

# COMP201

## Computer Systems & Programming

Lecture #22 – More Cache Memories



**KOÇ**  
**UNIVERSITY**

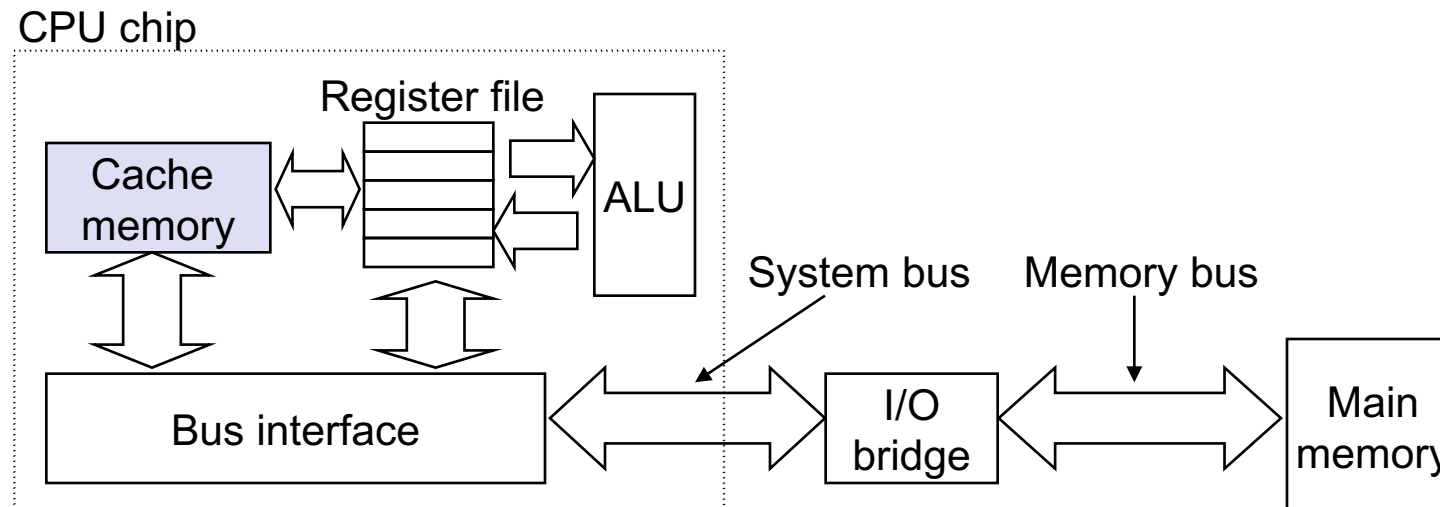
Aykut Erdem // Koc University // Spring 2023

# Recap

- Cache basics
- Principle of locality
- Cache memory organization and operation

# Recap: Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:

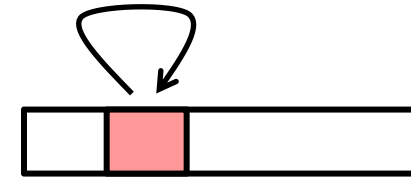


# Recap: Why Caches Work

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

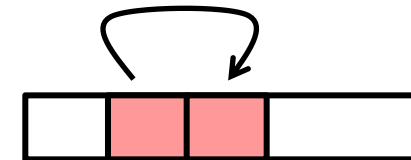
- **Temporal locality:**

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



- **Spatial locality:**

- Items with nearby addresses tend to be referenced close together in time



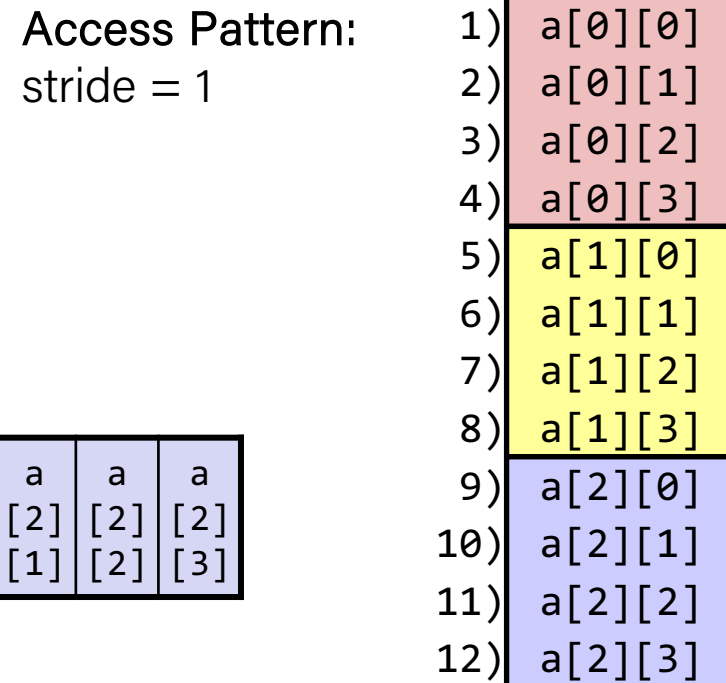
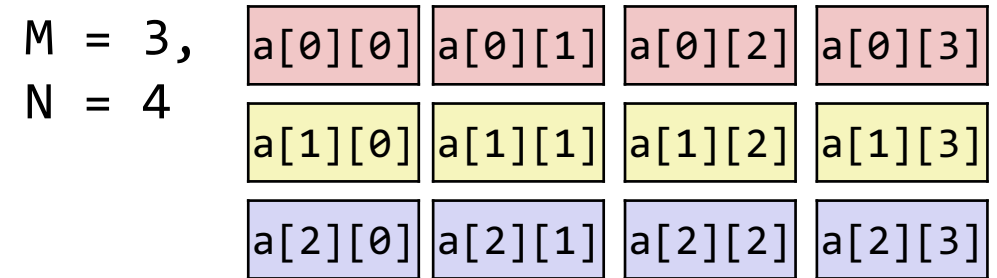
# Recap: Good Locality Example

- Does this function have good locality with respect to array `a`?

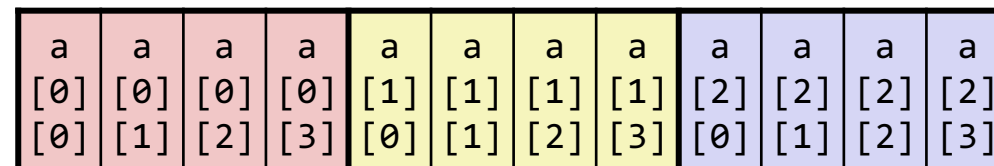
```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

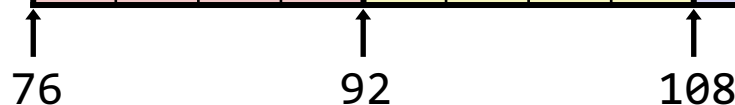
    return sum;
}
```



Layout in Memory



Note: 76 is just one possible starting address of array `a`

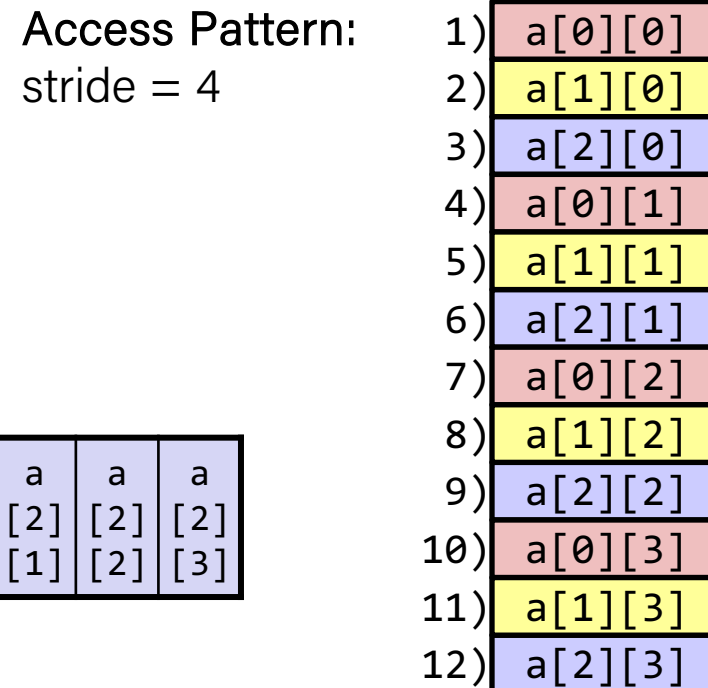
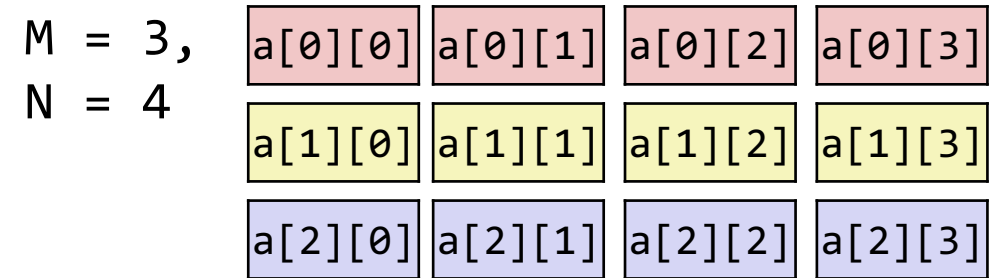


# Recap: Bad Locality Example

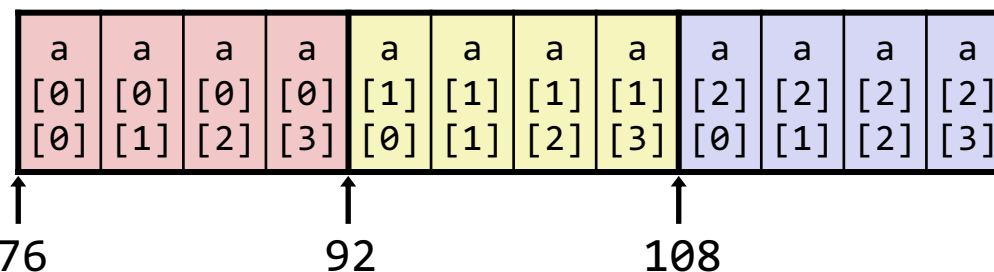
- Does this function have good locality with respect to array a?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

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



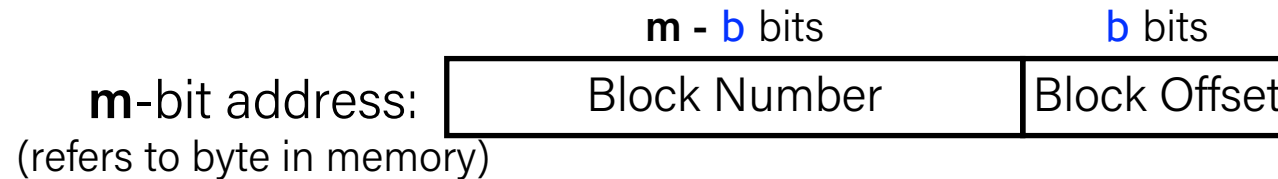
Layout in Memory



Note: 76 is just one possible starting address of array a

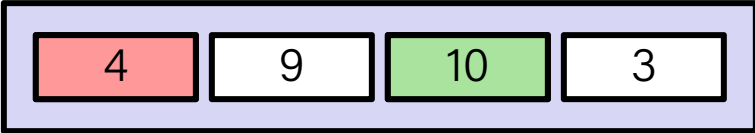
# Recap: Cache Organization

- **Block Size (B):** unit of transfer between cache and main memory
  - Given in bytes and always a power of 2 (e.g. 64 bytes)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!
- Offset field
  - Low-order  $\log_2(B) = b$  bits of address tell you which byte within a block
    - $(\text{address}) \bmod 2^n = n$  lowest bits of address
  - $(\text{address}) \bmod (\# \text{ of bytes in a block})$

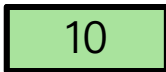


# Recap: Cache Organization

Cache

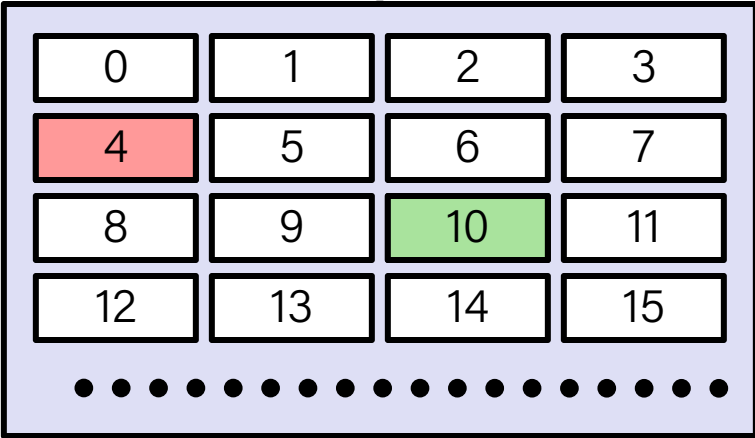


Smaller, faster, more expensive memory caches a subset of the blocks



Data is copied in block-sized transfer units

Memory



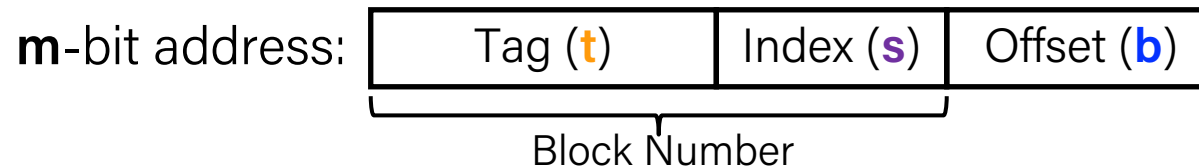
Larger, slower, cheaper memory viewed as partitioned into "blocks"



# Recap: Checking for a Requested Address

- CPU sends address request for chunk of data
  - Address and requested data are not the same thing!
    - Analogy: your friend  $\neq$  their phone number

- TIO address breakdown:

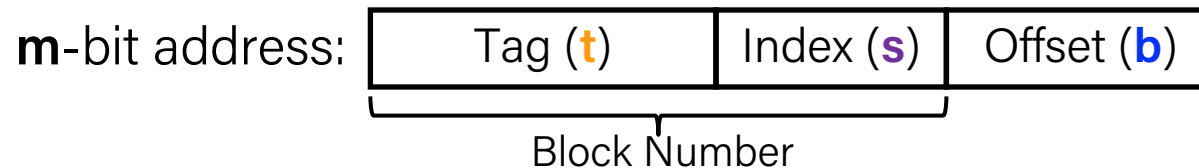


- **Index** field tells you where to look in cache
- **Tag** field lets you check that data is the block you want
- **Offset** field selects specified start byte within block

- **Note:** **t** and **s** sizes will change based on hash function

# Recap: Checking for a Requested Address

- Using 8-bit addresses.
- Cache Params: block size (B) = 4 bytes, cache size (C) = 32 bytes (which means number of sets is  $C/B = 8$  sets).
  - Offset bits (b) =  $\log_2(B) = 2$  bits
  - Index bits (s) =  $\log_2(\text{number of sets}) = 3$  bits
  - Tag bits (t) = Rest of the bits in the address =  $8 - 2 - 3 = 3$  bits



- What are the fields for address 0xBA?
  - Tag bits (unique id for block): 0x5      101 110 10
  - Index bits (cache set block maps to): 0x6      5   6   2
  - Offset bits (byte offset within block): 0x2

# Plan for Today

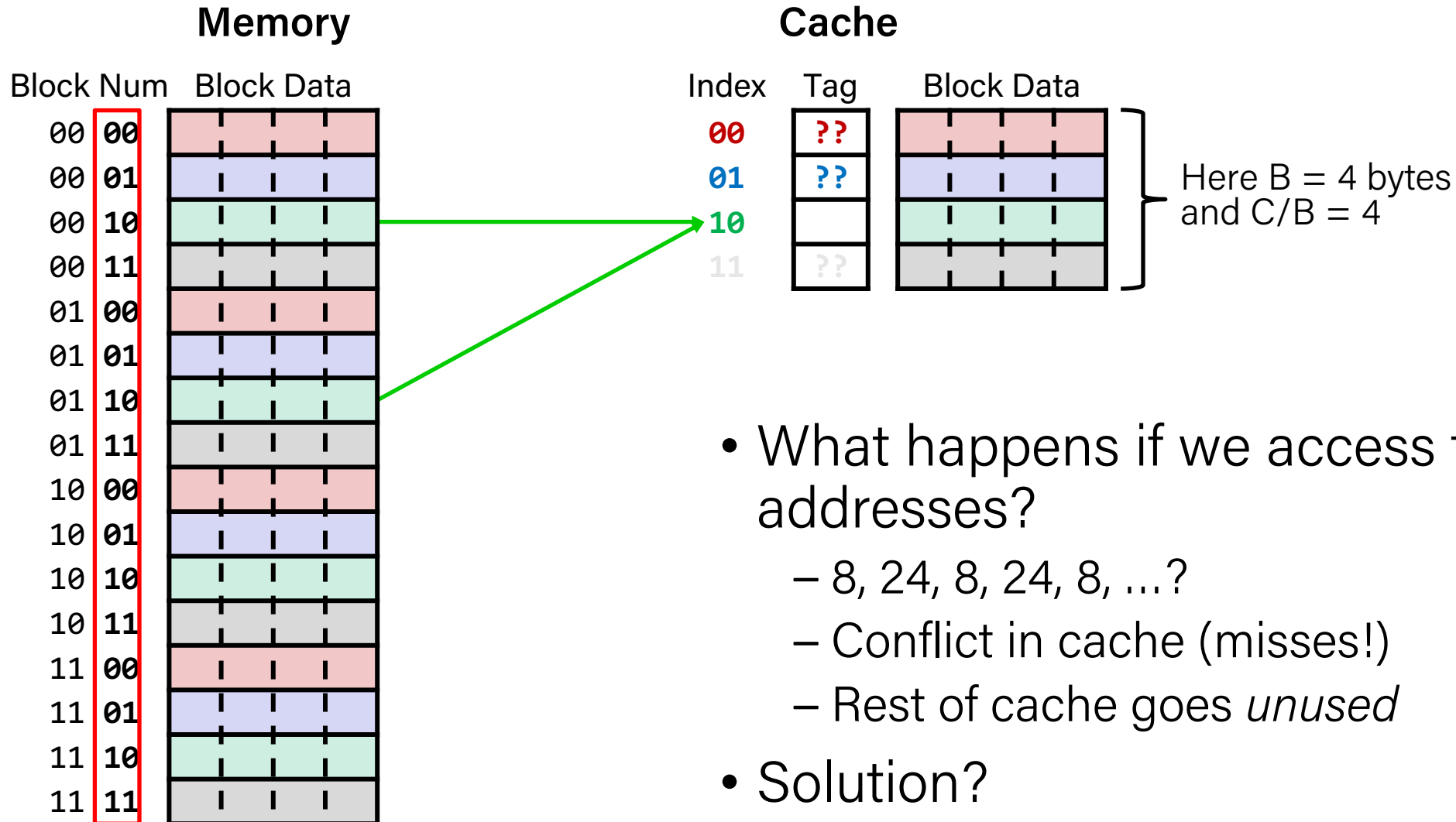
- Cache memory organization and operation
- Memory Mountain

**Disclaimer:** Slides for this lecture were borrowed from  
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class  
—Porter Jones' UW CSE 351 class

# Lecture Plan

- Cache memory organization and operation
- Memory Mountain

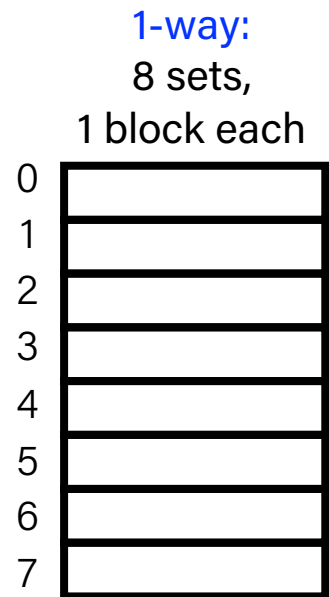
# Direct-Mapped Cache Problem



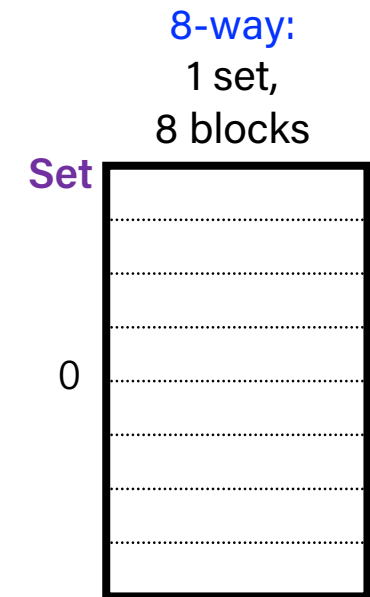
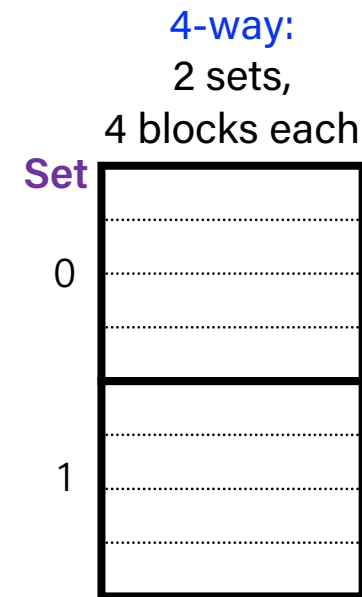
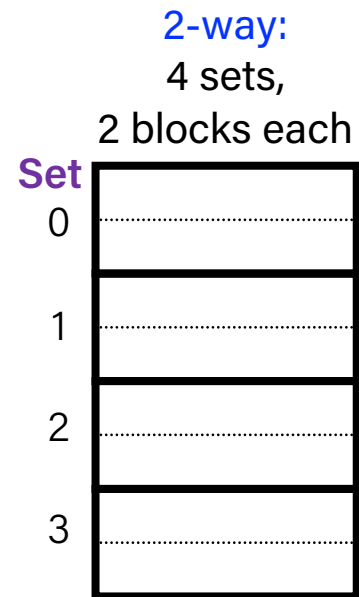
- What happens if we access the following addresses?
  - 8, 24, 8, 24, 8, ...?
  - Conflict in cache (misses!)
  - Rest of cache goes *unused*
- Solution?

# Associativity

- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we *combine* the two ideas:
  - Each address maps to exactly one **set**
  - Each set can store block in more than one **way**



direct-mapped



fully associative

# Cache Puzzle

- Based on the following behavior, which of the following block sizes is NOT possible for our cache?
  - Cache starts *empty*, also known as a **cold cache**
  - Access (addr: hit/miss) stream:
    - (14: miss), (15: hit), (16: miss)
- A. 4 bytes
- B. 8 bytes
- C. 16 bytes
- D. 32 bytes
- E. We're lost...

# Cache Puzzle



- Based on the following behavior, which of the following block size is NOT possible for our cache?

- Cache starts *empty*, also known as a **cold cache**
- Access (addr: hit/miss) stream:
  - (14: miss), (15: hit), (16: miss)

hit: block is already in cache!  
miss: block is not in cache, pulls block from memory and puts it in cache

- ① Pulls block /w 14 into cache
- ② 15 is in the same block at 14
- ③ 16 is not in block w/ 14 and 15

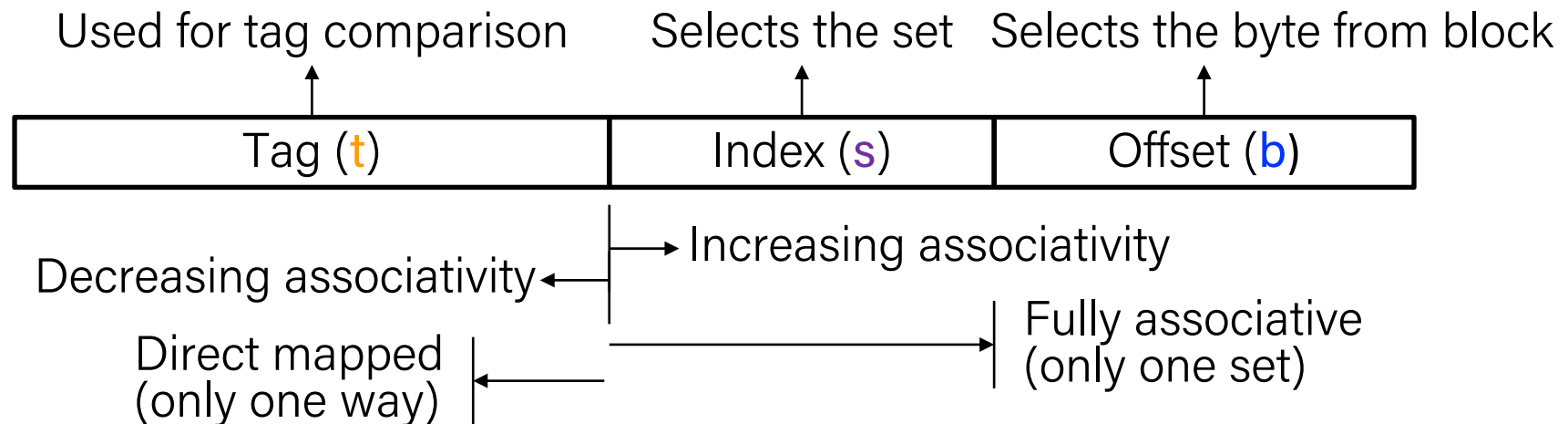
- A. 4 bytes
- B. 8 bytes
- C. 16 bytes
- D. 32 bytes**
- E. We're lost...

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
Mem																			
K = 4													X	✓	X				
K = 8																	X	✓	X
K = 16																	X	✓	X
K = 32																	X	✓	✓



# Cache Organization

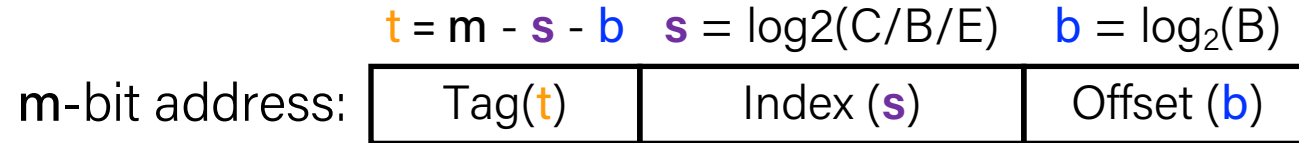
- **Associativity (E)**: # of ways for each set
  - Such a cache is called an "*E-way set associative cache*"
  - We now index into cache sets, of which there are  $S = C/B/E$
  - Use lowest  $\log_2(C/B/E) = s$  bits of block address
    - Direct-mapped:  $E = 1$ , so  $s = \log_2(C/B)$  as we saw previously
    - Fully associative:  $E = C/B$ , so  $s = 0$  bits



# Example Placement

block size:	16 bytes
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`



**s** = ?  
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

**s** = ?  
2-way set associative

Set	Tag	Data
0		
1		
2		
3		

**s** = ?  
4-way set associative

Set	Tag	Data
0		
1		

# Example Placement

block size:	16 bytes
capacity:	8 blocks
address:	16 bits

- Where would data from address 0x1833 be placed?

– Binary: 0b 0001 1000 0011 0011  
E=4  
E=2  
E=1

$$t = m - s - b \quad s = \log_2(C/B/E) \quad b = \log_2(B)$$



**s** = log<sub>2</sub>(8)=3 bits  
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		✓
4		
5		
6		
7		

**s** = log<sub>2</sub>(8/2)=2 bits  
2-way set associative

Set	Tag	Data
0		
1		
2		
3		✓
		✓

**s** = log<sub>2</sub>(8/4)=1 bit  
4-way set associative

Set	Tag	Data
0		
1		✓
		✓
		✓
		✓

# Block Placement

- *Any* empty block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches
  - Caches typically use something close to **least recently used (LRU)** (hardware usually implements "*not most recently used*")

Direct-mapped

	Set	Tag	Data
0			
1			
2			
3			✓
4			
5			
6			
7			

2-way set associative

	Set	Tag	Data
0			
1			
2			
3			✓
			✓

4-way set associative

	Set	Tag	Data
0			
1			✓
			✓
			✓
			✓

# Question

- We have a cache of size 2 KB with block size of 128 bytes. If our cache has 2 sets, what is its associativity?
  - A. 2
  - B. 4
  - C. 8
  - D. 16
  - E. We're lost...
- If addresses are 16 bits wide, how wide is the Tag field?

# Question

$$(C = 2 \cdot 2^{10} \text{ bytes})$$

$$(B = 2^7 \text{ bytes})$$

- We have a cache of size 2 KB with block size of 128 bytes. If our cache has 2 sets, what is its associativity?

$$(S = 2)$$

A. 2

$$\text{num blocks} = C / K = 2^{11} / 2^7 = 2^4 = 16 \text{ blocks}$$

B. 4

**C. 8**

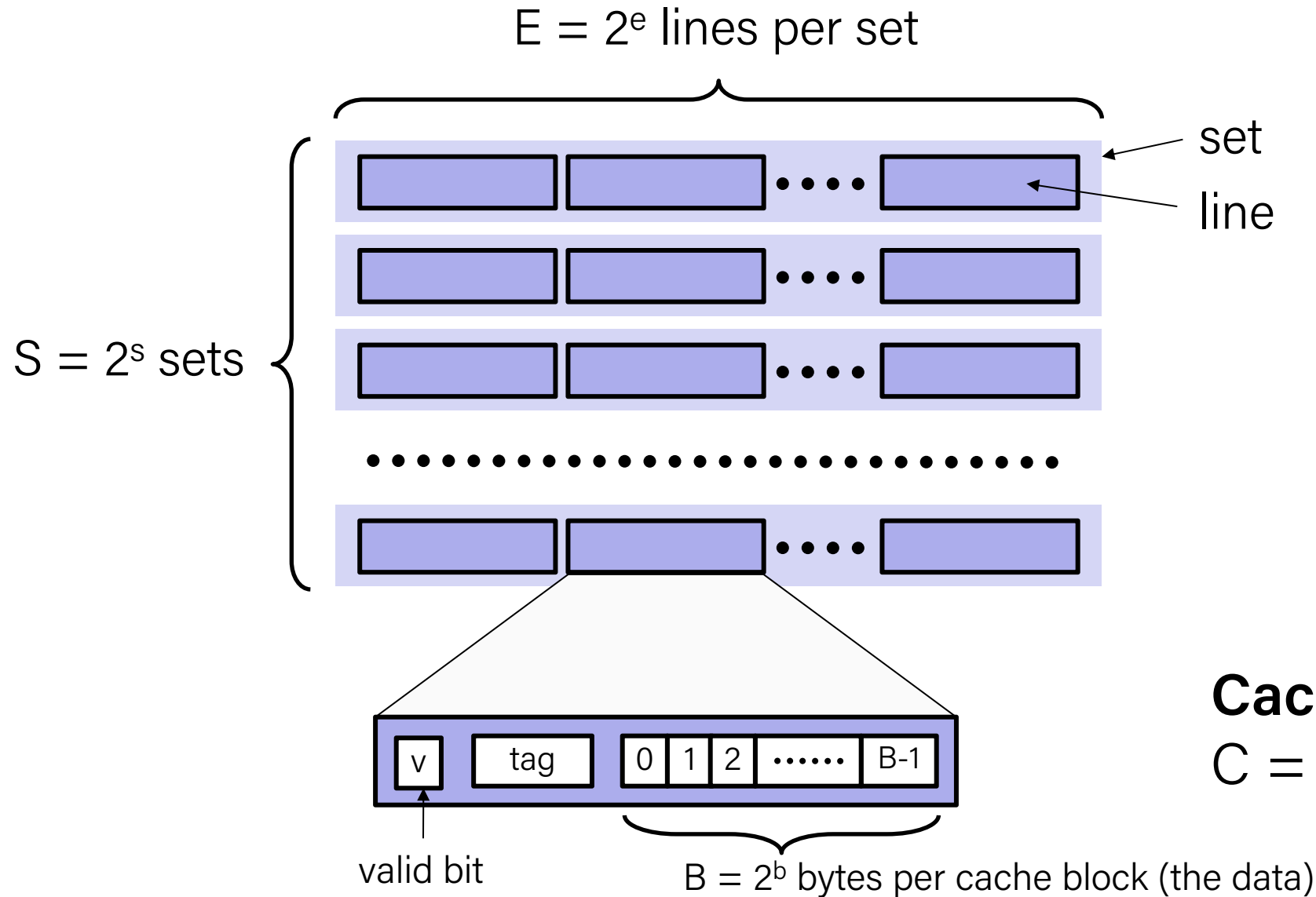
$$\text{blocks per set} = E = 16 / 2 = 8$$

D. 16

E. We're lost...

- If addresses are 16 bits wide, how wide is the Tag field?  $= 16 - 7 - 1 = 8$

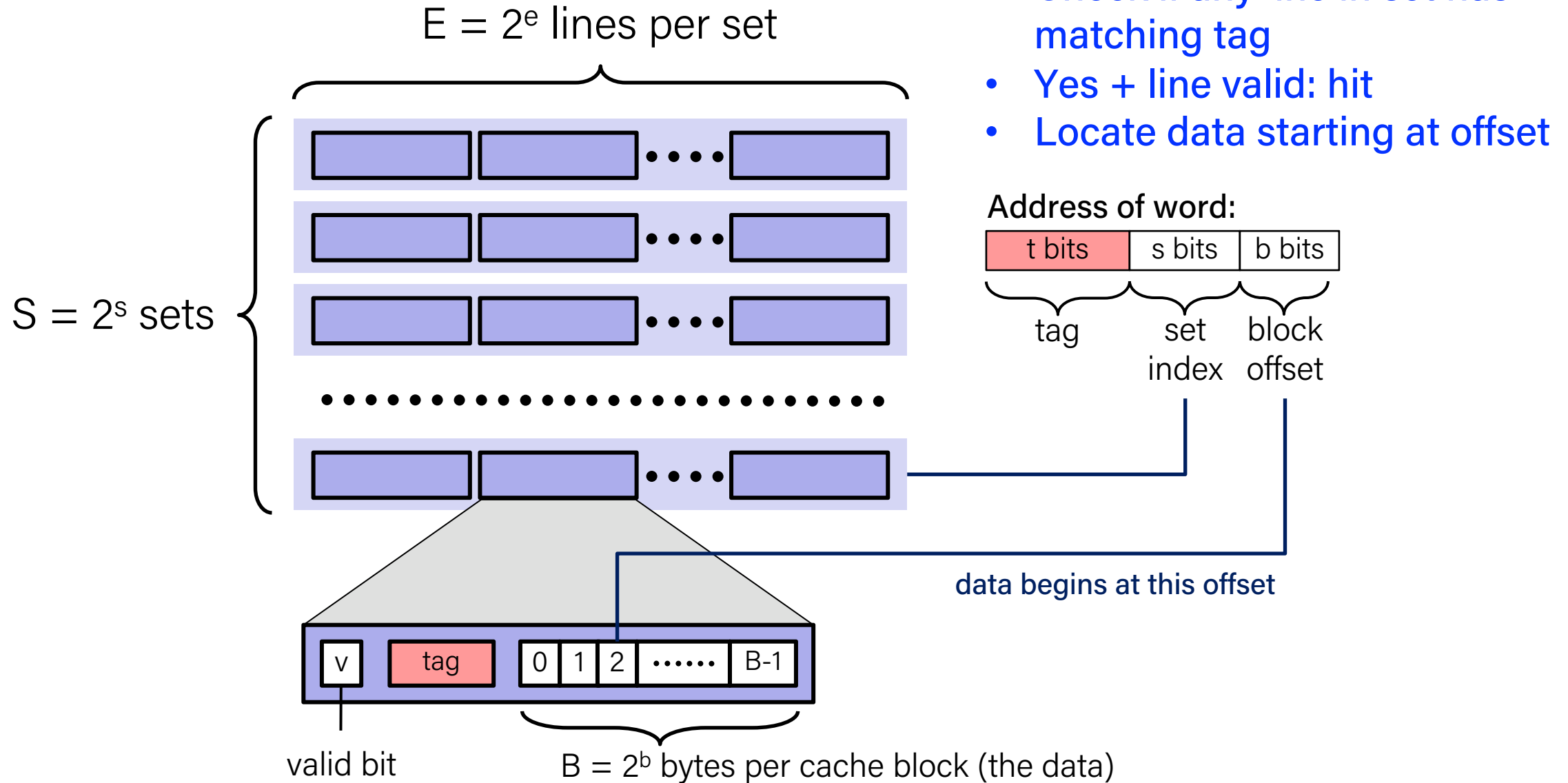
# General Cache Organization (S, E, B)



**Cache size:**

$$C = S \times E \times B \text{ data bytes}$$

# Cache Read



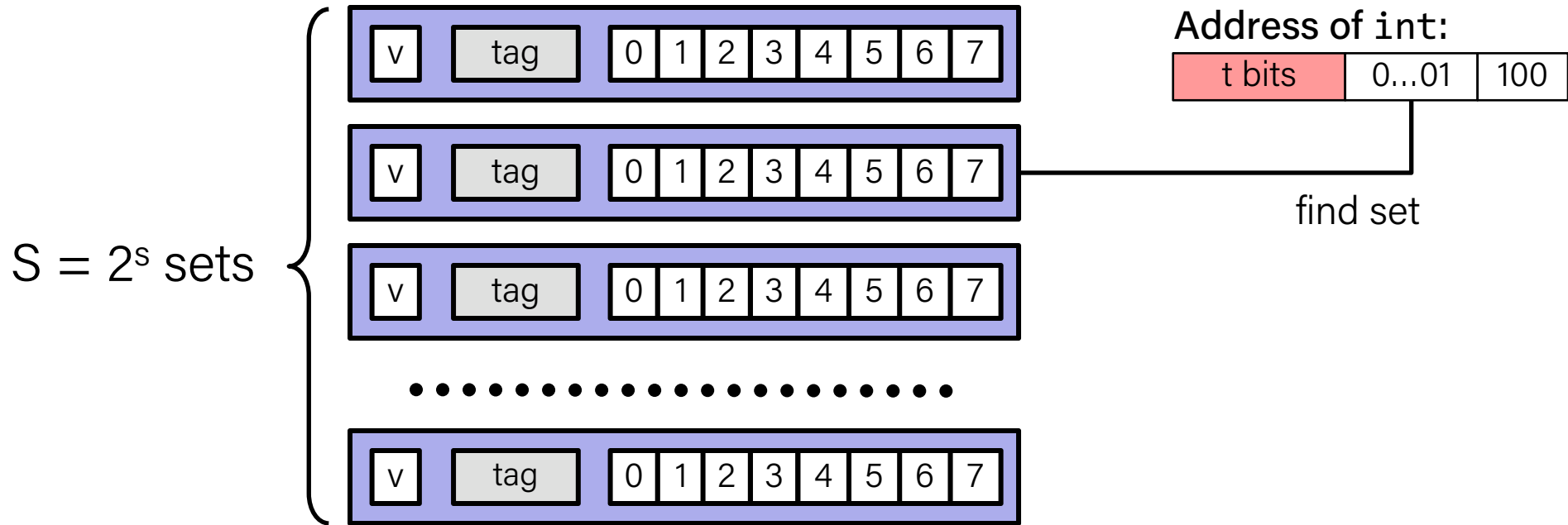
- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

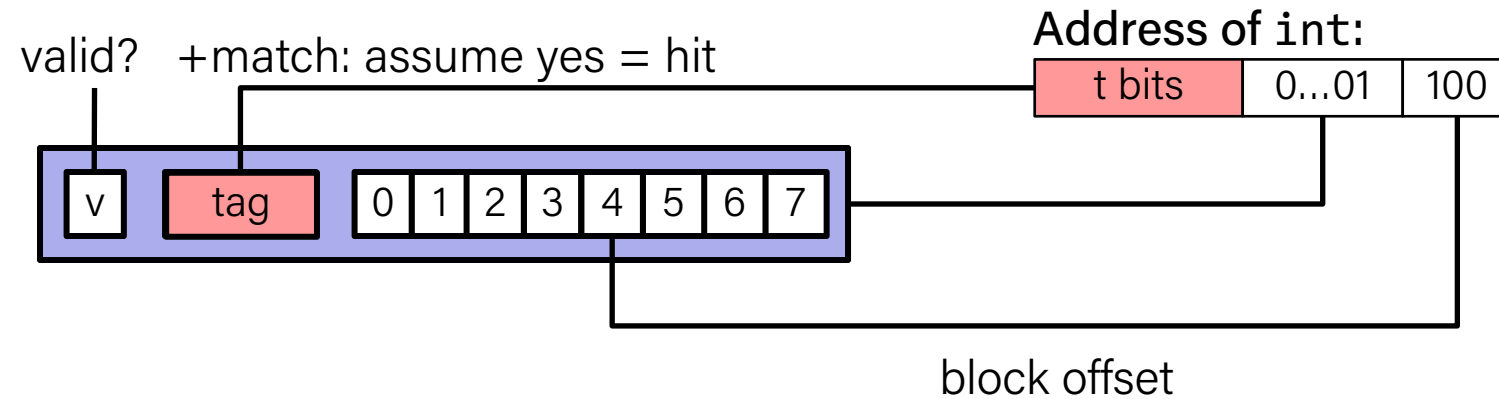
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

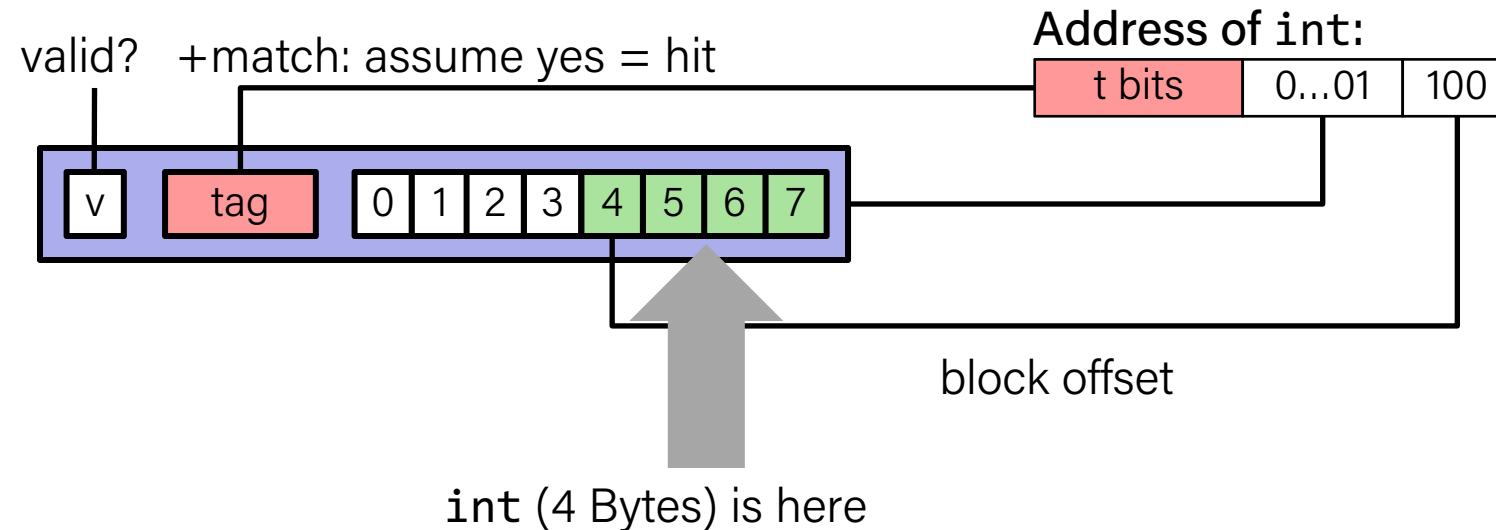
Direct mapped: One line per set

Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set  
Assume: cache block size 8 bytes



**If tag doesn't match:** old line is evicted and replaced

# Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block,  
S=4 sets, E=1 Blocks/set

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

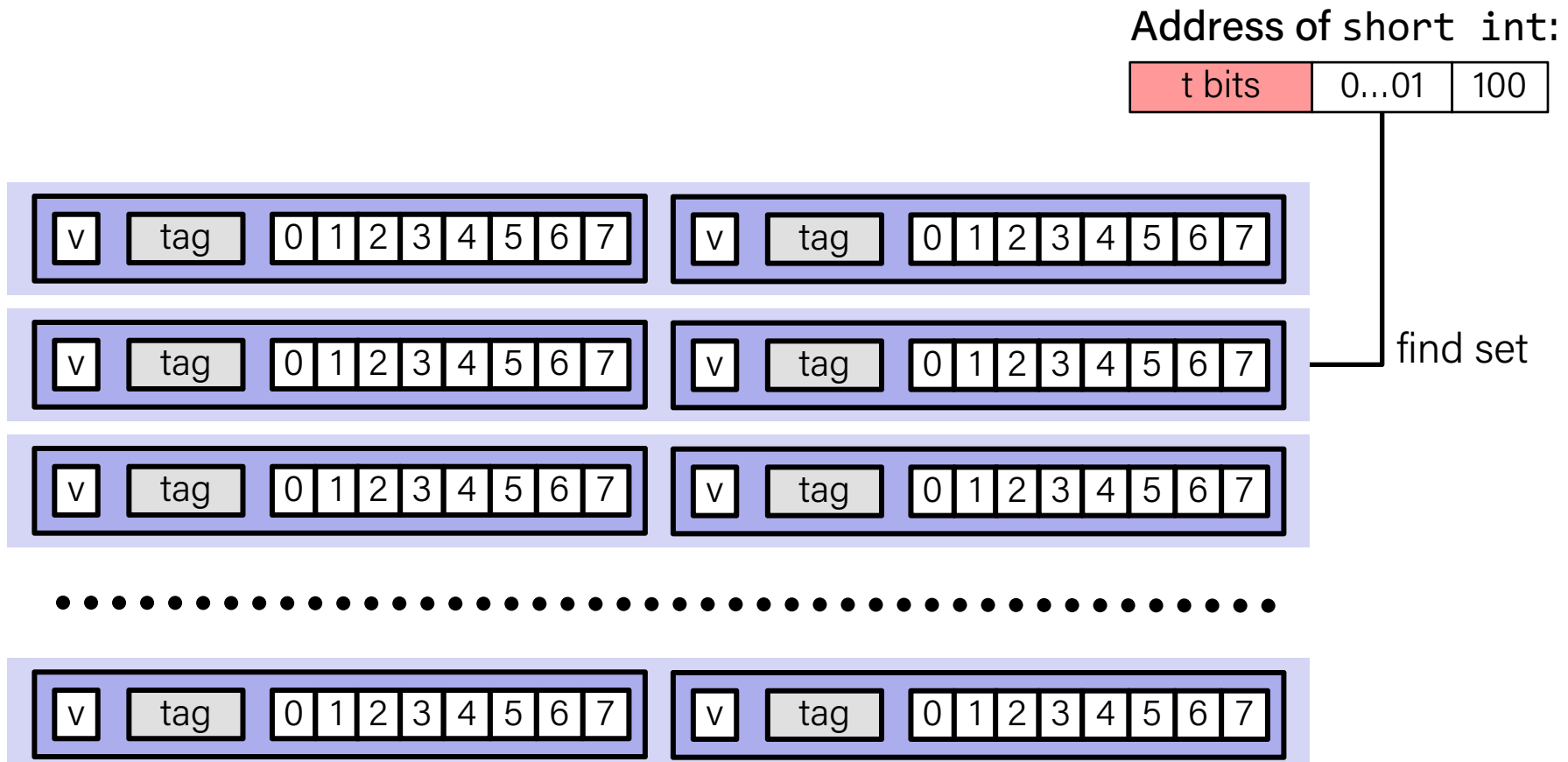
Address trace (reads, one byte per read):

0	[ <u>0000</u> <sub>2</sub> ],	miss
1	[ <u>0001</u> <sub>2</sub> ],	hit
7	[ <u>0111</u> <sub>2</sub> ],	miss
8	[ <u>1000</u> <sub>2</sub> ],	miss
0	[ <u>0000</u> <sub>2</sub> ]	miss

# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

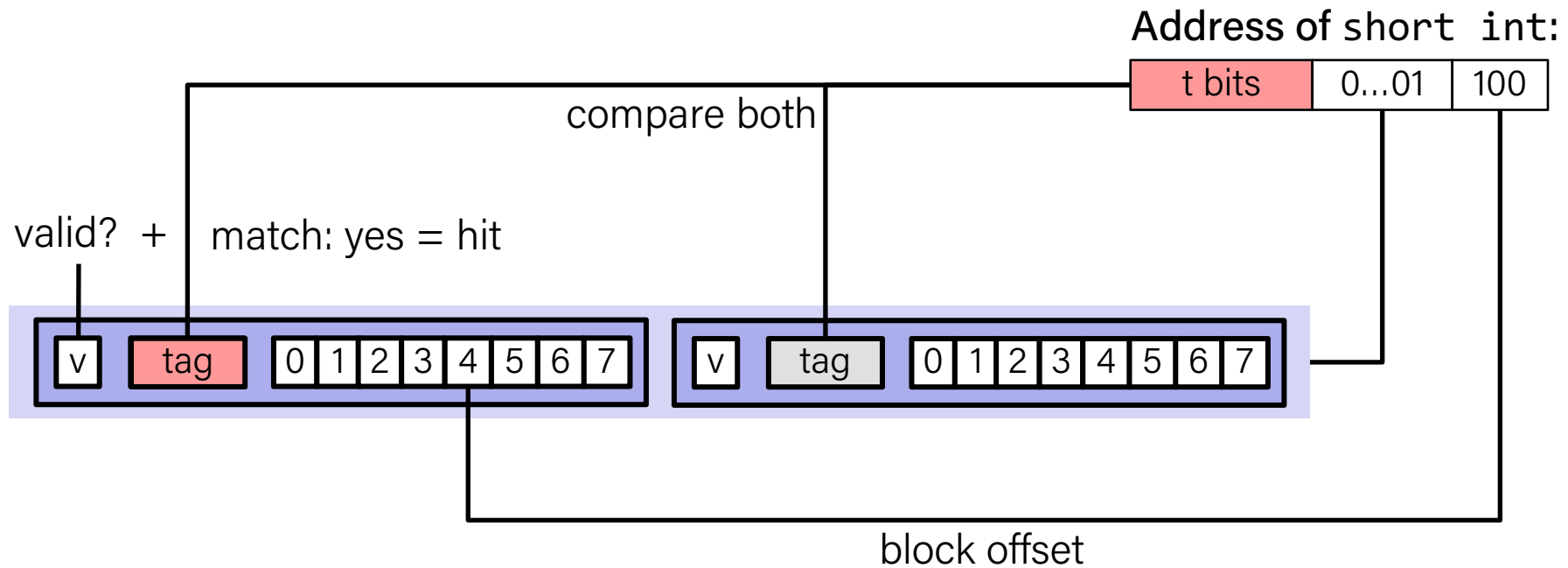
Assume: cache block size 8 bytes



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

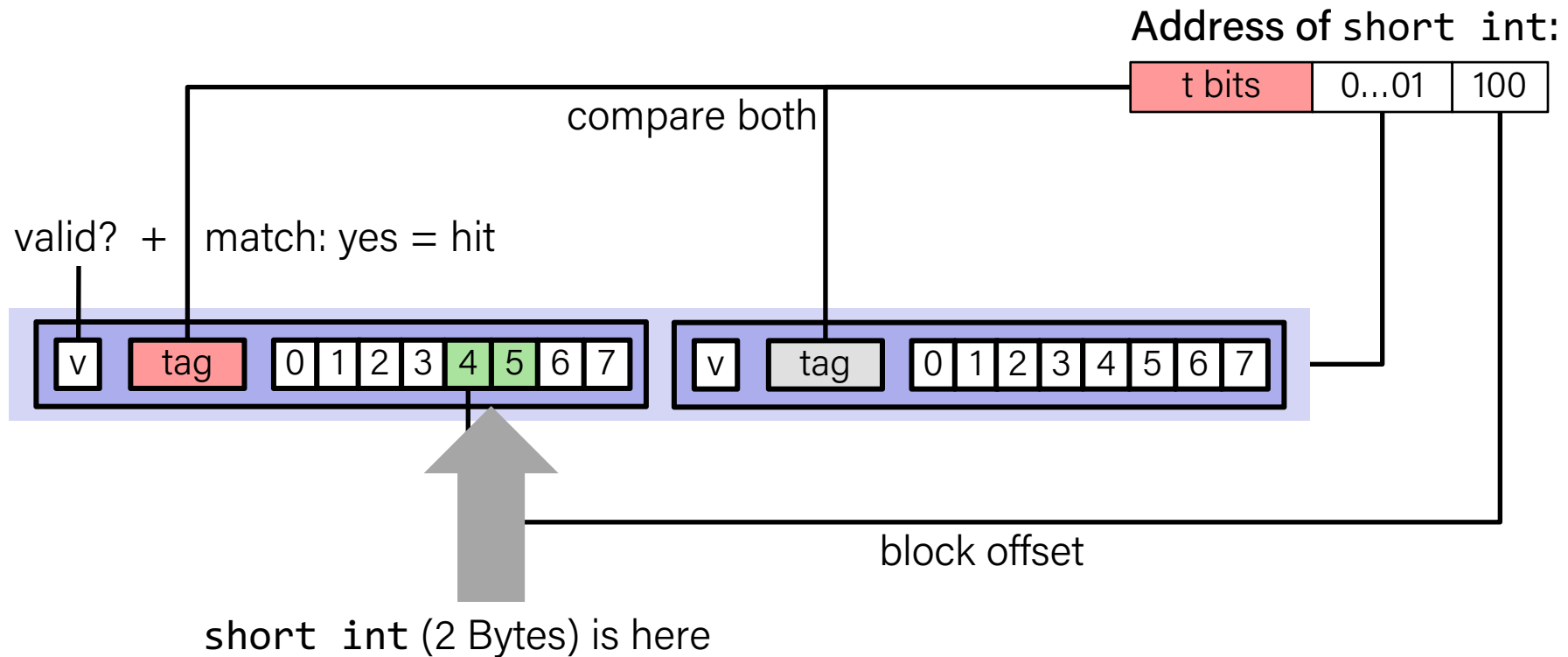
Assume: cache block size 8 bytes



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



## No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

# 2-Way Set Associative Cache Simulation

t=2 s=1 b=1

XX	X	X
----	---	---

M=16 byte addresses, B=2 bytes/block,  
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 <sub>2</sub> ],	miss
1	[00 <u>0</u> 1 <sub>2</sub> ],	hit
7	[0 <u>1</u> 1 <u>1</u> <sub>2</sub> ],	miss
8	[1 <u>0</u> 0 <u>0</u> <sub>2</sub> ],	miss
0	[00 <u>0</u> 0 <sub>2</sub> ]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

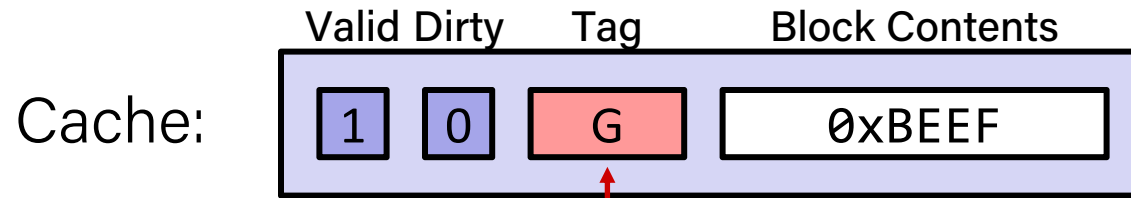


# What about writes?

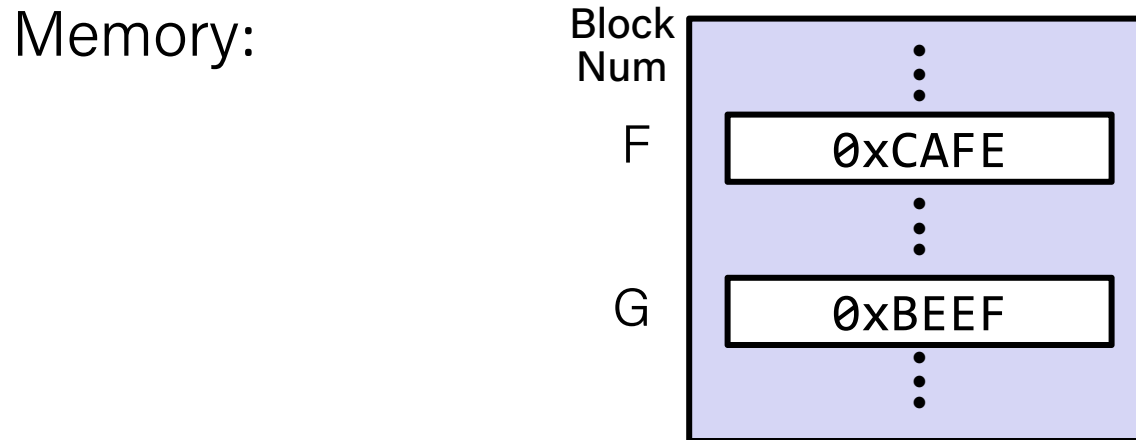
- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - **Write-through** (write immediately to memory)
  - **Write-back** (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - **Write-allocate** (load into cache, update line in cache)
    - Good if more writes to the location follow
  - **No-write-allocate** (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

# Write-back, Write Allocate Example

Note: While unrealistic, this example assumes that all requests have offset 0 and are for a block's worth of data.



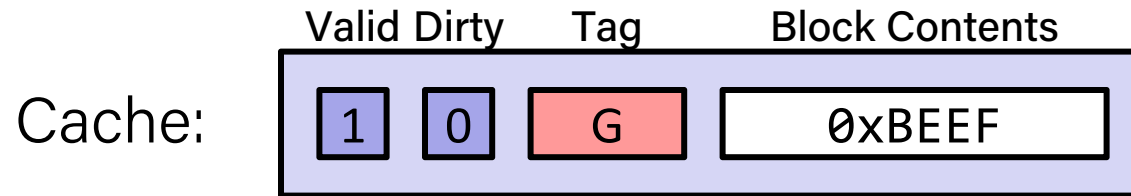
There is only one set in this tiny cache, so the tag is the entire block number!



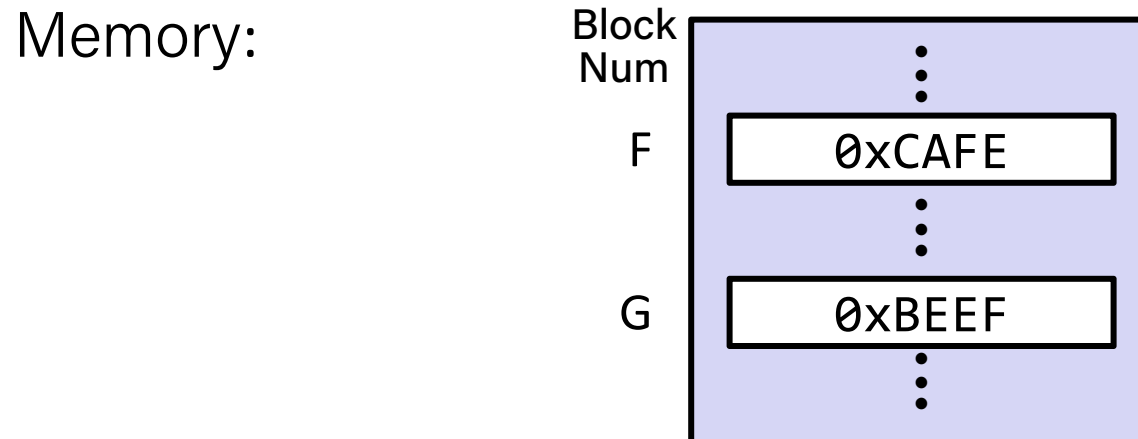
# Write-back, Write Allocate Example

1) `mov $0xFACE, (F)` Not valid x86, just using block num instead of full byte address to keep the example simple

Write Miss!



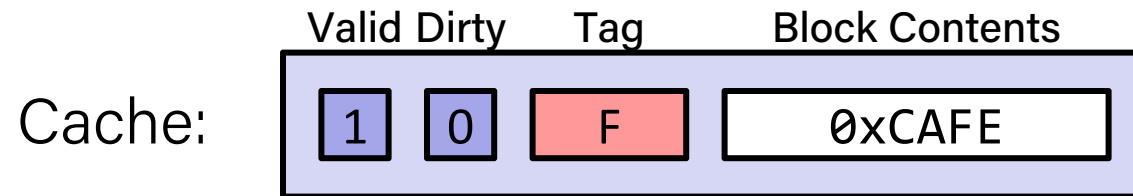
Step 1: Bring F into cache



# Write-back, Write Allocate Example

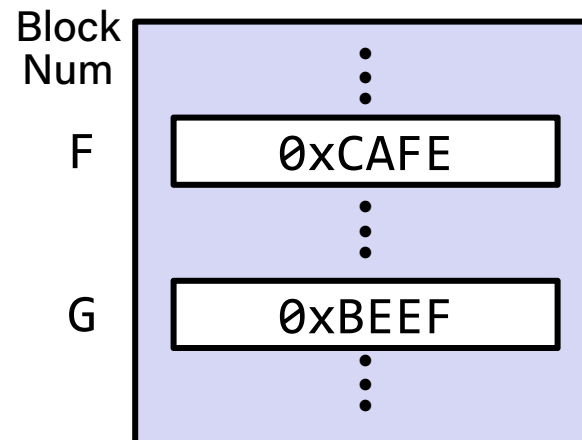
(1) `mov $0xFACE, (F)`

Write Miss



Step 1: Bring F into cache

Memory:

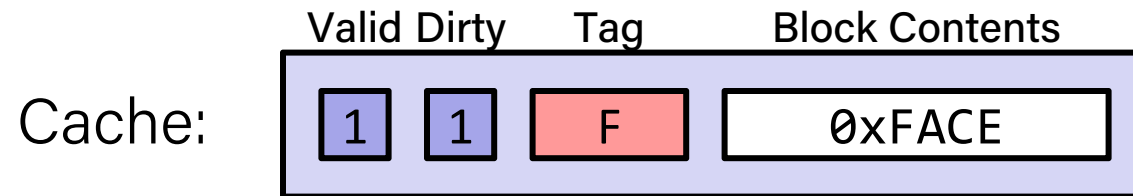


Step 2: Write 0xFACE to cache only and set the dirty bit

# Write-back, Write Allocate Example

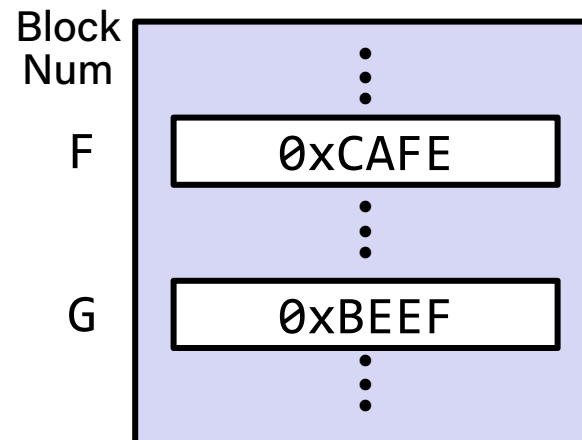
(1) `mov $0xFACE, (F)`

Write Miss



Step 1: Bring F into cache

Memory:

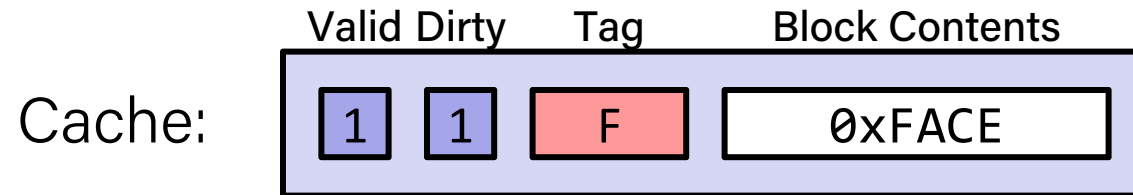


Step 2: Write 0xFACE to cache only and set the dirty bit

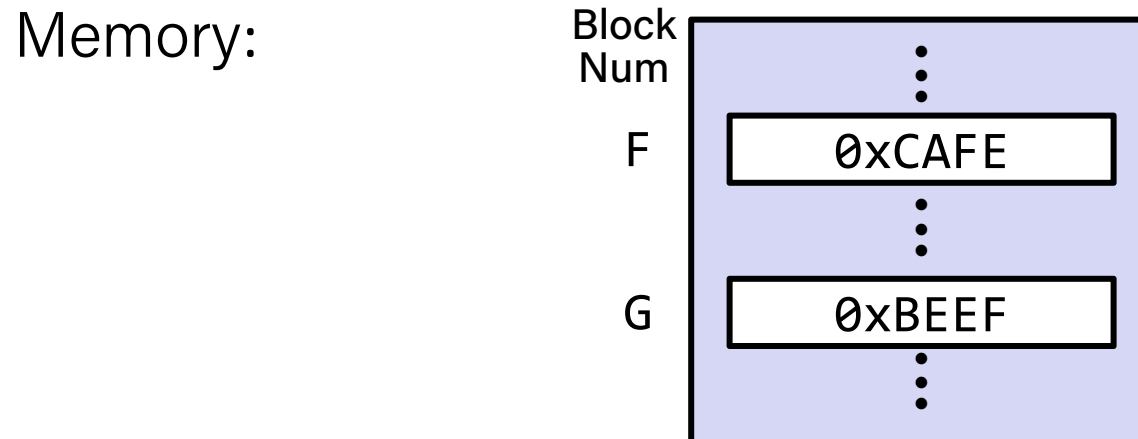
# Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`  
Write Miss

(2) `mov $0xFEED, (F)`  
Write Hit!



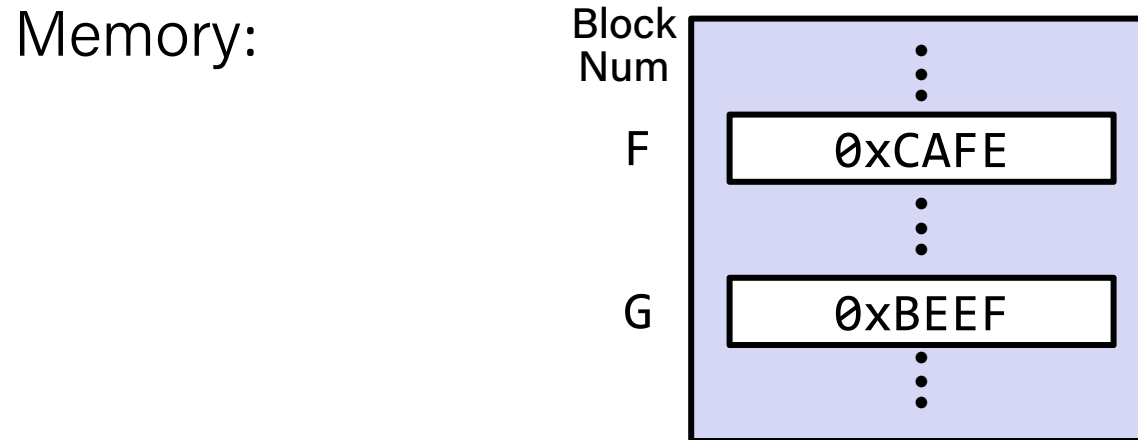
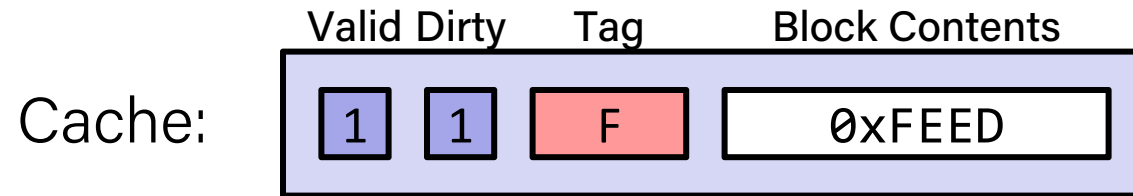
Step: Write 0xFEED to cache only (and set the dirty bit)



# Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`  
Write Miss

(2) `mov $0xFEED, (F)`  
Write Hit!

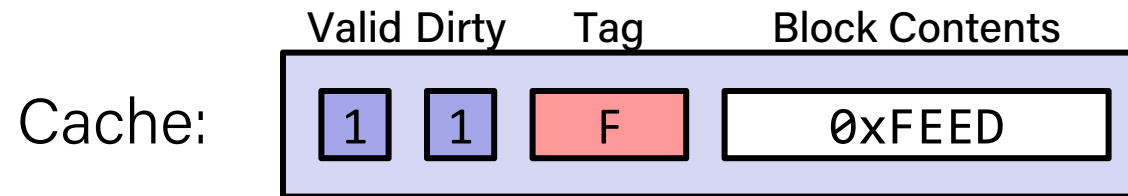


# Write-back, Write Allocate Example

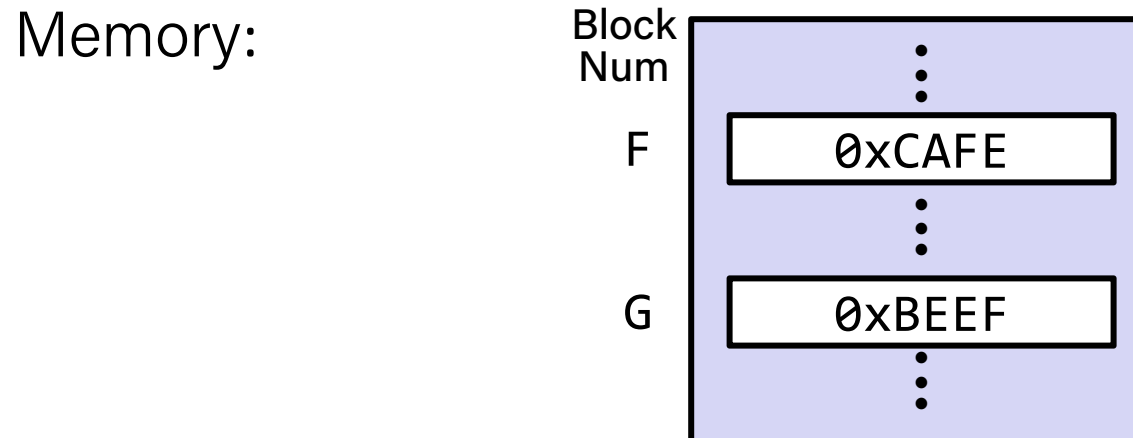
(1) `mov $0xFACE, (F)`  
Write Miss

(2) `mov $0xFEED, (F)`  
Write Hit!

(3) `mov (G), %ax`  
Read Miss!



Step 1: Write F back to memory since it is dirty



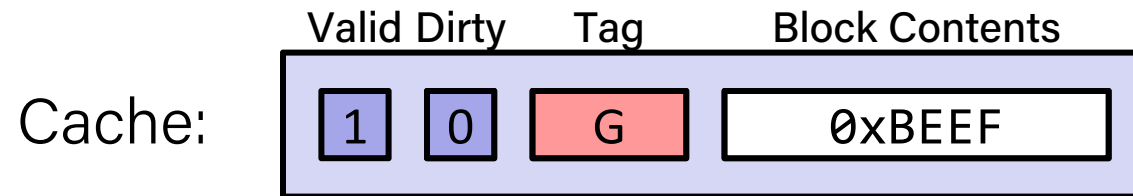


# Write-back, Write Allocate Example

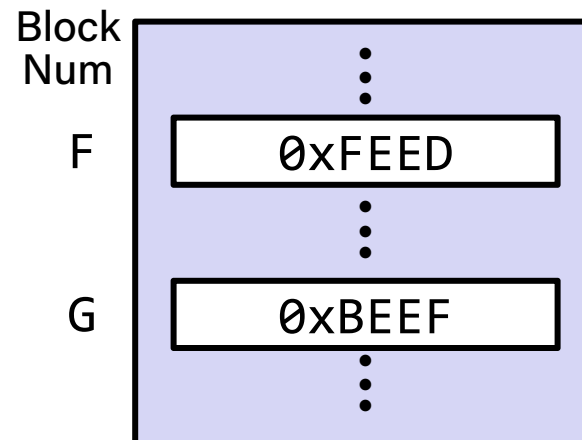
(1) `mov $0xFACE, (F)`  
Write Miss

(2) `mov $0xFEED, (F)`  
Write Hit!

(3) `mov (G), %ax`  
Read Miss!



Memory:



Step 1: Write F back to memory since it is dirty

Step 2: Bring G into the cache so that we can copy it into %ax

# Cache Simulator

<https://courses.cs.washington.edu/courses/cse351/cachesim>



# Polling Question

- Which of the following cache statements is FALSE?
  - A. We can reduce compulsory misses by decreasing our block size
  - B. We can reduce conflict misses by increasing associativity
  - C. A write-back cache will save time for code with good temporal locality on writes
  - D. A write-through cache will always match data with the memory hierarchy level below it
  - E. We're lost...

# Polling Question

- Which of the following cache statements is FALSE?

A. We can reduce compulsory misses by decreasing our block size

smaller block size pulls fewer bytes into cache on a miss

B. We can reduce conflict misses by increasing associativity

more options to place blocks before evictions occur

C. A write-back cache will save time for code with good temporal locality on writes

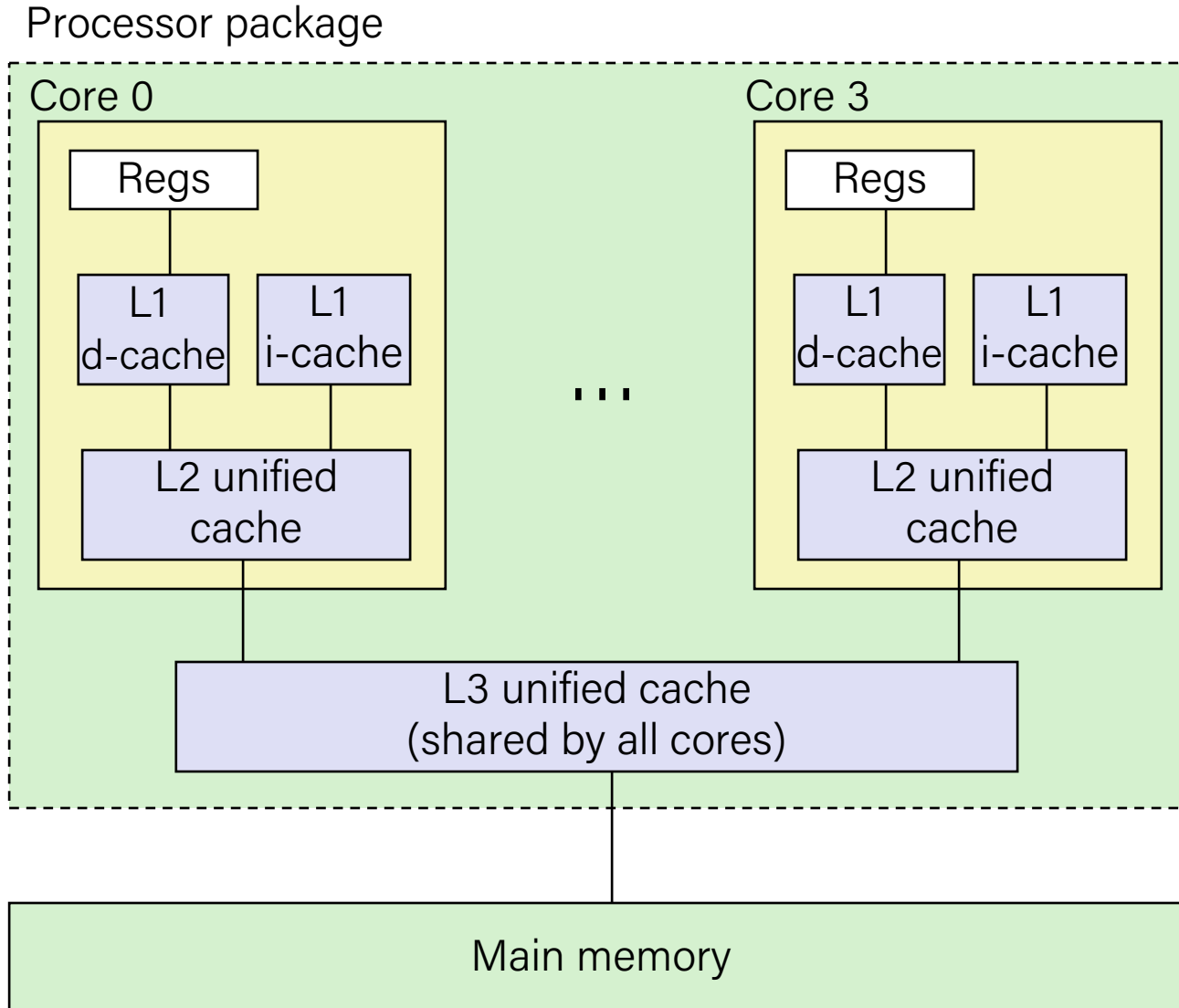
yes, its main goal is data consistency

D. A write-through cache will always match data with the memory hierarchy level below it

frequently-used blocks rarely get evicted, so fewer write-backs

E. We're lost...

# Intel Core i7 Cache Hierarchy



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

**L2 unified cache:**  
256 KB, 8-way,  
Access: 10 cycles

**L3 unified cache:**  
8 MB, 16-way,  
Access: 40-75 cycles

**Block size:** 64 bytes for all caches.

# Lecture Plan

- Cache memory organization and operation
- The memory mountain

# Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
  - Repeated references to variables are good ([temporal locality](#))
  - Stride-1 reference patterns are good ([spatial locality](#))

**Key idea:** Our qualitative notion of locality is quantified through our understanding of cache memories

# The Memory Mountain

- **Read throughput** (read bandwidth)
  - Number of bytes read from memory per second (MB/s)
- **Memory mountain:** Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.



# Memory Mountain Test Function

```
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *      array "data" with stride of "stride", using
 *      using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

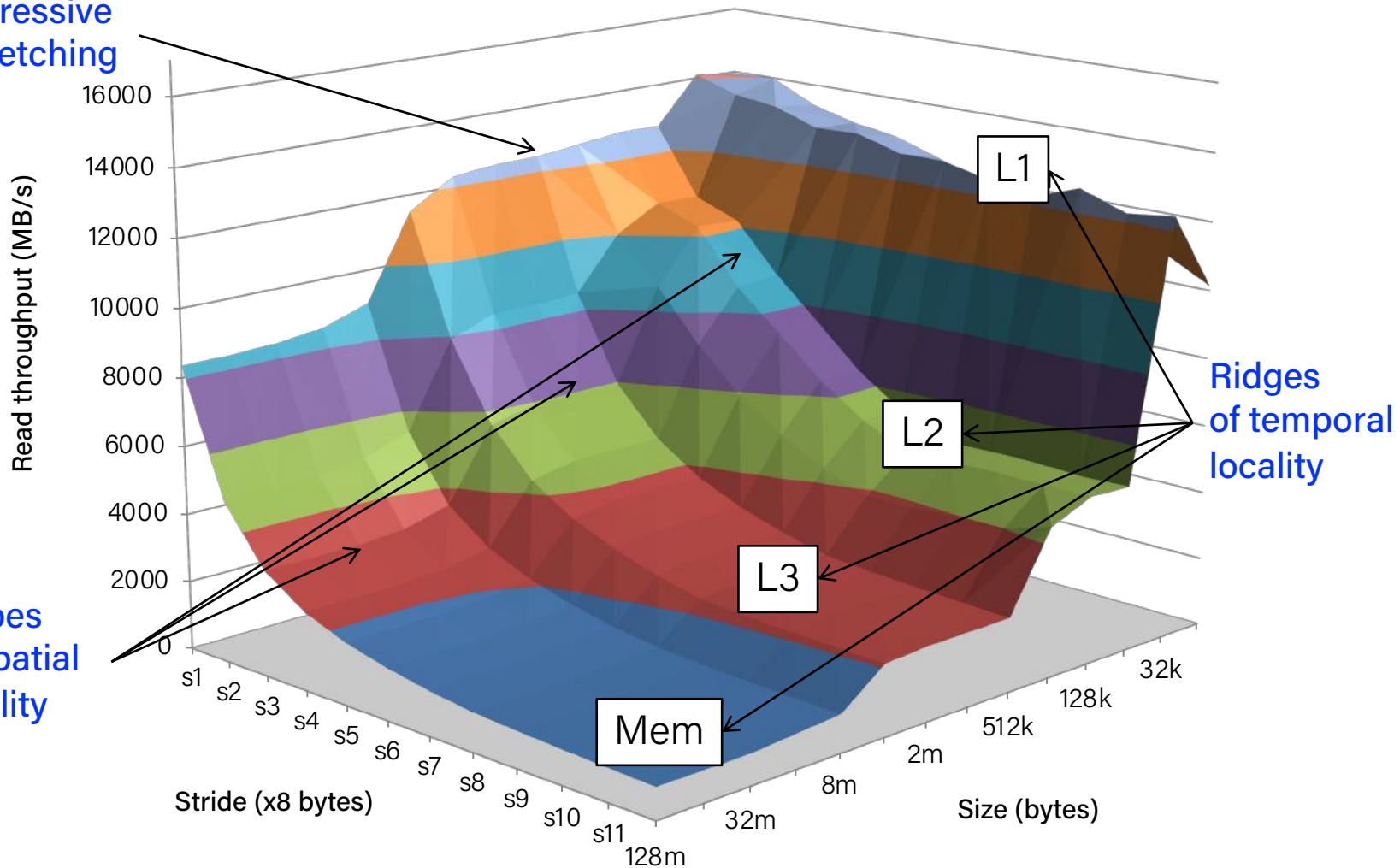
1. Call `test()` once to warm up the caches.
2. Call `test()` again and measure the read throughput (MB/s)

mountain/mountain.c

# The Memory Mountain

Aggressive prefetching

Slopes of spatial locality



Core i7 Haswell  
2.1 GHz  
32 KB L1 d-cache  
256 KB L2 cache  
8 MB L3 cache  
64 B block size

# Lecture Plan

- Cache memory organization and operation
- Memory Mountain

# Recap

- Cache memories can have significant performance impact
- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

**Next time:** *Optimization*