

CS 240 Stage 3

Abstractions for Practical Systems

Caching and the memory hierarchy

Operating systems and the process model

Virtual memory

Dynamic memory allocation

Victory lap

Memory Hierarchy: Cache

Memory hierarchy

Cache basics

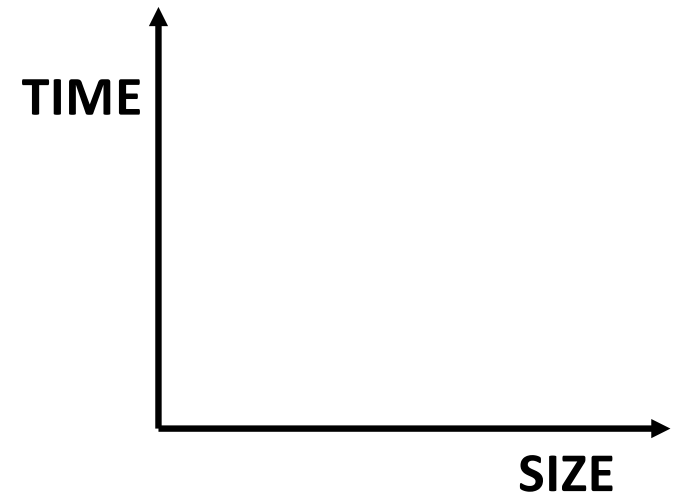
Locality

Cache organization

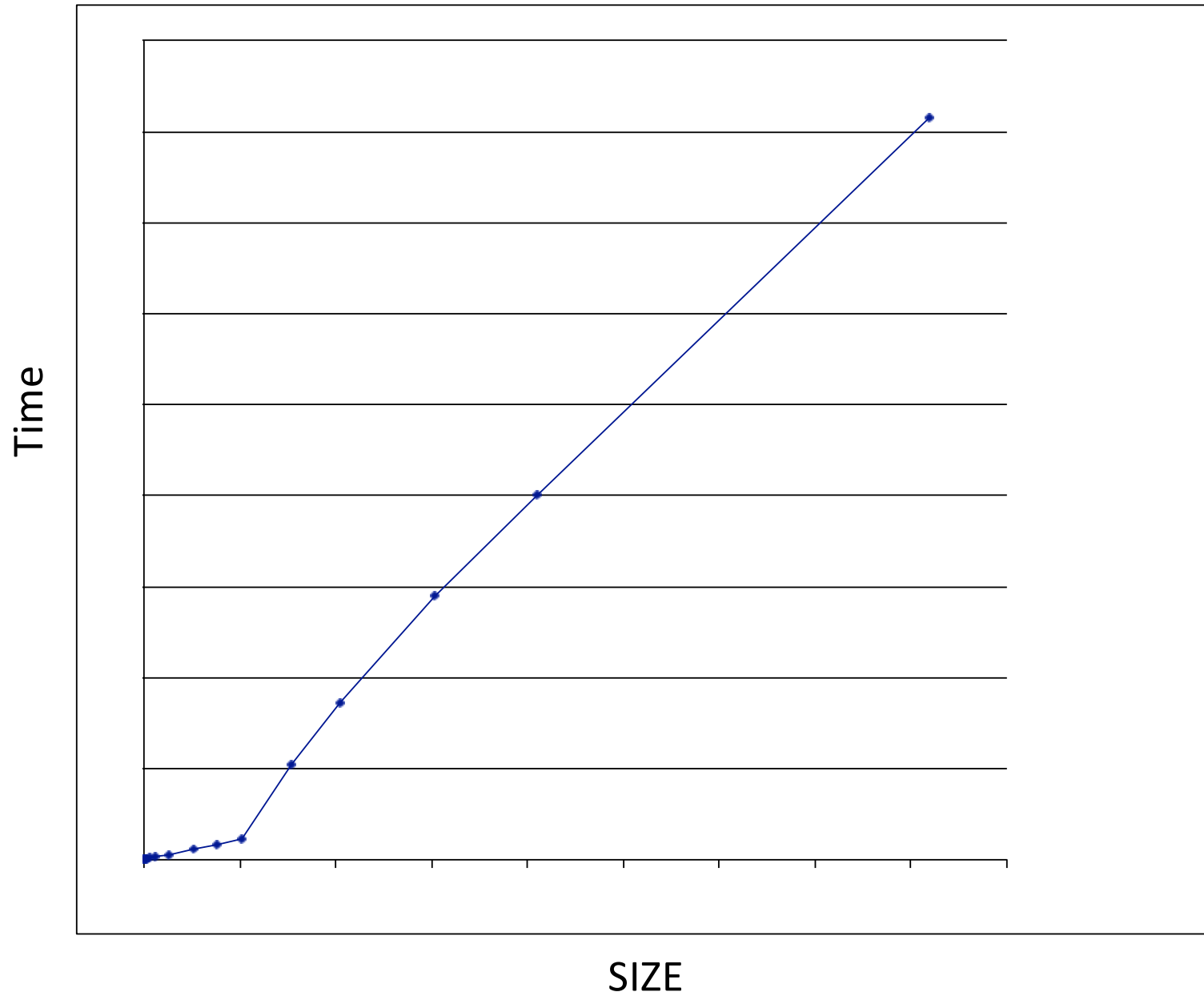
Cache-aware programming

How does execution time grow with SIZE?

```
int array[SIZE];  
fillArrayRandomly(array);  
int s = 0;  
  
for (int i = 0; i < 200000; i++) {  
    for (int j = 0; j < SIZE; j++) {  
        s += array[j];  
    }  
}
```



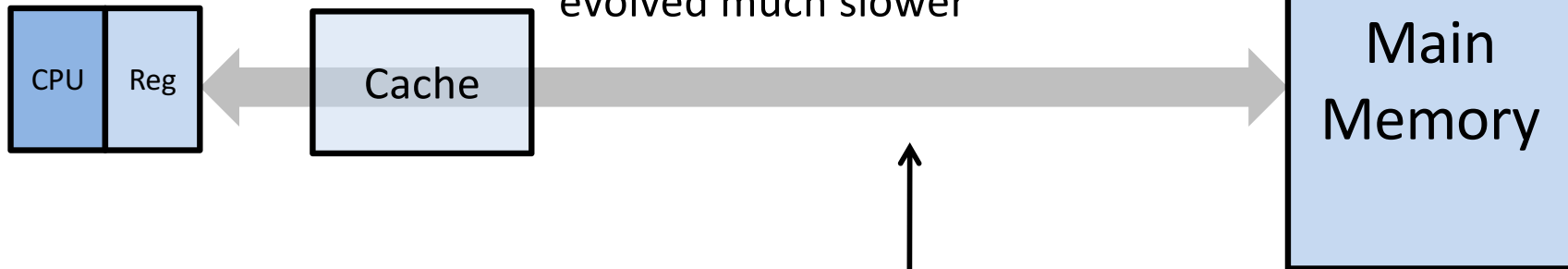
reality



Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months

Bus bandwidth
evolved much slower



Bandwidth: 256 bytes/cycle
Latency: 1-few cycles

Bandwidth: 2 Bytes/cycle
Latency: 100 cycles

Example

Solution: caches

Cache

English:

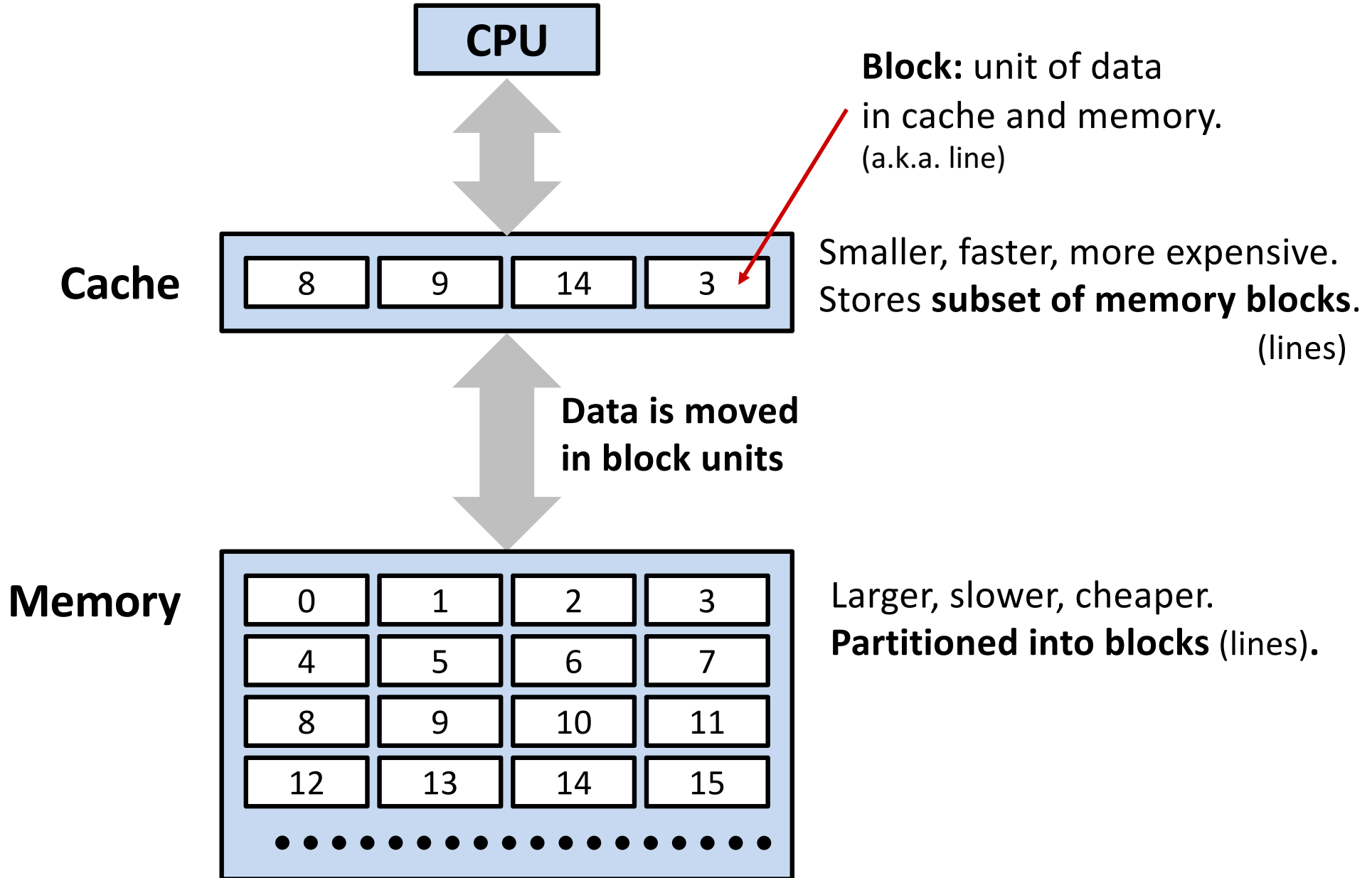
n. a hidden storage space for provisions, weapons, or treasures
v. to store away in hiding for future use

Computer Science:

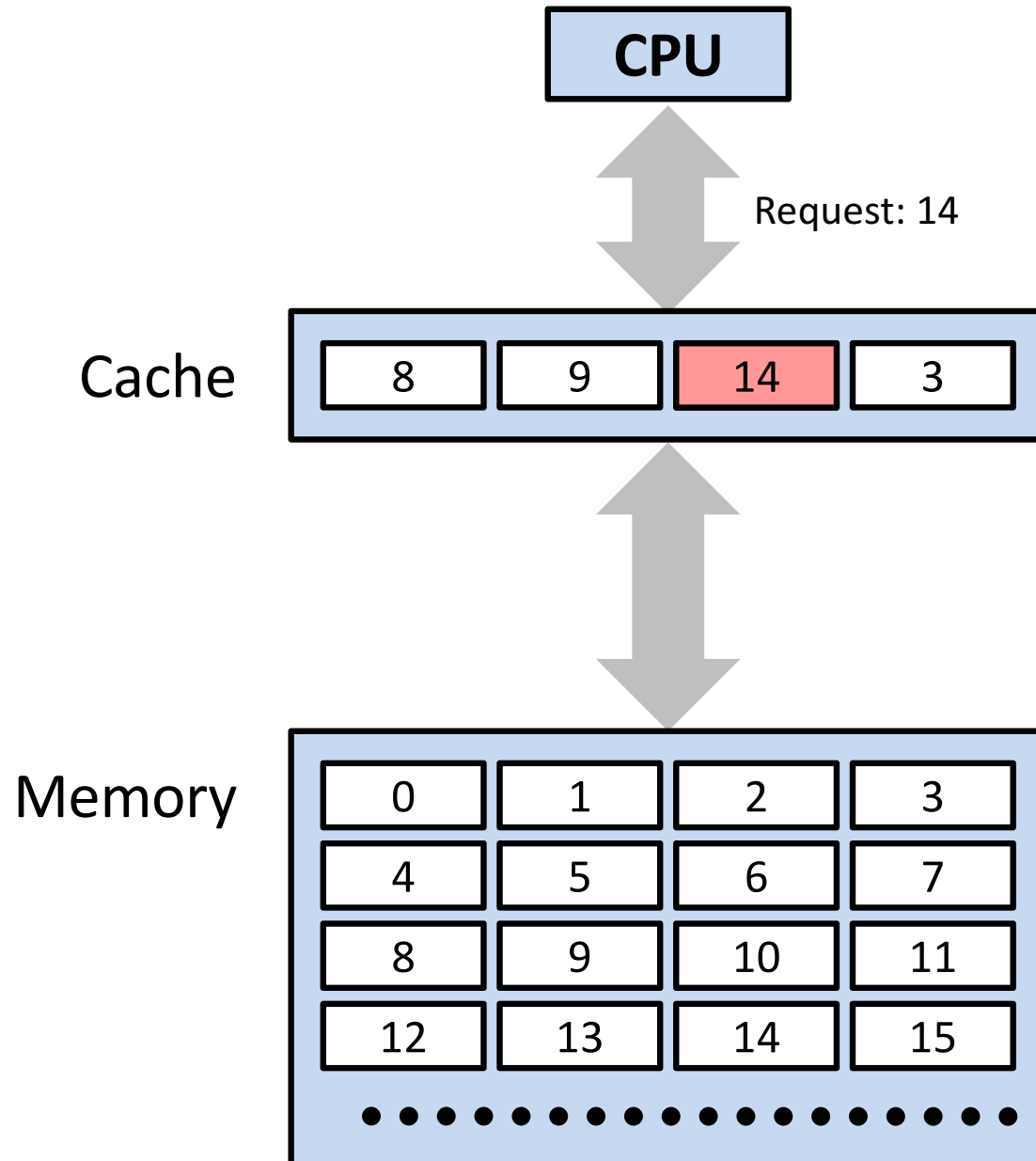
n. a computer memory with short access time used to store frequently or recently used instructions or data
v. to store [data/instructions] temporarily for later quick retrieval

Also used more broadly in CS: software caches, file caches, etc.

General Cache Mechanics



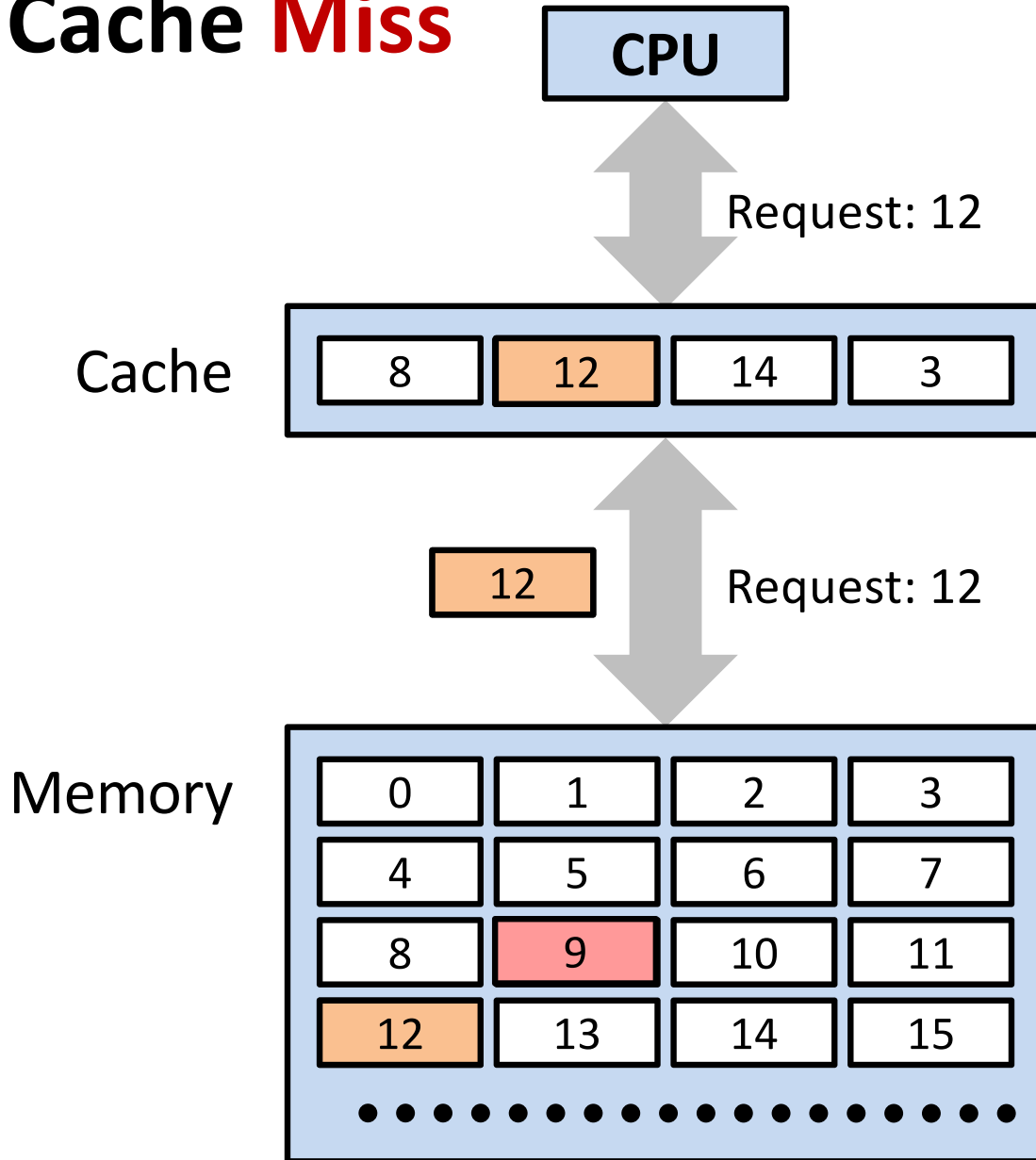
Cache Hit



1. Request data in block *b*.

2. Cache hit:
*Block *b* is in cache.*

Cache Miss



1. **Request data in block *b*.**
2. **Cache miss:**
block is not in cache
3. **Cache eviction:**
Evict a block to make room, maybe store to memory.
4. **Cache fill:**
Fetch block from memory, store in cache.

Placement Policy:
where to put block in cache

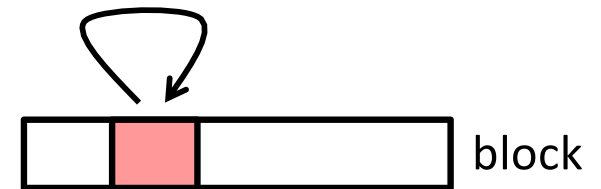
Replacement Policy:
which block to evict

Locality: why caches work

Programs tend to use data and instructions at addresses near or equal to those they have used recently.

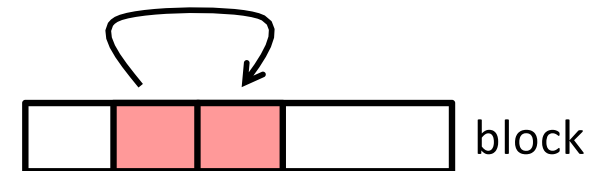
Temporal locality:

Recently referenced items are *likely* to be referenced again in the near future.



Spatial locality:

Items with nearby addresses are *likely* to be referenced close together in time.



How do caches exploit temporal and spatial locality?

Locality #1

```
sum = 0;
for (i = 0; i < n; i++) {
    sum += a[i];
}
return sum;
```

Data:

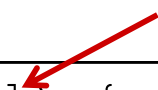
Instructions:

What is stored in memory?

Locality #2

row-major M x N 2D array in C

```
int sum_array_rows(int a[M][N]) {  
    int sum = 0;  
  
    for (int i = 0; i < M; i++) {  
        for (int j = 0; j < N; j++) {  
            sum += a[i][j];  
        }  
    }  
    return sum;  
}
```




a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Locality #3

row-major M x N 2D array in C

```
int sum_array_cols(int a[M][N]) {  
    int sum = 0;  
  
    for (int j = 0; j < N; j++) {  
        for (int i = 0; i < M; i++) {  
            sum += a[i][j];  
        }  
    }  
    return sum;  
}
```



a[0][0]	a[0][1]	a[0][2]	a[0][3]	
a[1][0]	a[1][1]	a[1][2]	a[1][3]	...
a[2][0]	a[2][1]	a[2][2]	a[2][3]	
		...		

Locality #4

```
int sum_array_3d(int a[M][N][N]) {
    int sum = 0;

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < M; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

What is "wrong" with this code?

How can it be fixed?

Cost of Cache Misses

Miss cost could be $100 \times$ hit cost.

99% hits could be twice as good as 97%. How?

Assume cache hit time of 1 cycle, miss penalty of 100 cycles

Mean access time:

$$97\% \text{ hits: } 1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$$

$$99\% \text{ hits: } 1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$$

hit/miss rates

Cache Performance Metrics

Miss Rate

Fraction of memory accesses to data not in cache (misses / accesses)
Typically: 3% - 10% for L1; maybe < 1% for L2, depending on size, etc.

Hit Time

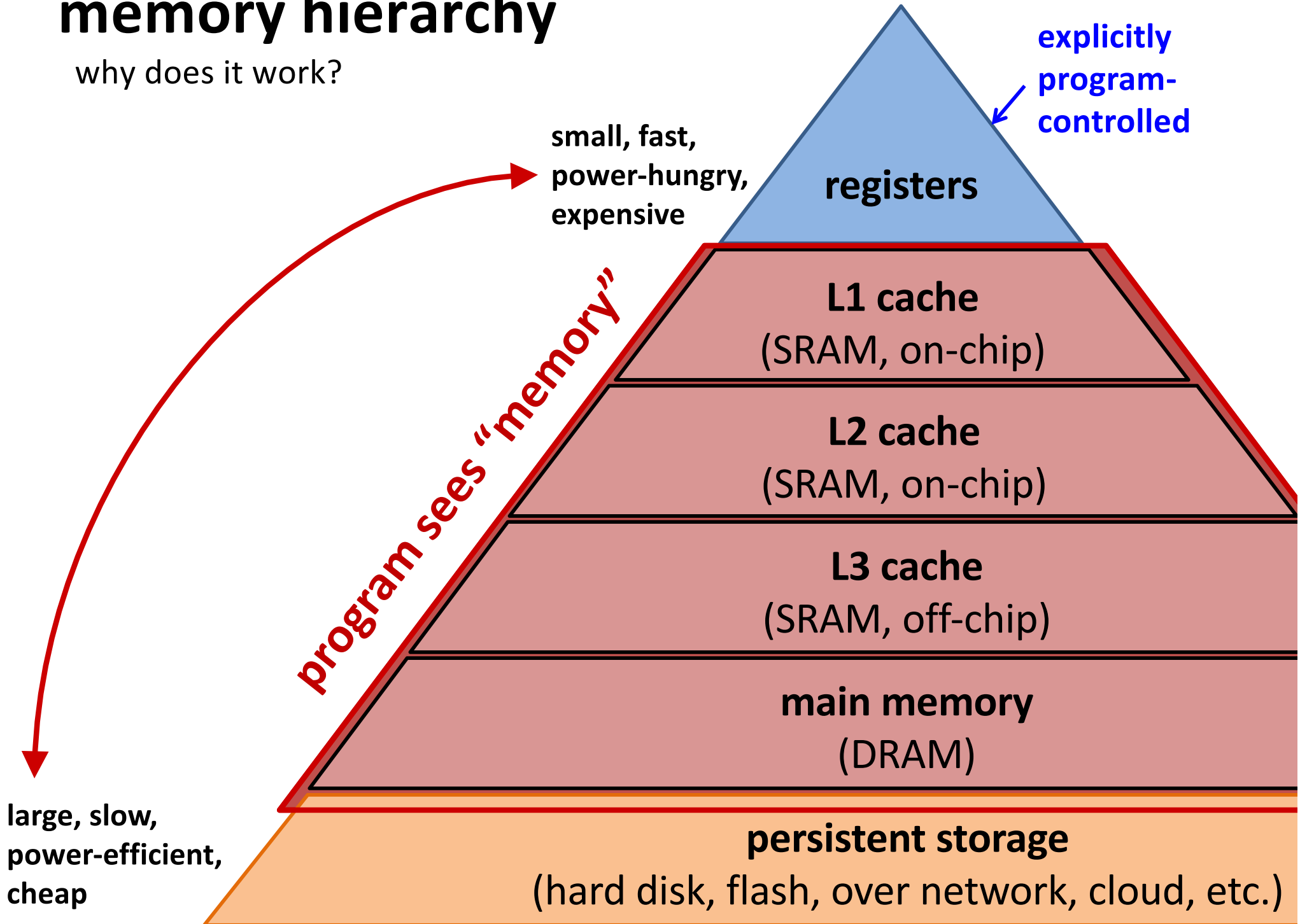
Time to find and deliver a block in the cache to the processor.
Typically: **1 - 2 clock cycles** for L1; **5 - 20 clock cycles** for L2

Miss Penalty

Additional time required on cache miss = main memory access time
Typically **50 - 200 cycles** for L2 (*trend: increasing!*)

memory hierarchy

why does it work?



Cache Organization: Key Points

Block

Fixed-size **unit of data** in memory/cache

Placement Policy

Where in the cache should a given block be stored?

- direct-mapped, set associative

Replacement Policy

What if there is no room in the cache for requested data?

- least recently used, most recently used

Write Policy

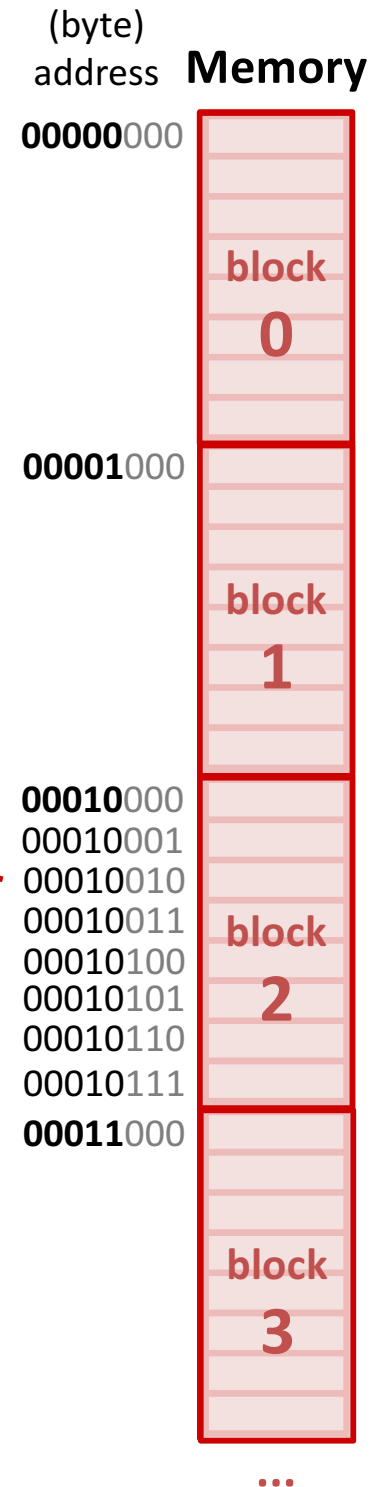
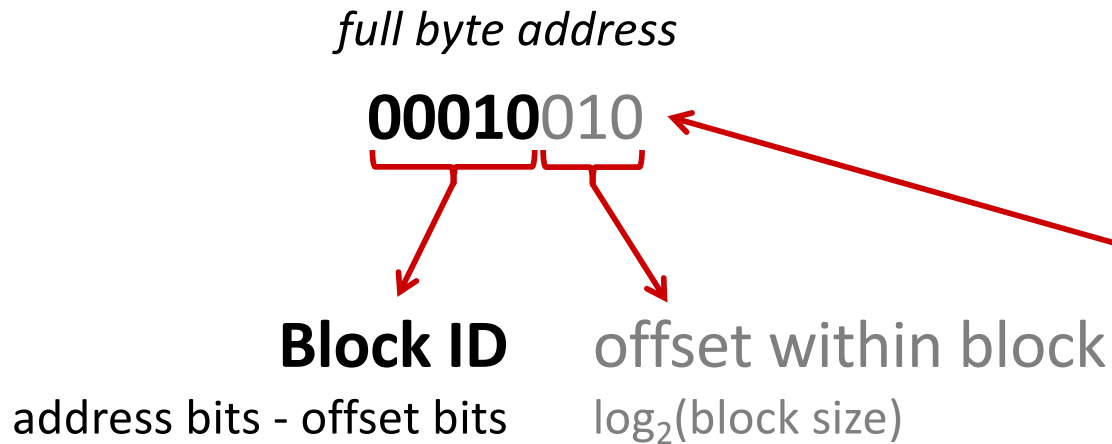
When should writes update lower levels of memory hierarchy?

- write back, write through, write allocate, no write allocate

Blocks

Divide address space into fixed-size aligned blocks.
power of 2

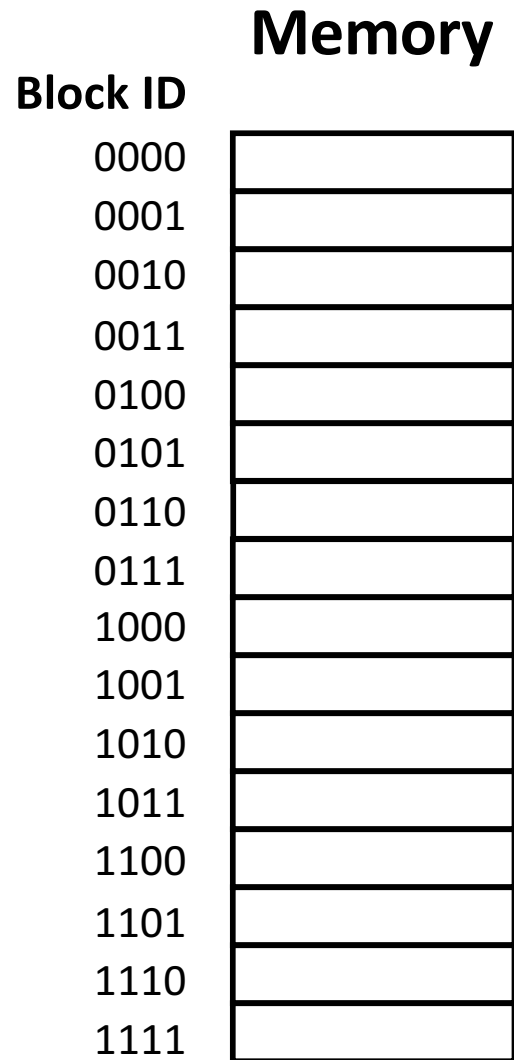
Example: block size = 8



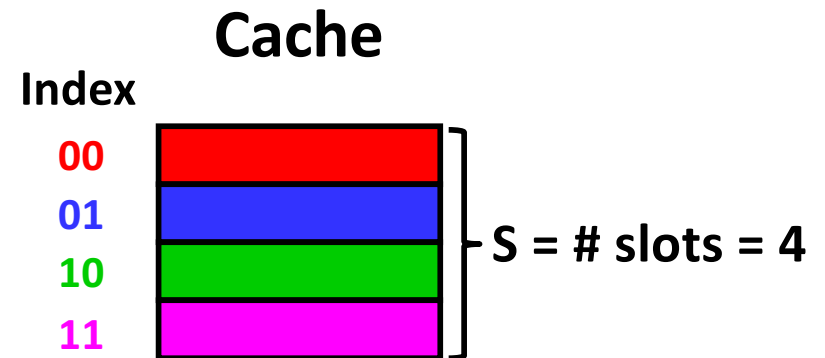
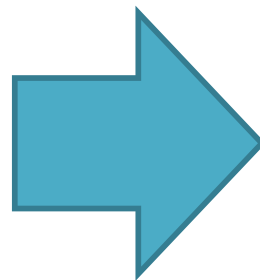
Note: drawing address order differently from here on!

remember withinSameBlock? (Pointers Lab)

Placement Policy



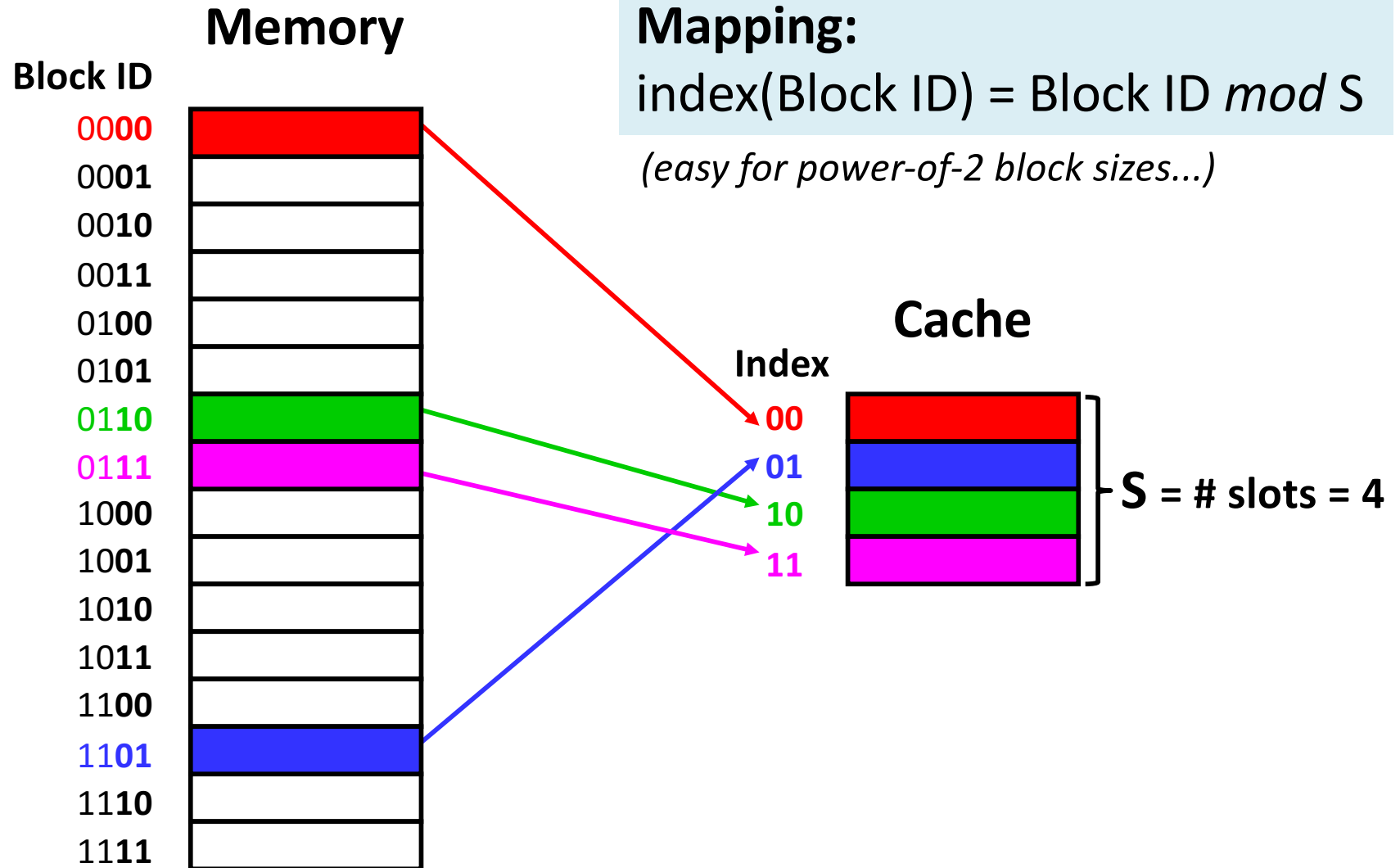
Mapping:
 $\text{index}(\text{Block ID}) = ???$



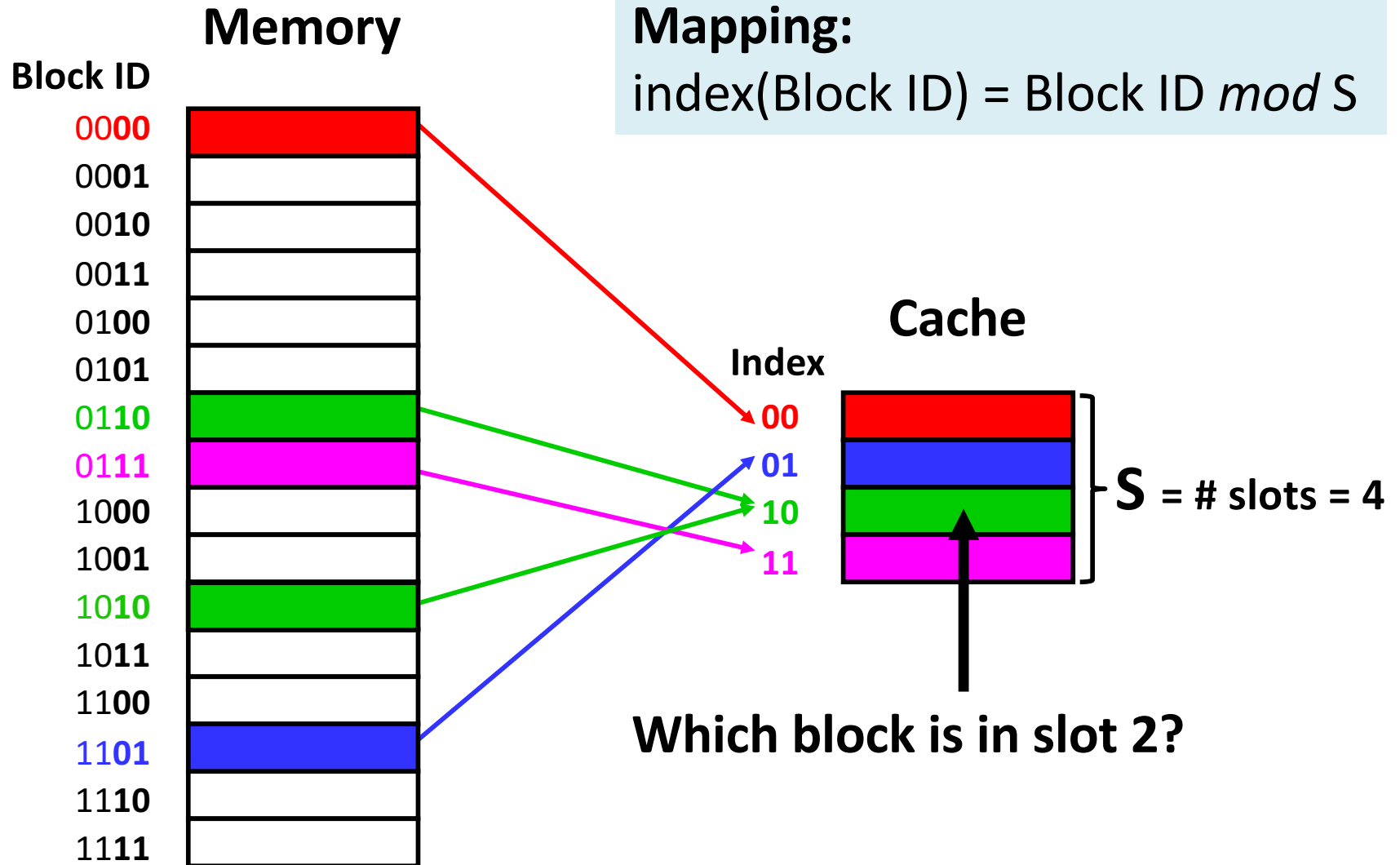
Small, fixed number of block slots.

Large, fixed number of block slots.

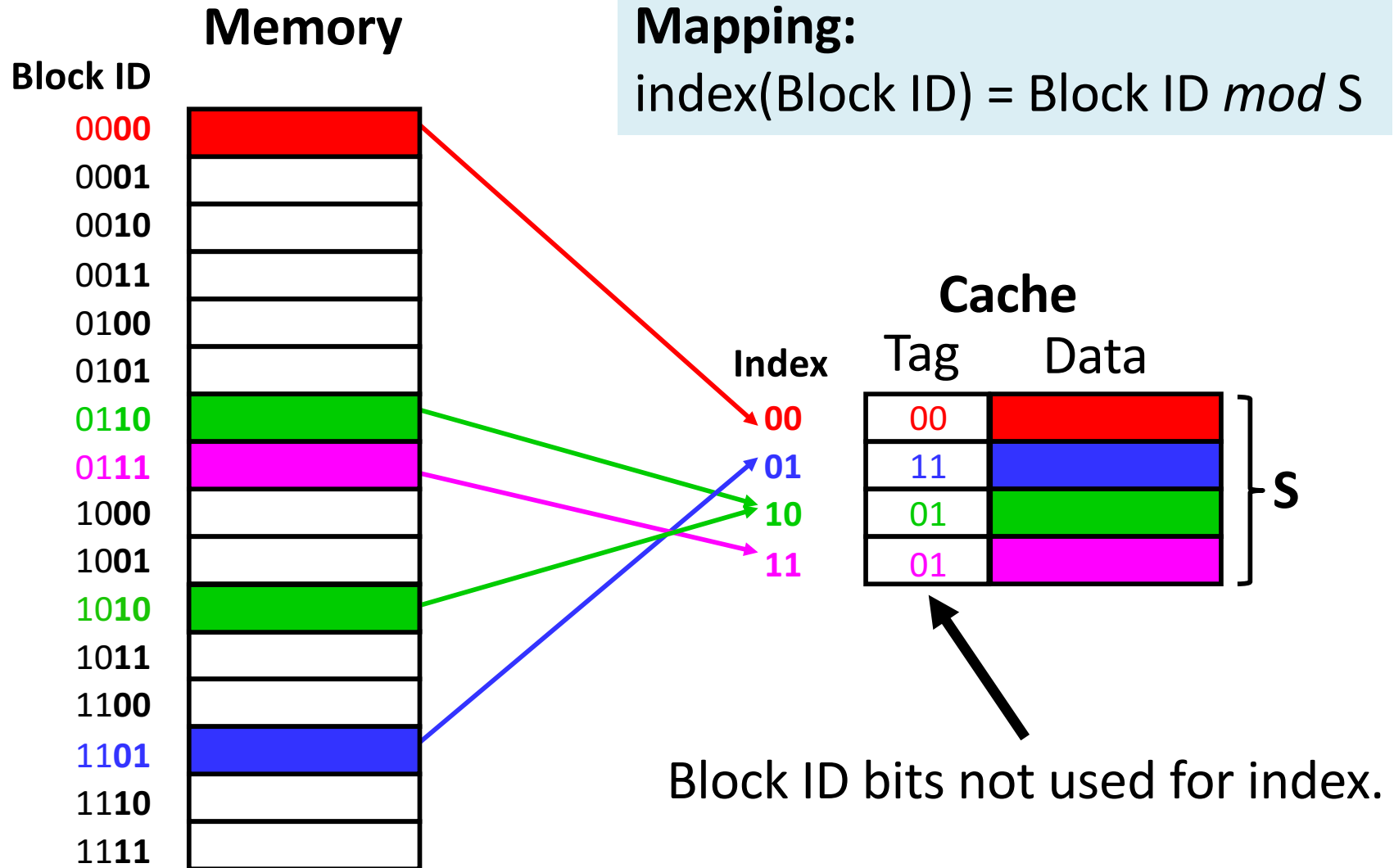
Placement: *Direct-Mapped*



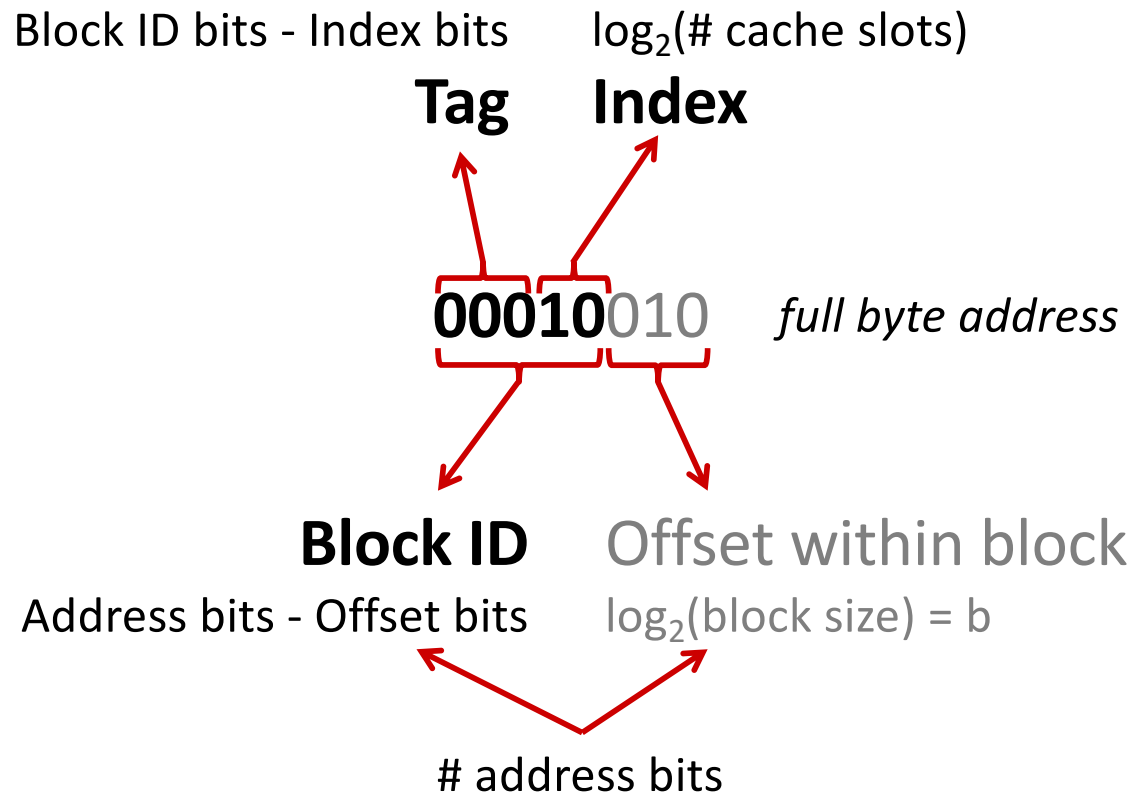
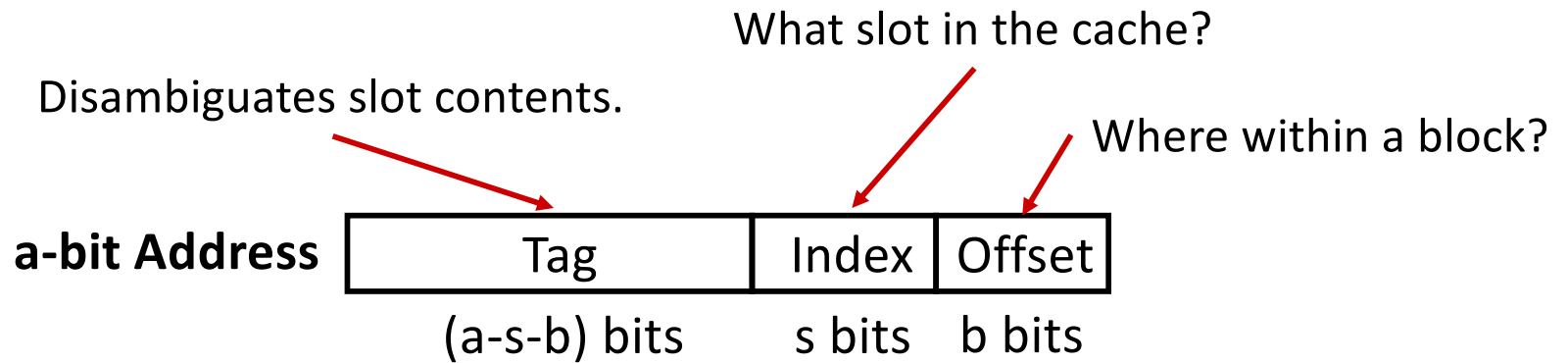
Placement: mapping ambiguity



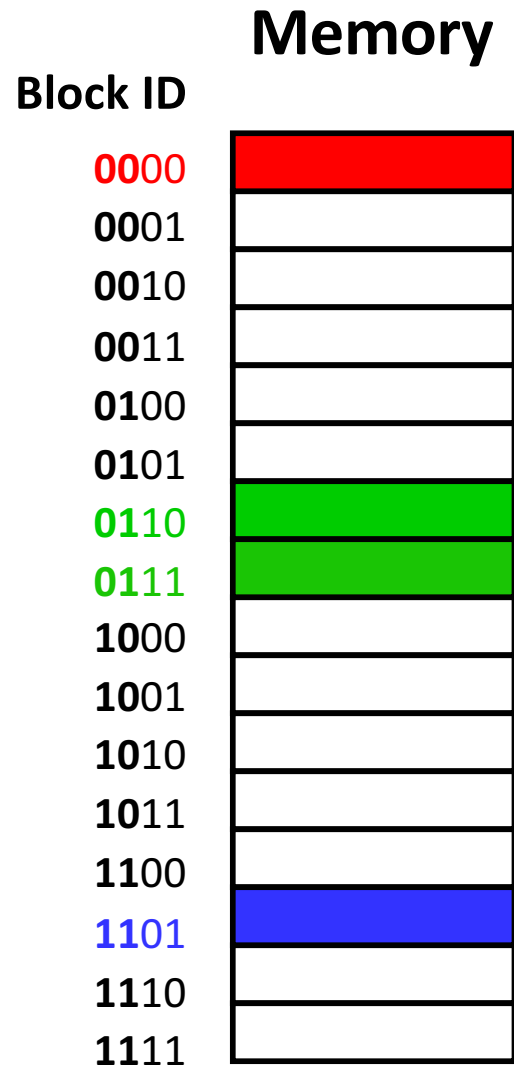
Placement: Tags resolve ambiguity



Address = Tag, Index, Offset



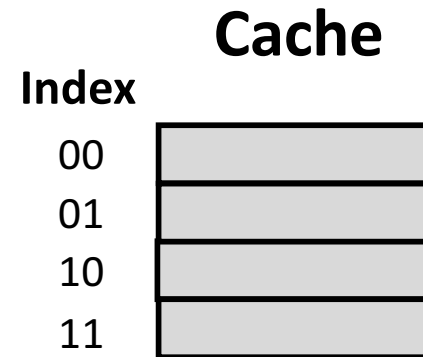
Placement: ~~Direct-Mapped~~



Why not this mapping?

$$\text{index}(\text{Block ID}) = \text{Block ID} / S$$

(still easy for power-of-2 block sizes...)



A puzzle.

Cache starts *empty*.

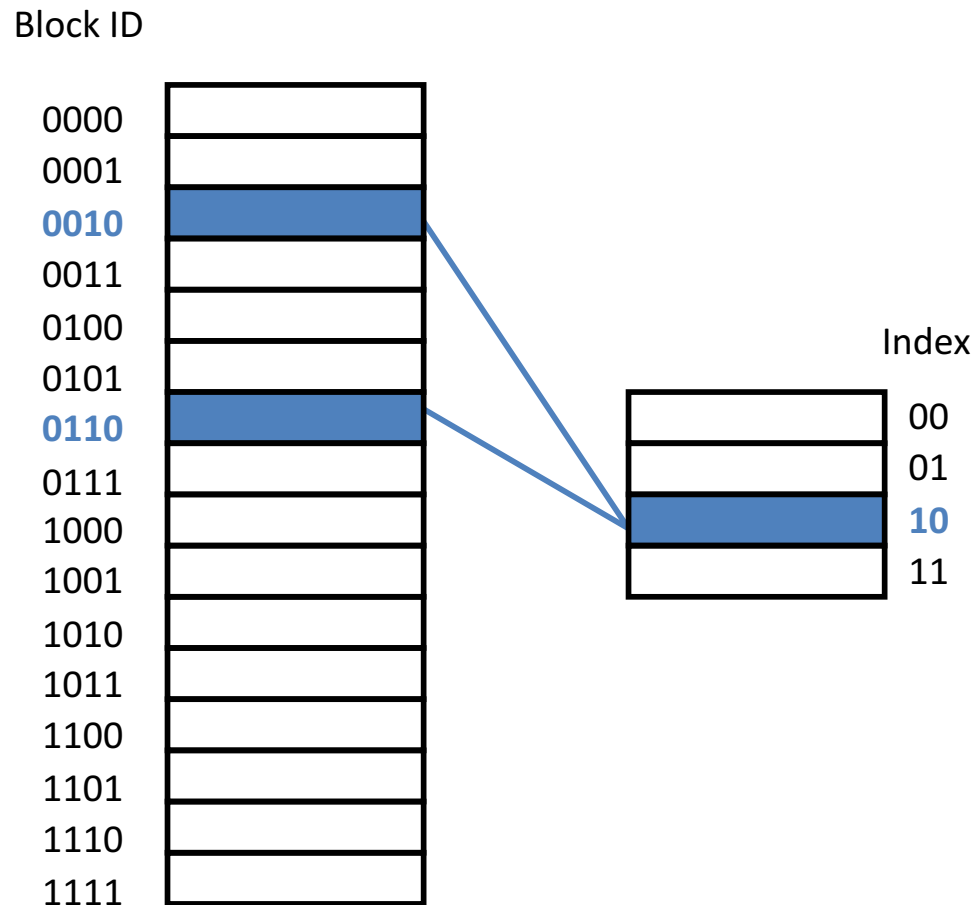
Access (address, hit/miss) stream:

(10, miss), (11, hit), (12, miss)



What could the block size be?

Placement: direct mapping conflicts



What happens when accessing
in repeated pattern:
0010, 0110, 0010, 0110, 0010...?

cache conflict

Every access suffers a miss,
evicts cache line needed
by next access.

Placement: *Set Associative*

sets
 $S = \#$ ~~slots~~ in cache

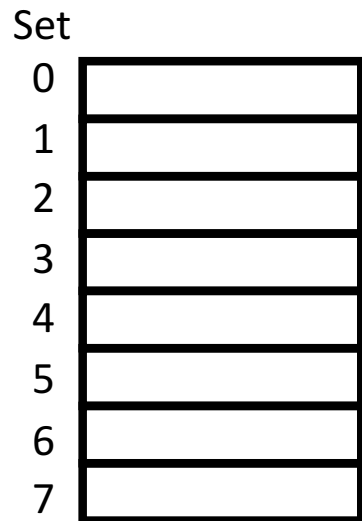
One index per *set* of block slots.
Store block in *any* slot within set.

Mapping:

$$\text{index}(\text{Block ID}) = \text{Block ID} \bmod S$$

1-way

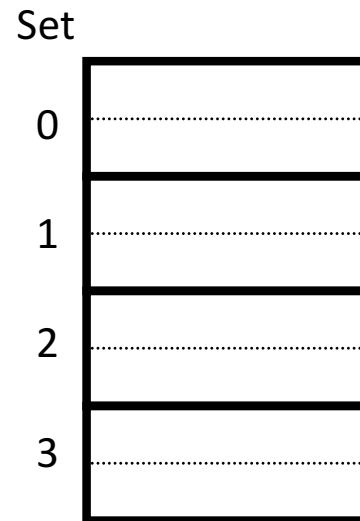
8 sets,
1 block each



direct mapped

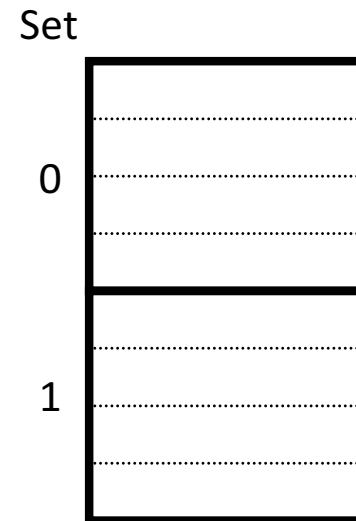
2-way

4 sets,
2 blocks each



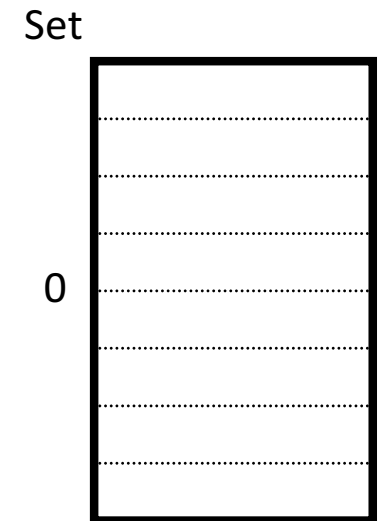
4-way

2 sets,
4 blocks each



8-way

1 set,
8 blocks



fully associative

Replacement policy: if set is full, what block should be replaced?
Common: **least recently used (LRU)**
but hardware usually implements “not most recently used”

Example: Tag, Index, Offset?



Direct-mapped

4 slots

2-byte blocks

tag bits _____

set index bits _____

block offset bits _____

index(1101) = _____

Example: Tag, Index, Offset?

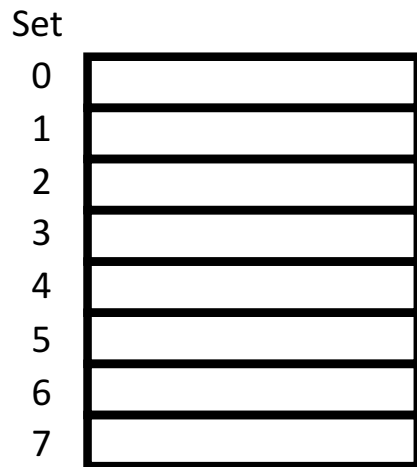
E-way set-associative

S slots

16-byte blocks



E = 1-way
S = 8 sets



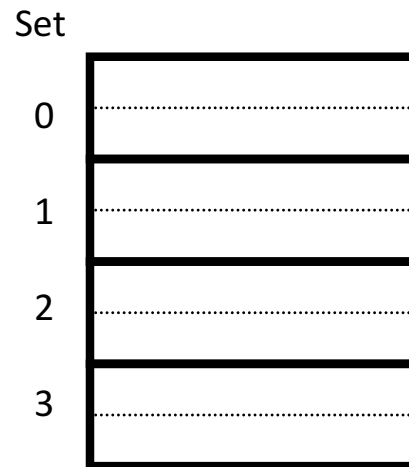
tag bits _____

set index bits _____

block offset bits _____

index(0x1833) _____

E = 2-way
S = 4 sets



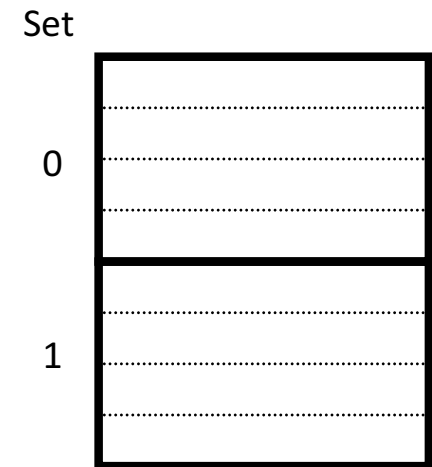
tag bits _____

set index bits _____

block offset bits _____

index(0x1833) _____

E = 4-way
S = 2 sets



tag bits _____

set index bits _____

block offset bits _____

index(0x1833) _____

Replacement Policy

If set is full, what block should be replaced?

Common: **least recently used (LRU)**

(but hardware usually implements “not most recently used”)

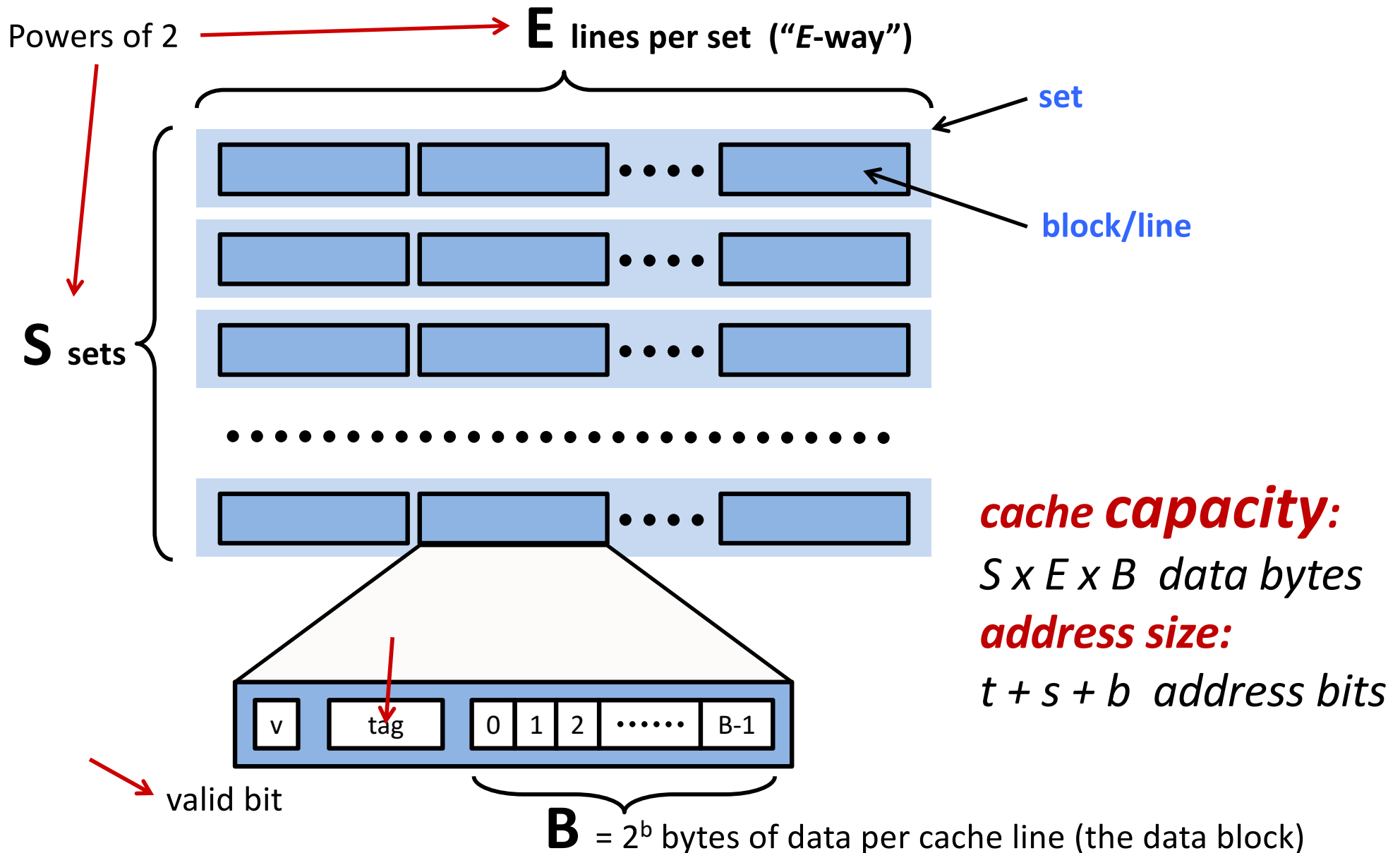
Another puzzle: Cache starts *empty*, uses LRU.

Access (address, hit/miss) stream

(10, miss); (12, miss); (10, miss)

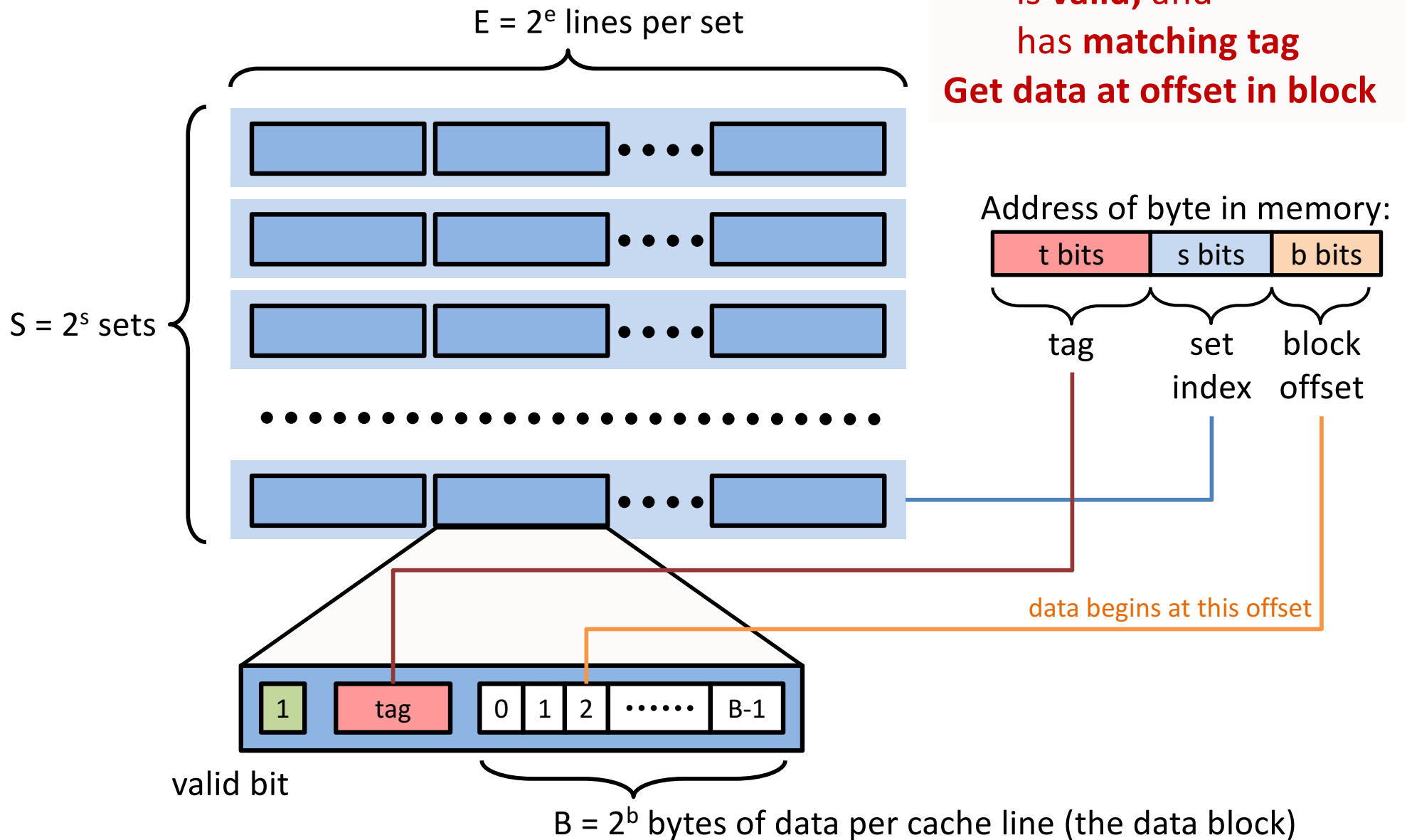
associativity of cache?

General Cache Organization (S, E, B)



Cache Read

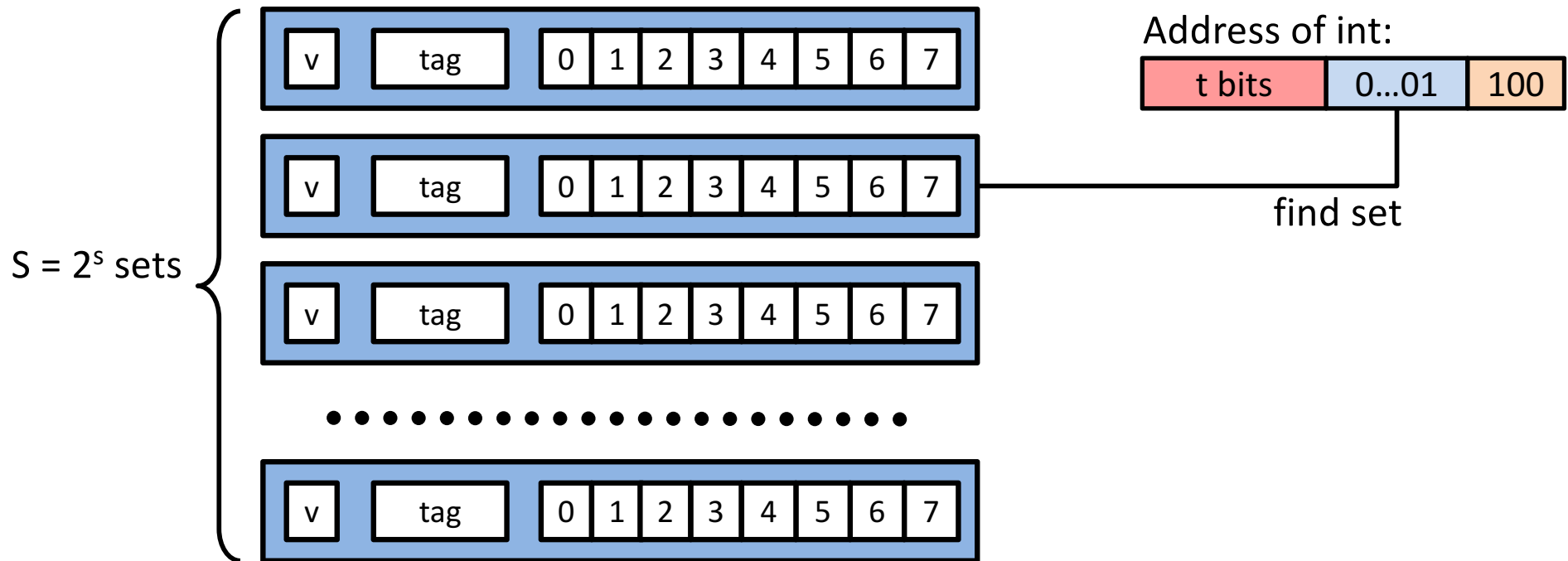
Locate set by index
Hit if any block in set:
 is valid; and
 has matching tag
Get data at offset in block



Cache Read: Direct-Mapped ($E = 1$)

This cache:

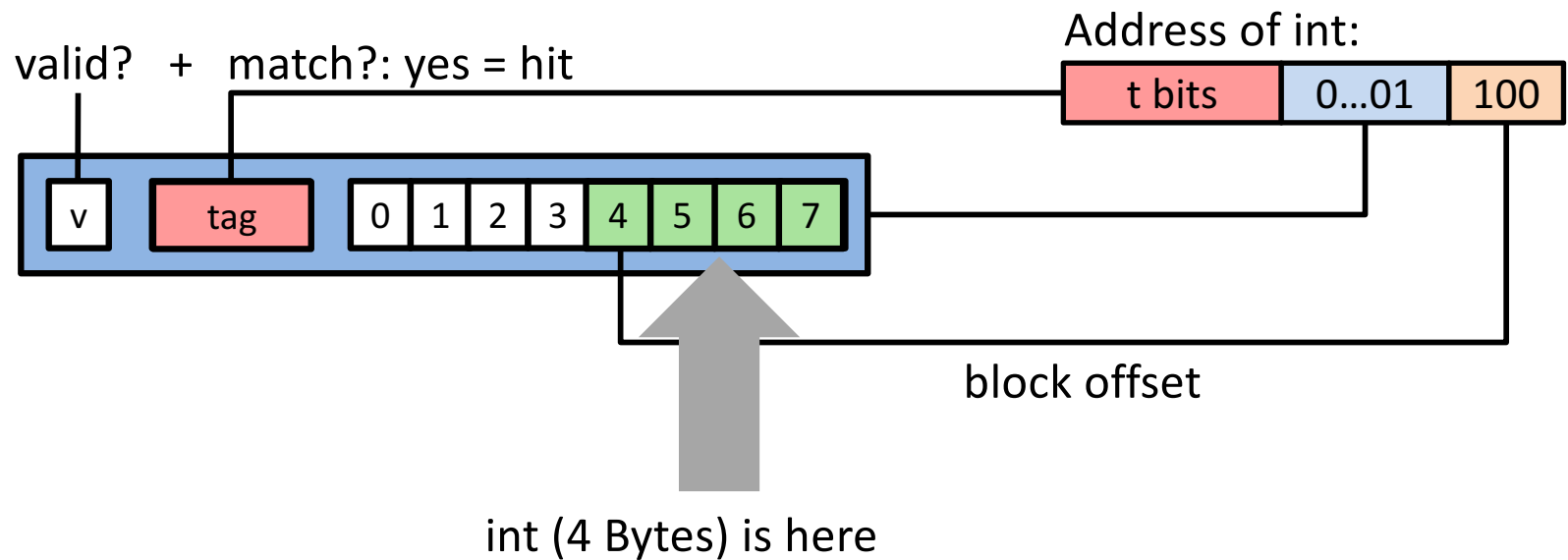
- Block size: 8 bytes
- Associativity: 1 block per set (direct mapped)



Cache Read: Direct-Mapped ($E = 1$)

This cache:

- Block size: 8 bytes
- Associativity: 1 block per set (direct mapped)



If no match: old line is evicted and replaced

Direct-Mapped Cache Practice

12-bit address

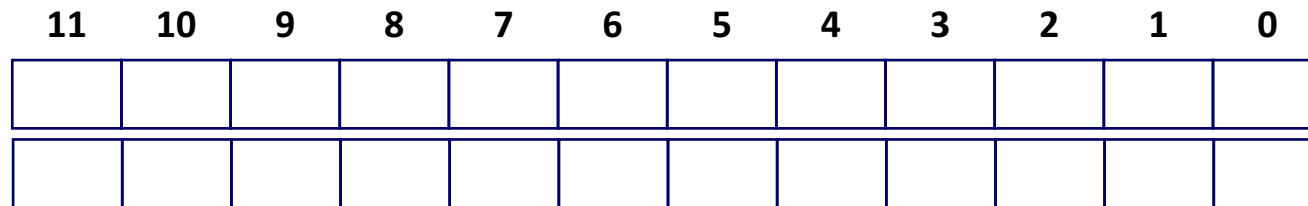
0x354

16 lines, 4-byte block size

0xA20

Direct mapped

Offset bits? Index bits? Tag bits?



<i>Index</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

<i>Index</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Example (E = 1)

Locals in registers.

Assume **a** is aligned such that

&a[r][c] is **aa...a rrrr cccc 000**

```
int sum_array_rows(double a[16][16]) {
    double sum = 0;

    for (int r = 0; r < 16; r++) {
        for (int c = 0; c < 16; c++) {
            sum += a[r][c];
        }
    }
    return sum;
}
```

```
int sum_array_cols(double a[16][16]) {
    double sum = 0;

    for (int c = 0; c < 16; c++) {
        for (int r = 0; r < 16; r++) {
            sum += a[r][c];
        }
    }
    return sum;
}
```

Assume: cold (empty) cache
3-bit set index, 5-bit offset

aa...arrr rcc cc000

0,0: aa...a000 000 00000

0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,a	0,b
0,c	0,d	0,e	0,f
1,0	1,1	1,2	1,3
1,4	1,5	1,6	1,7
1,8	1,9	1,a	1,b
1,c	1,d	1,e	1,f

32 bytes = 4 doubles

4 misses per row of array
4*16 = 64 misses

32 bytes = 4 doubles
every access a miss
16*16 = 256 misses

0,0	0,1	0,2	0,3
3,0	3,1	3,2	3,3

Example (E = 1)

```
int dotprod(int x[8], int y[8]) {
    int sum = 0;

    for (int i = 0; i < 8; i++) {
        sum += x[i]*y[i];
    }
    return sum;
}
```

block = 16 bytes; 8 sets in cache
 How many block offset bits?
 How many set index bits?

Address bits:

B =

S =

Addresses as bits

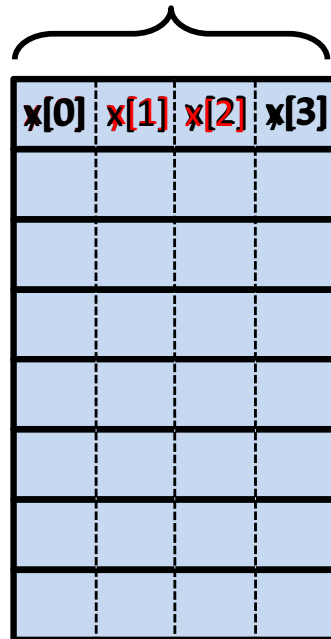
0x00000000:

0x00000080:

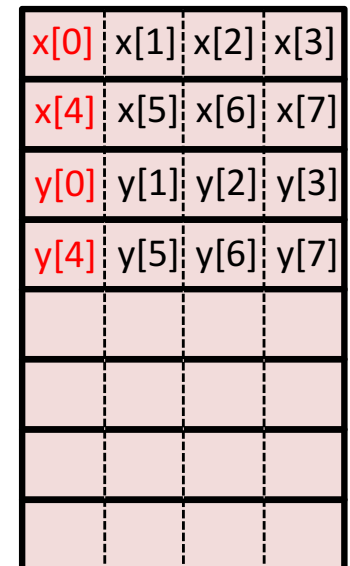
0x000000A0:

16 bytes = 4 ints

if x and y are mutually aligned,
 e.g., 0x00, 0x80



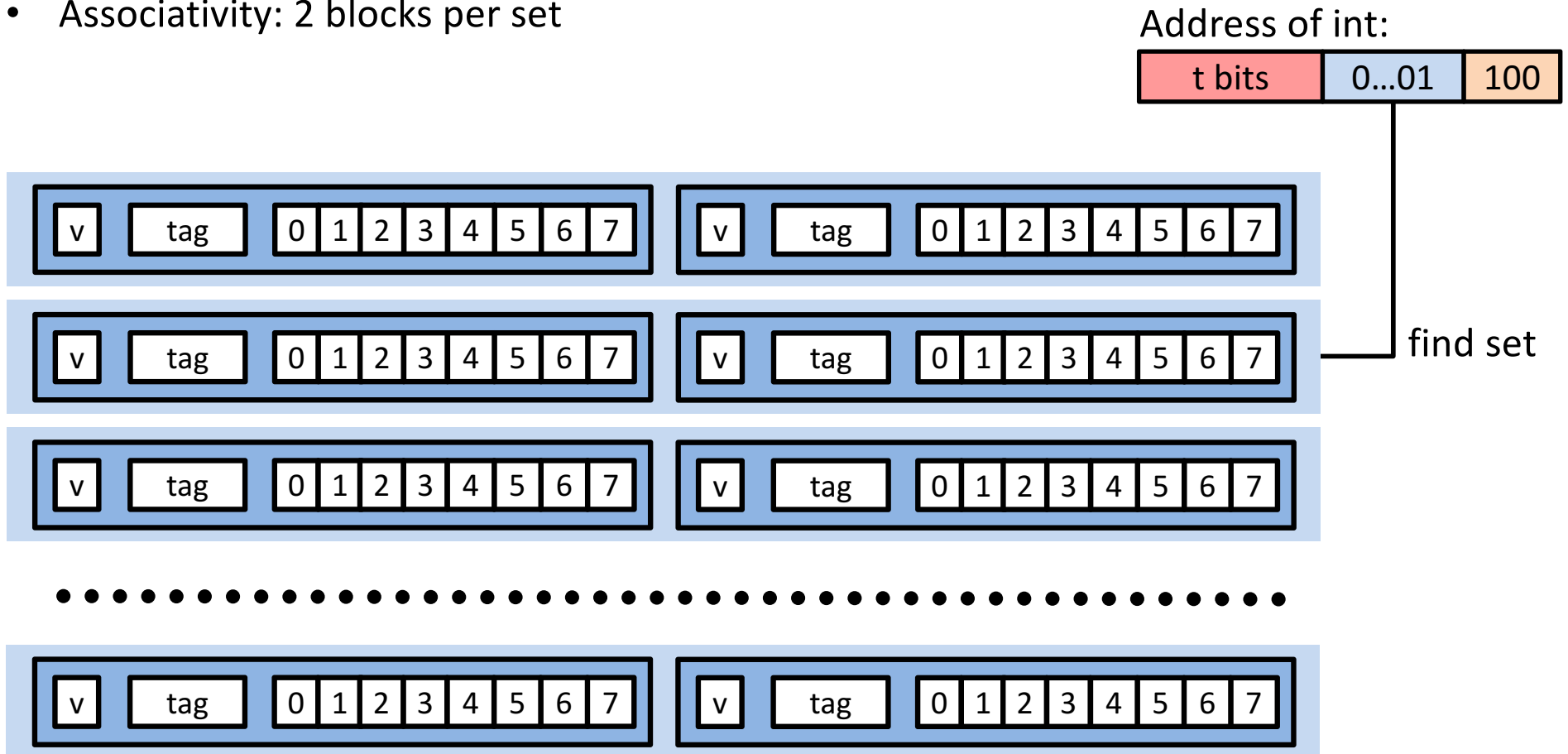
if x and y are mutually unaligned,
 e.g., 0x00, 0xA0



Cache Read: Set-Associative (Example: E = 2)

This cache:

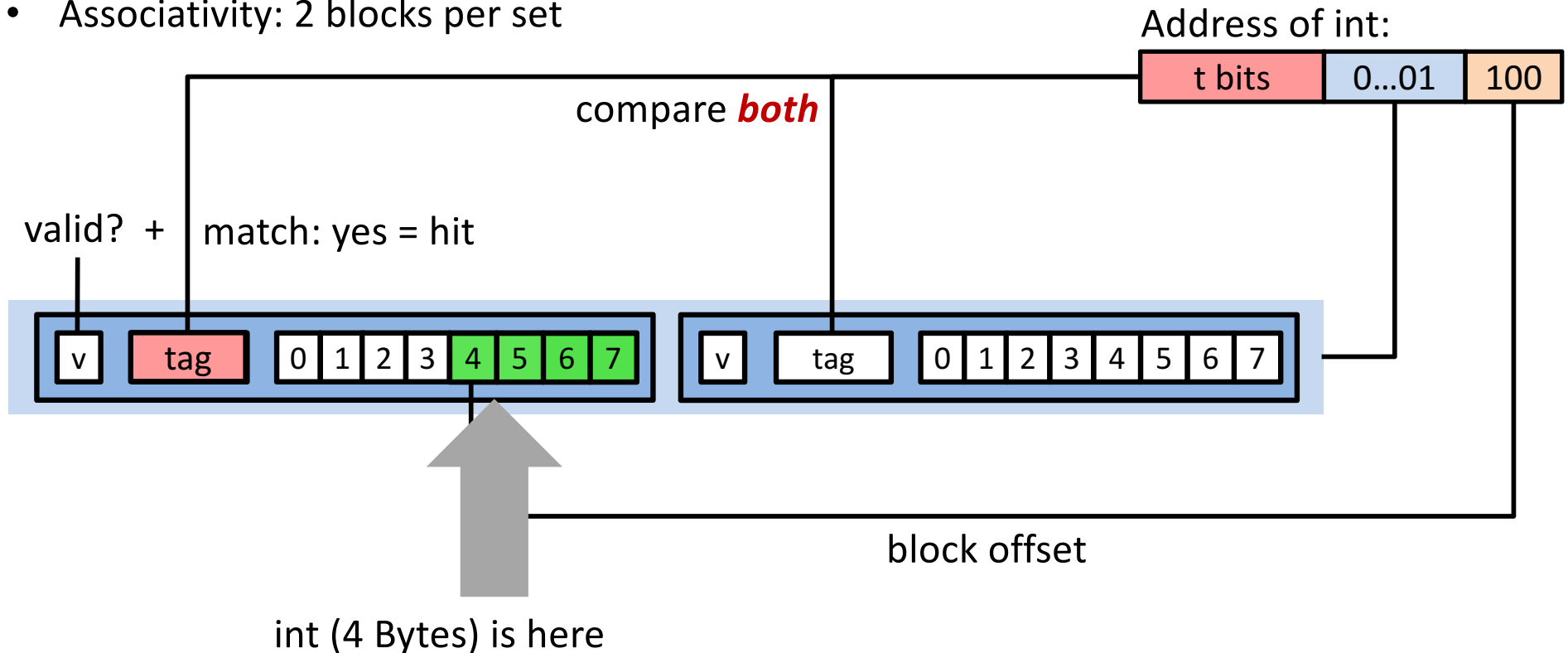
- Block size: 8 bytes
- Associativity: 2 blocks per set



Cache Read: Set-Associative (Example: E = 2)

This cache:

- Block size: 8 bytes
- Associativity: 2 blocks per set



If no match: Evict and replace one line in set.

Example (E = 2)

```
float dotprod(float x[8], float y[8]) {  
    float sum = 0;  
  
    for (int i = 0; i < 8; i++) {  
        sum += x[i]*y[i];  
    }  
    return sum;  
}
```

2 blocks/lines per set

If x and y aligned,
e.g. $\&x[0] = 0$, $\&y[0] = 128$,
can still fit both because each set
has space for two blocks/lines



4 sets

Types of Cache Misses

Cold (compulsory) miss

Conflict miss

Capacity miss

Which ones can we mitigate/eliminate? How?

Writing to cache

Multiple copies of data exist, must be kept in sync.

Write-hit policy

Write-through:

Write-back: needs a *dirty bit*

Write-miss policy

Write-allocate:

No-write-allocate:

Typical caches:

Write-back + Write-allocate, usually

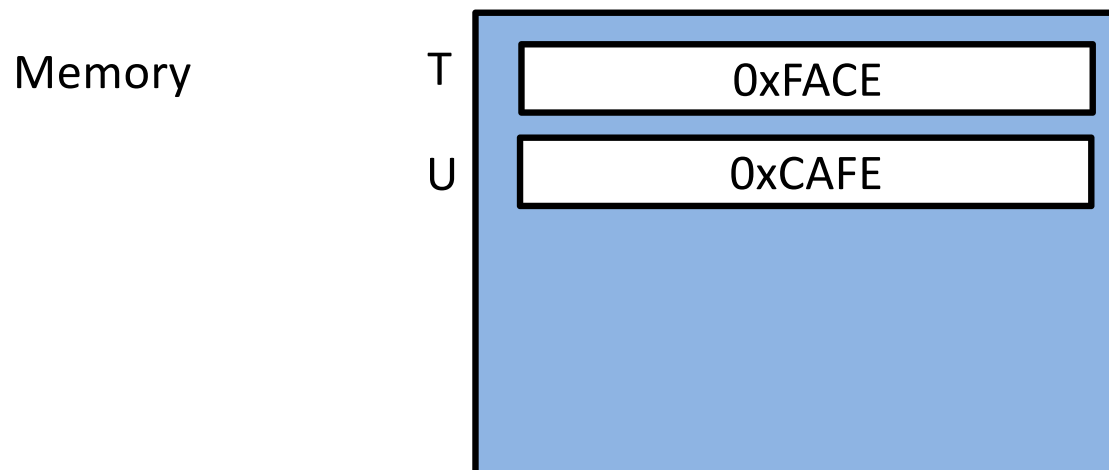
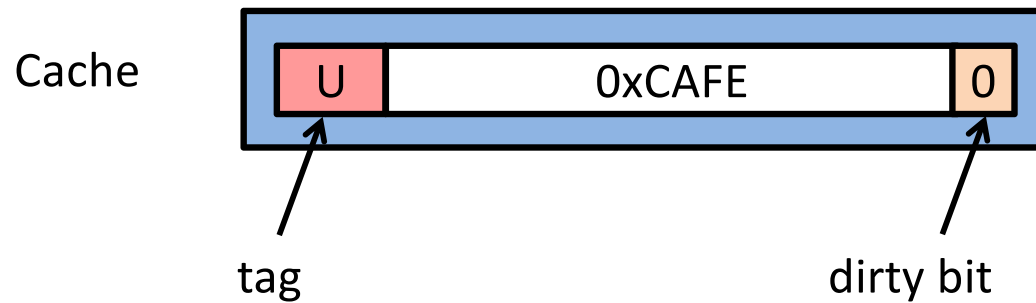
Write-through + No-write-allocate, occasionally

Write-back, write-allocate example

Cache/memory not involved

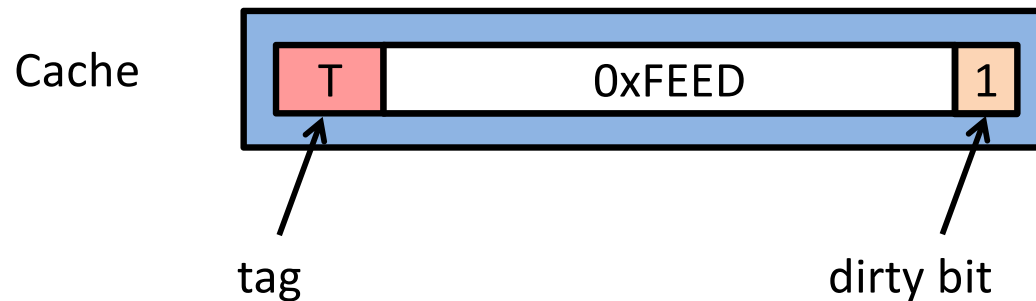
```
eax =  
ecx = T  
edx = U
```

1. `mov $T, %ecx`
2. `mov $U, %edx`
3. `mov $0xFEED, (%ecx)`
 - a. Miss on T.



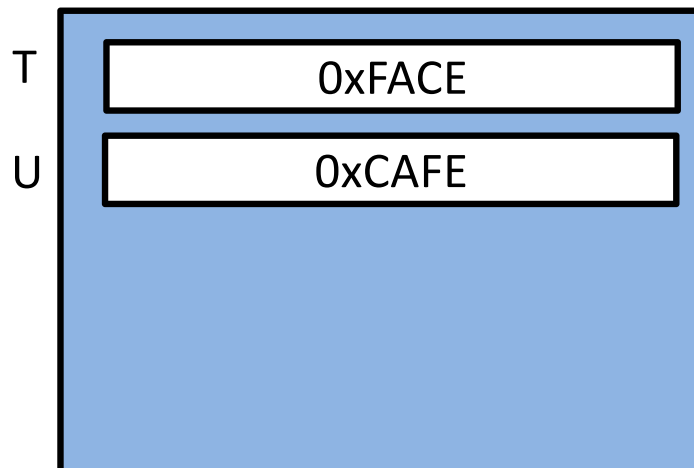
Write-back, write-allocate example

eax =
ecx = T
edx = U



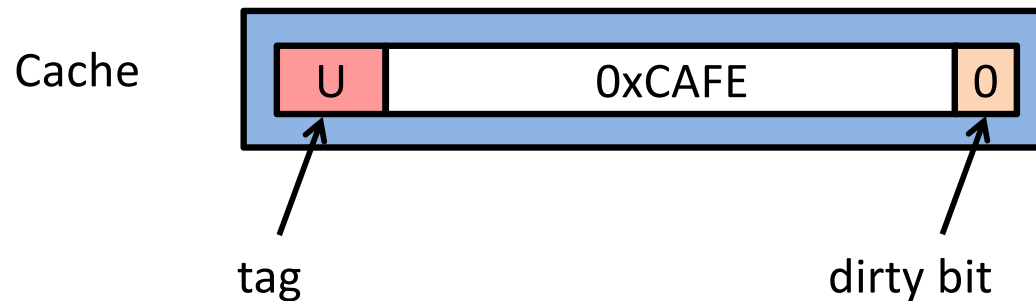
1. `mov $T, %ecx`
2. `mov $U, %edx`
3. `mov $0xFEED, (%ecx)`
 - a. Miss on T.
 - b. Evict U (clean: discard).
 - c. Fill T (write-allocate).
 - d. Write T in cache (dirty).
4. `mov (%edx), %eax`
 - a. Miss on U.

Memory

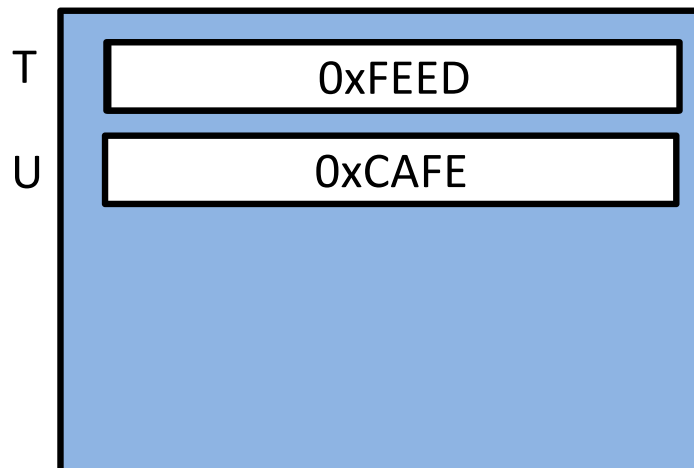


Write-back, write-allocate example

eax = 0xCAFE
ecx = T
edx = U



Memory

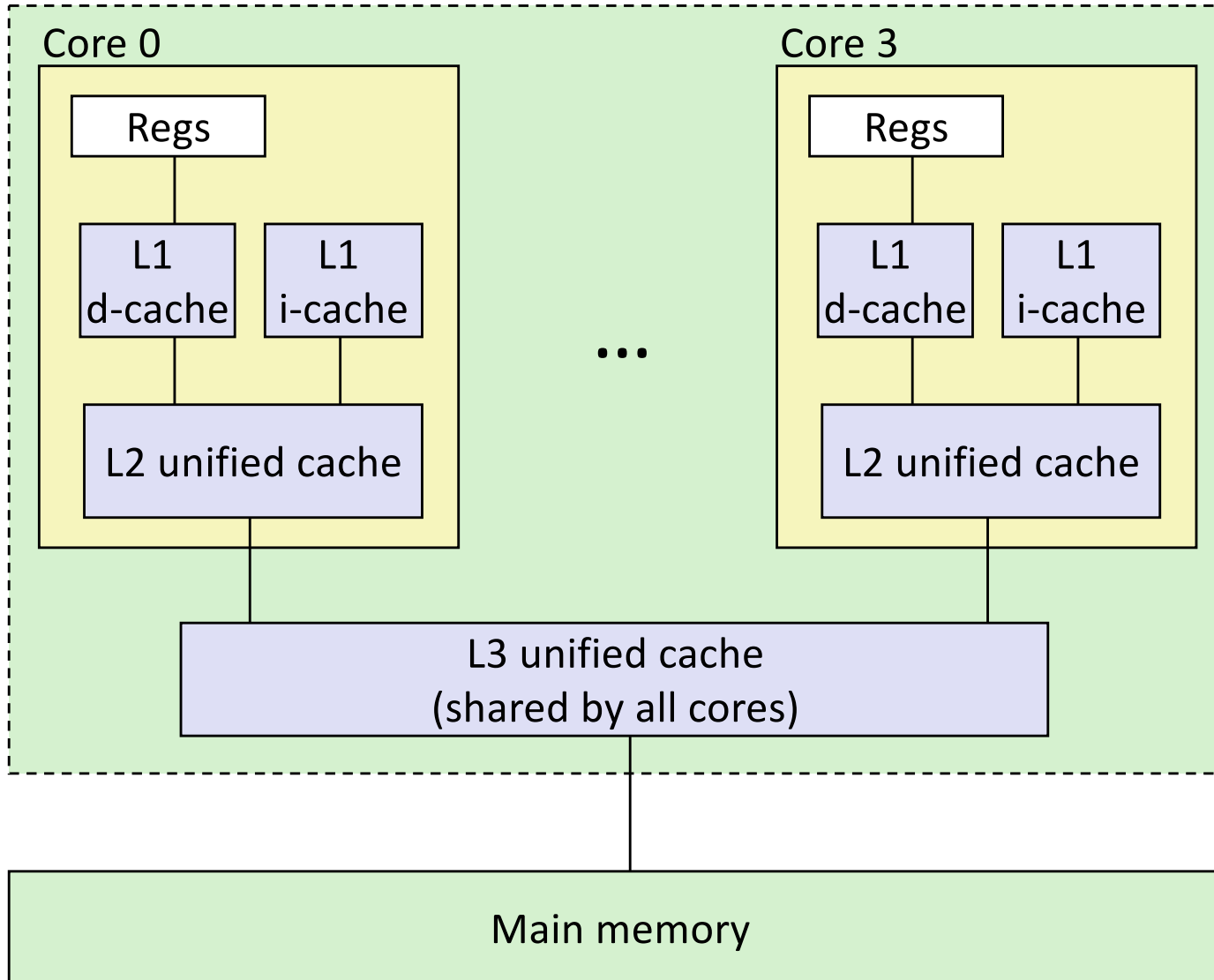


1. `mov $T, %ecx`
2. `mov $U, %edx`
3. `mov $0xFEED, (%ecx)`
 - a. Miss on T.
 - b. Evict U (clean: discard).
 - c. Fill T (write-allocate).
 - d. Write T in cache (dirty).
4. `mov (%edx), %eax`
 - a. Miss on U.
 - b. Evict T (dirty: write back).
 - c. Fill U.
 - d. Set %eax.
5. **DONE.**

Example Memory Hierarchy

Typical laptop/desktop processor
(c.a. 201_)

Processor package



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for
all caches.

slower, but
more likely
to hit

Aside: software caches

Examples

File system buffer caches, web browser caches, database caches, network CDN caches, etc.

Some design differences

Almost always fully-associative

Often use complex replacement policies

Not necessarily constrained to single “block” transfers

Cache-Friendly Code

Locality, locality, locality.

Programmer can optimize for cache performance

- Data structure layout

- Data access patterns

 - Nested loops

 - Blocking (see CSAPP 6.5)

All systems favor “cache-friendly code”

- Performance is hardware-specific

- Generic rules capture most advantages

 - Keep working set small (temporal locality)

 - Use small strides (spatial locality)

 - Focus on inner loop code