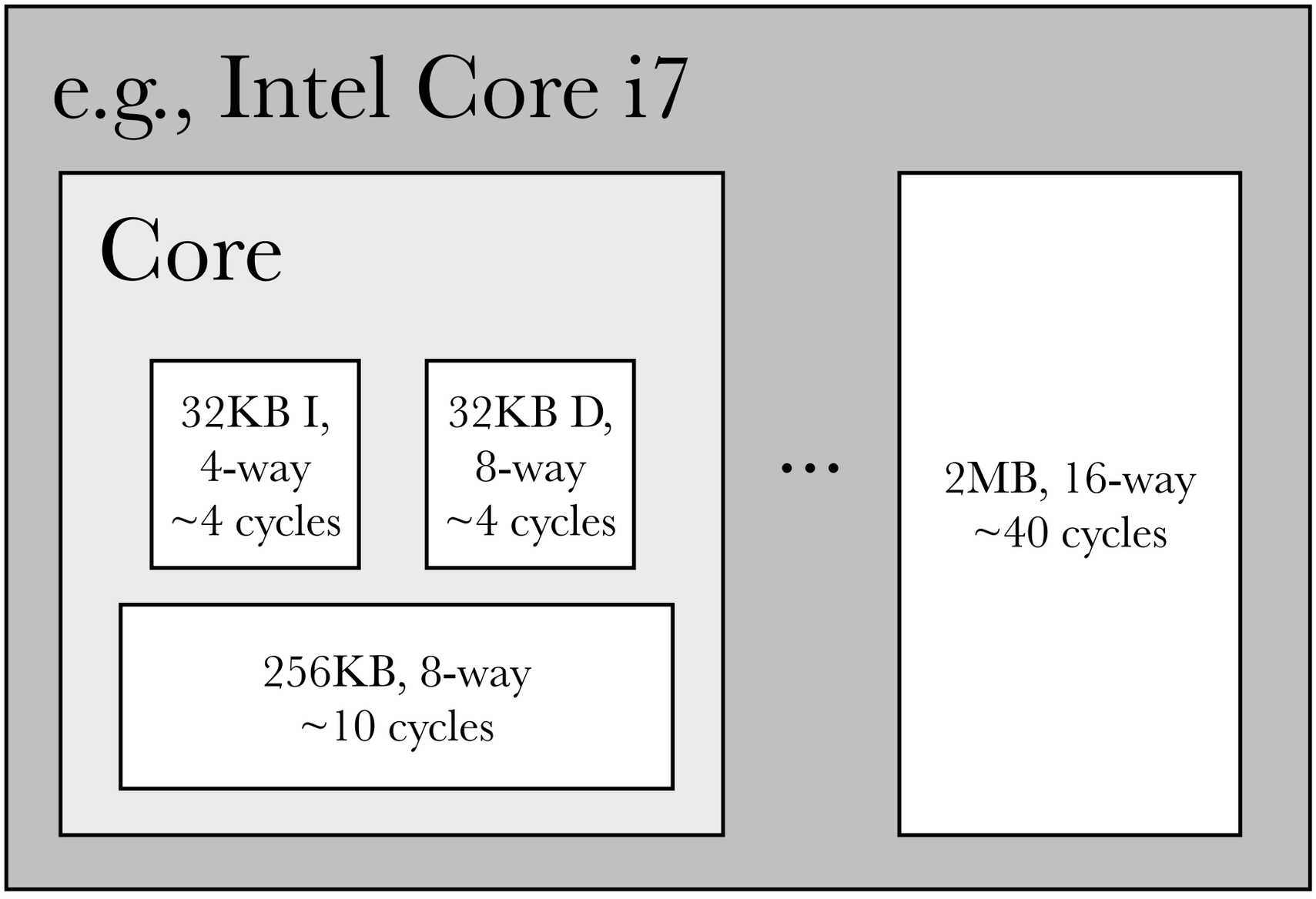
solution: use *multiple levels* of caching

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

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

In general, cache friendly code:

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

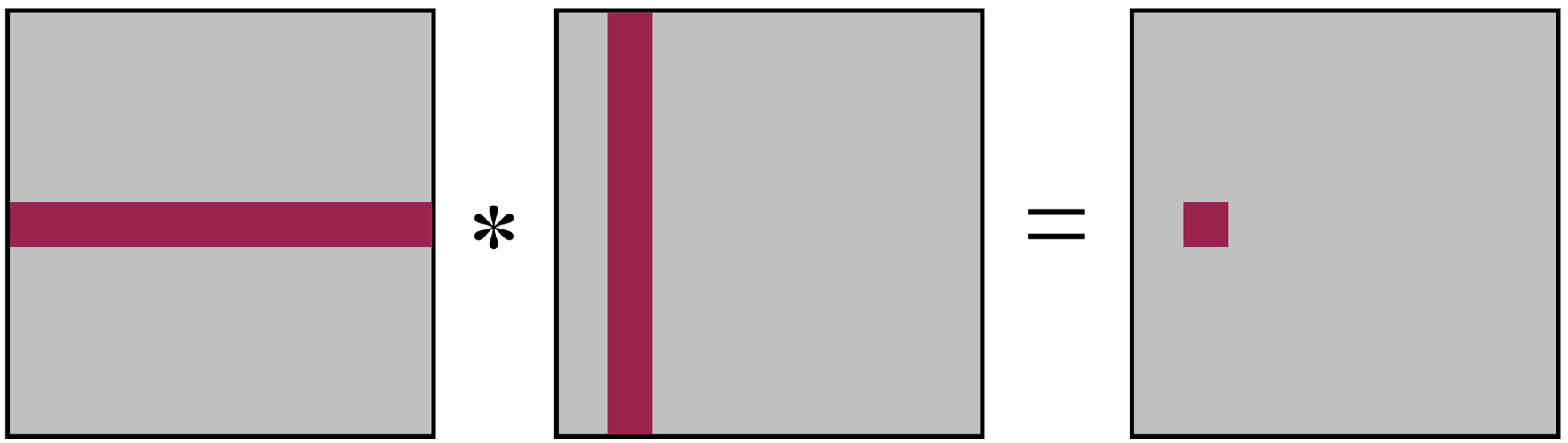
case study in software/cache interaction:

matrix multiplication

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

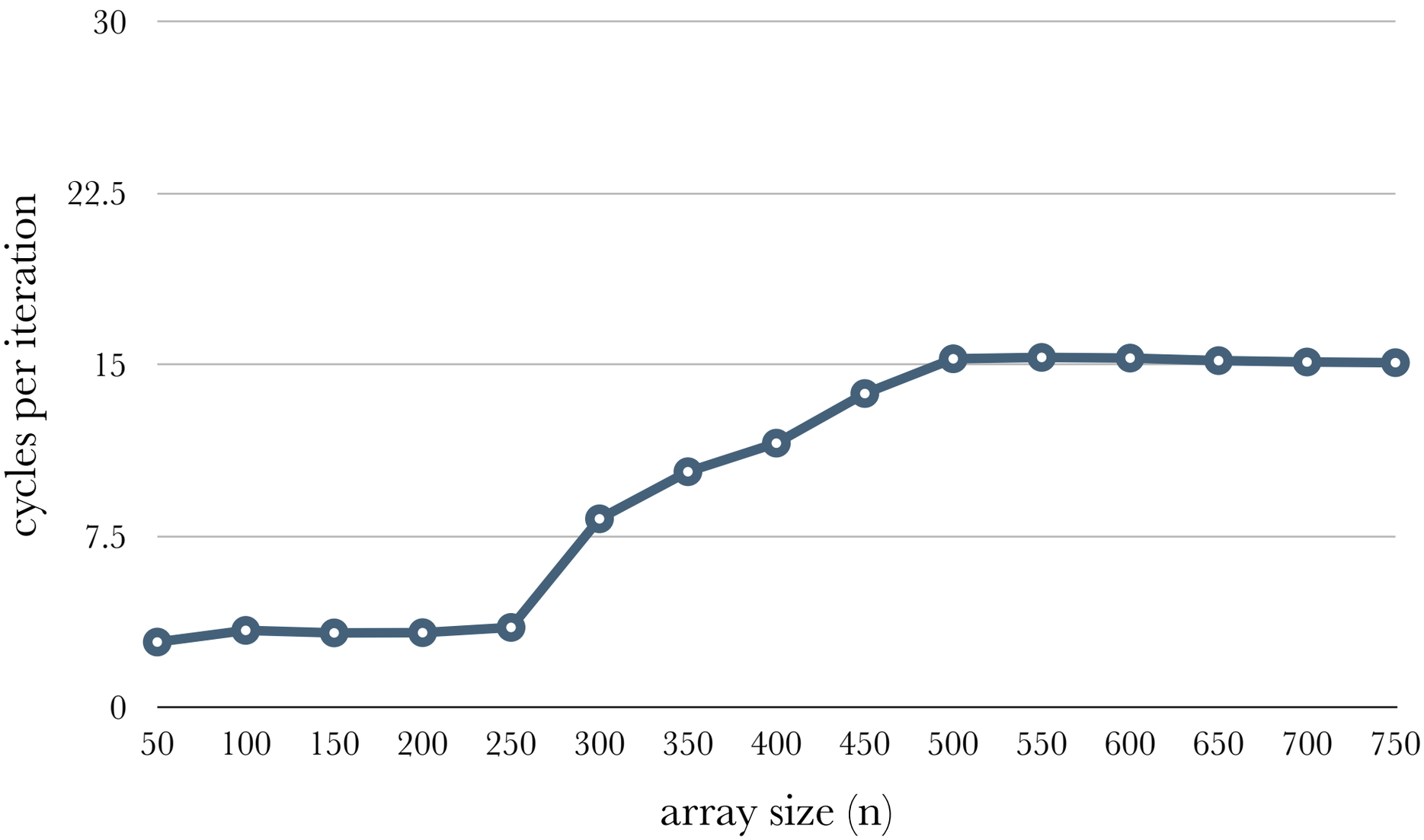
$$c_{ij} = \begin{pmatrix} a_{i1} & a_{i2} & a_{i3} \end{pmatrix} \cdot \begin{pmatrix} b_{1j} & b_{2j} & b_{3j} \end{pmatrix}$$

$$= a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j}$$

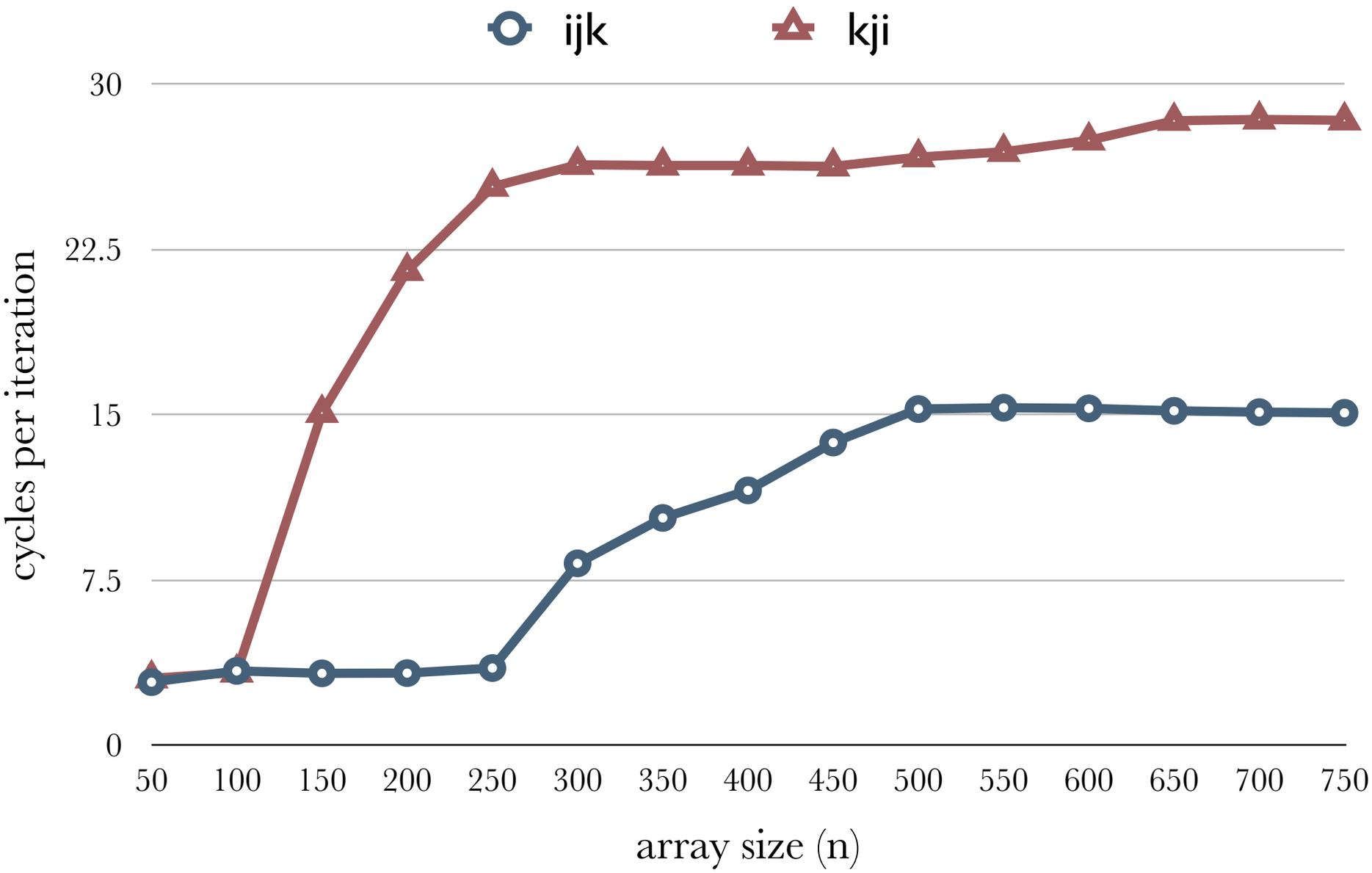


canonical implementation:

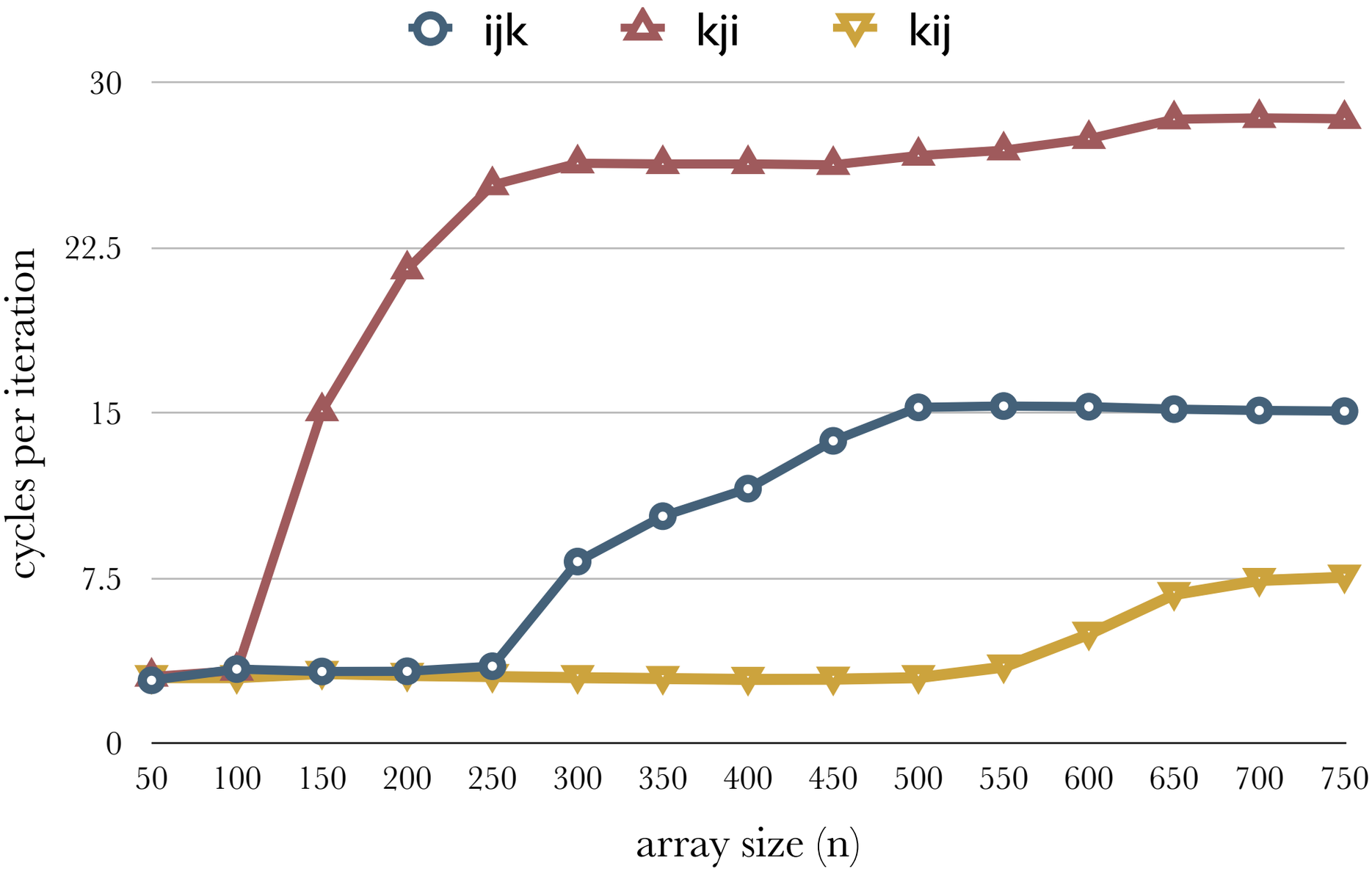
```
#define MAXN 1000
typedef double array[MAXN][MAXN];

/* 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

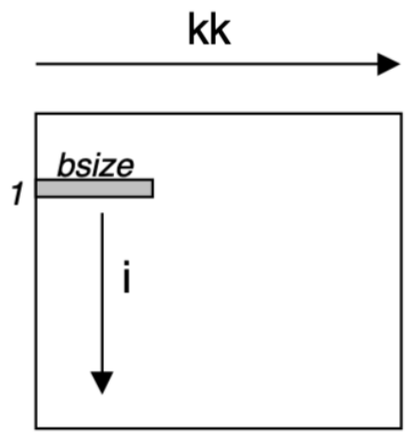
- e.g., a row of the matrix no longer fits in the cache

smaller strides can help, to an extent (by leveraging spatial locality)

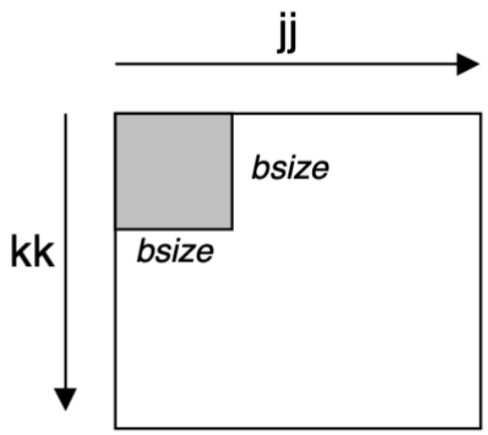
idea for optimization: deal with matrices in smaller chunks at a time that will fit in the cache — “blocking”

```
/* "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;
                }
            }
        }
    }
}
```

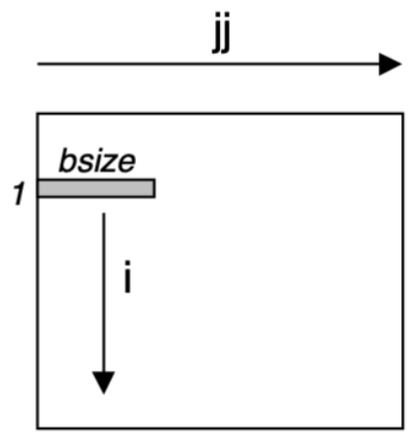
```
/* "blocked" matrix multiplication, assuming n is evenly
 * divisible by bsize */
void bijk(array A, array B, array C, int n, int bsize) {
    int i, j, k, kk, jj;
    double sum;

    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;
```



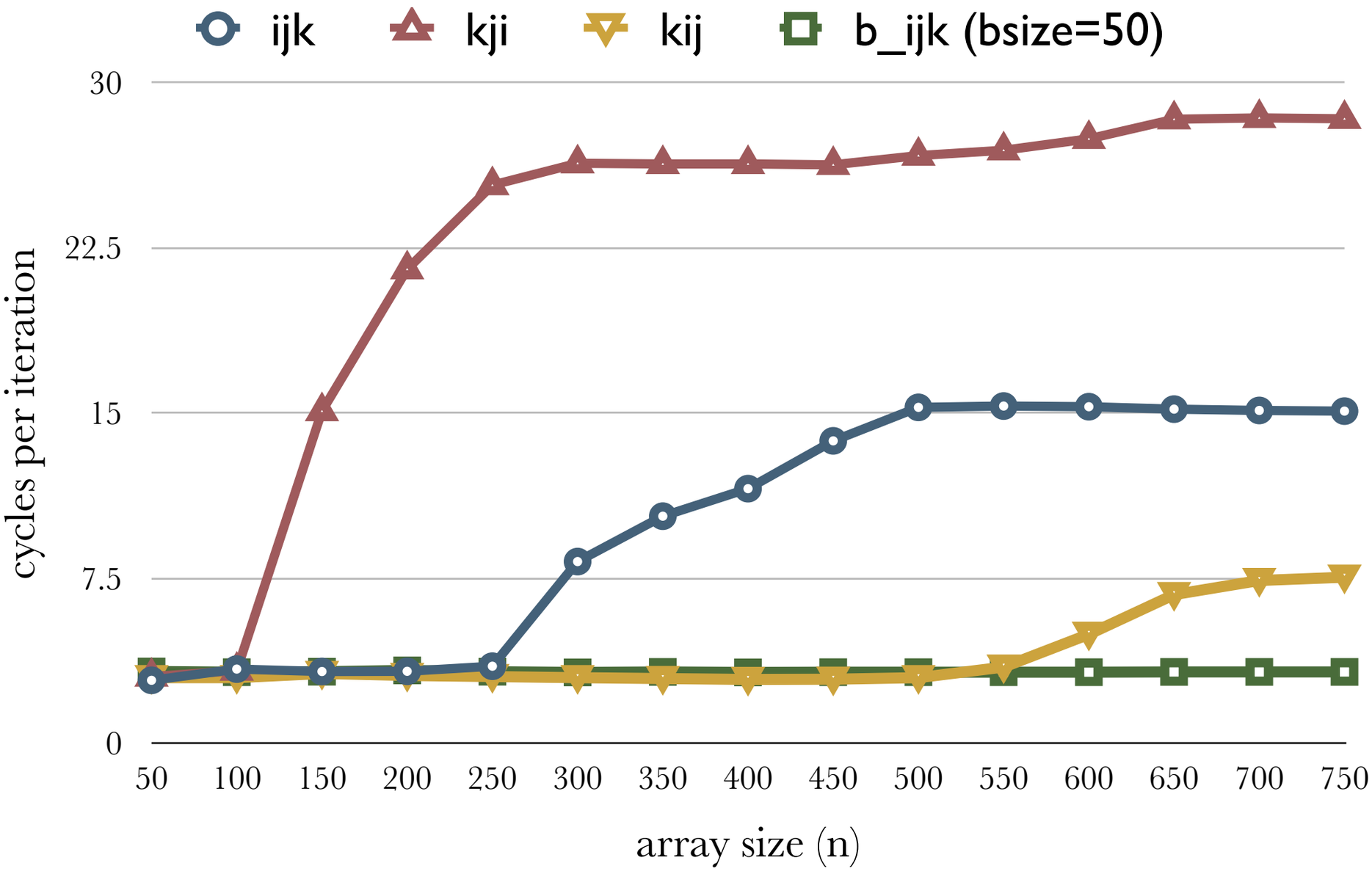
Use *1 x bsize* row sliver *bsize* times



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*