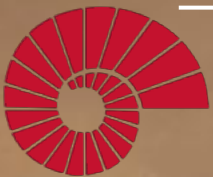


# COMP201

## Computer Systems & Programming

### Lecture #22 – Cache Memories



**KOÇ**  
**UNIVERSITY**

Aykut Erdem // Koç University // Fall 2021

# Good news, everyone!

- Your midterm exam grades will be out later today!
- There will be no labs this week!

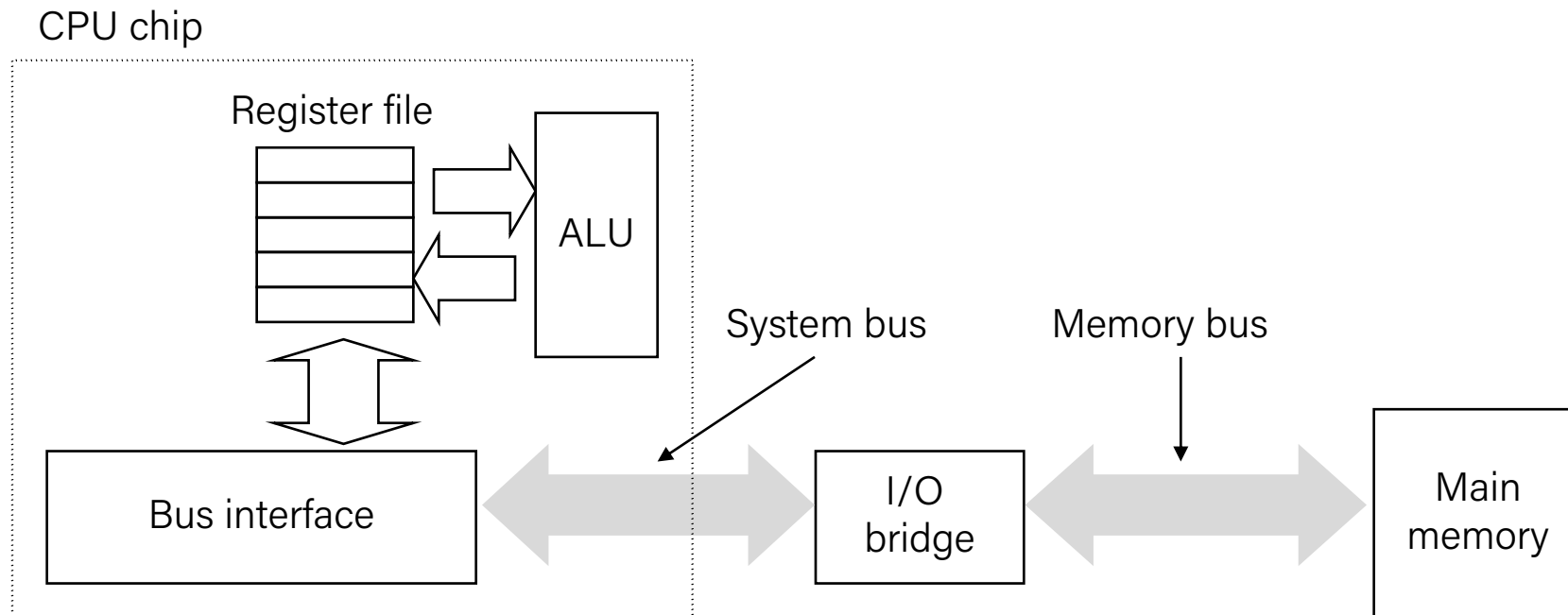
# Recap

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

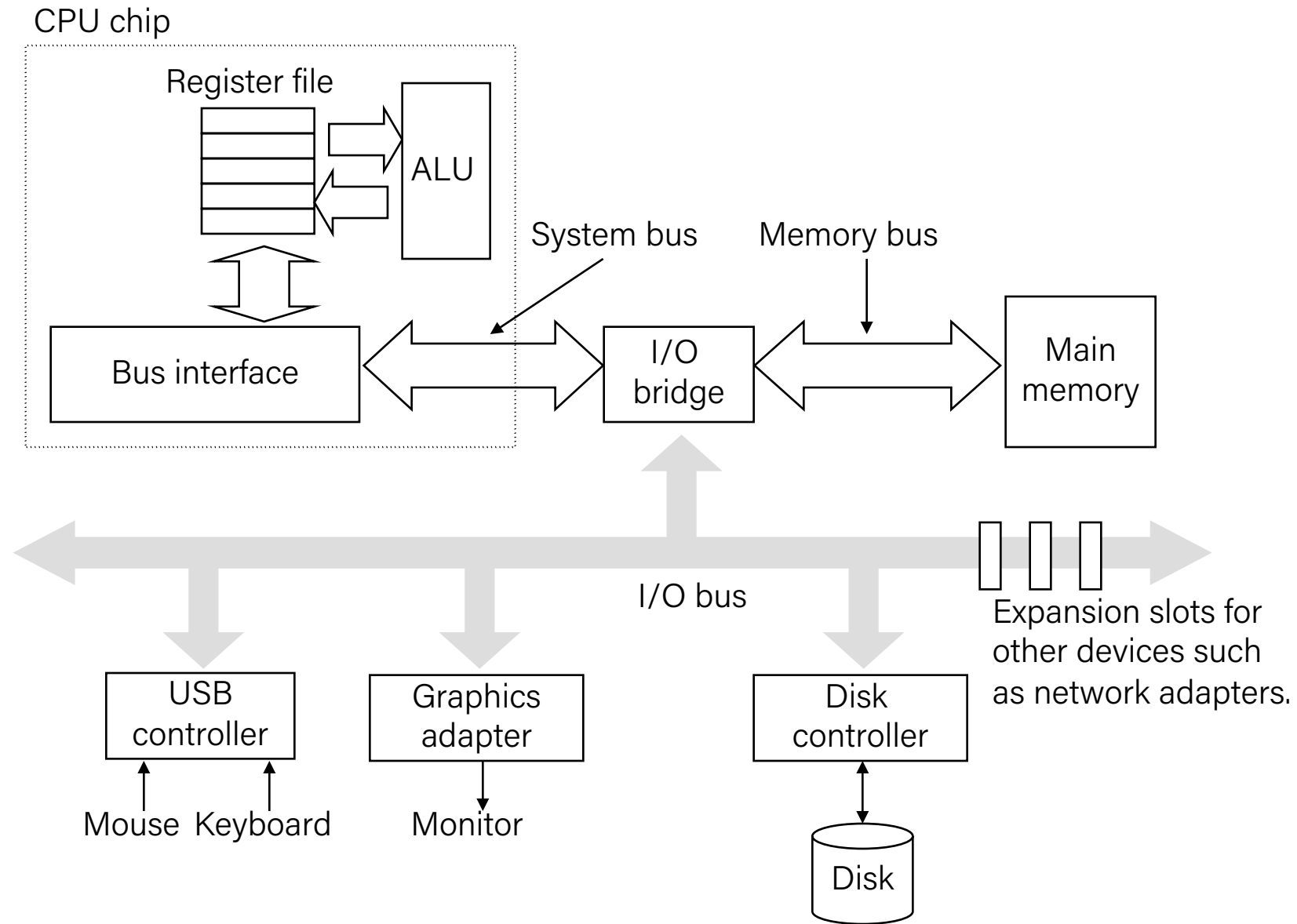


# Recap: Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



# Recap: I/O Bus

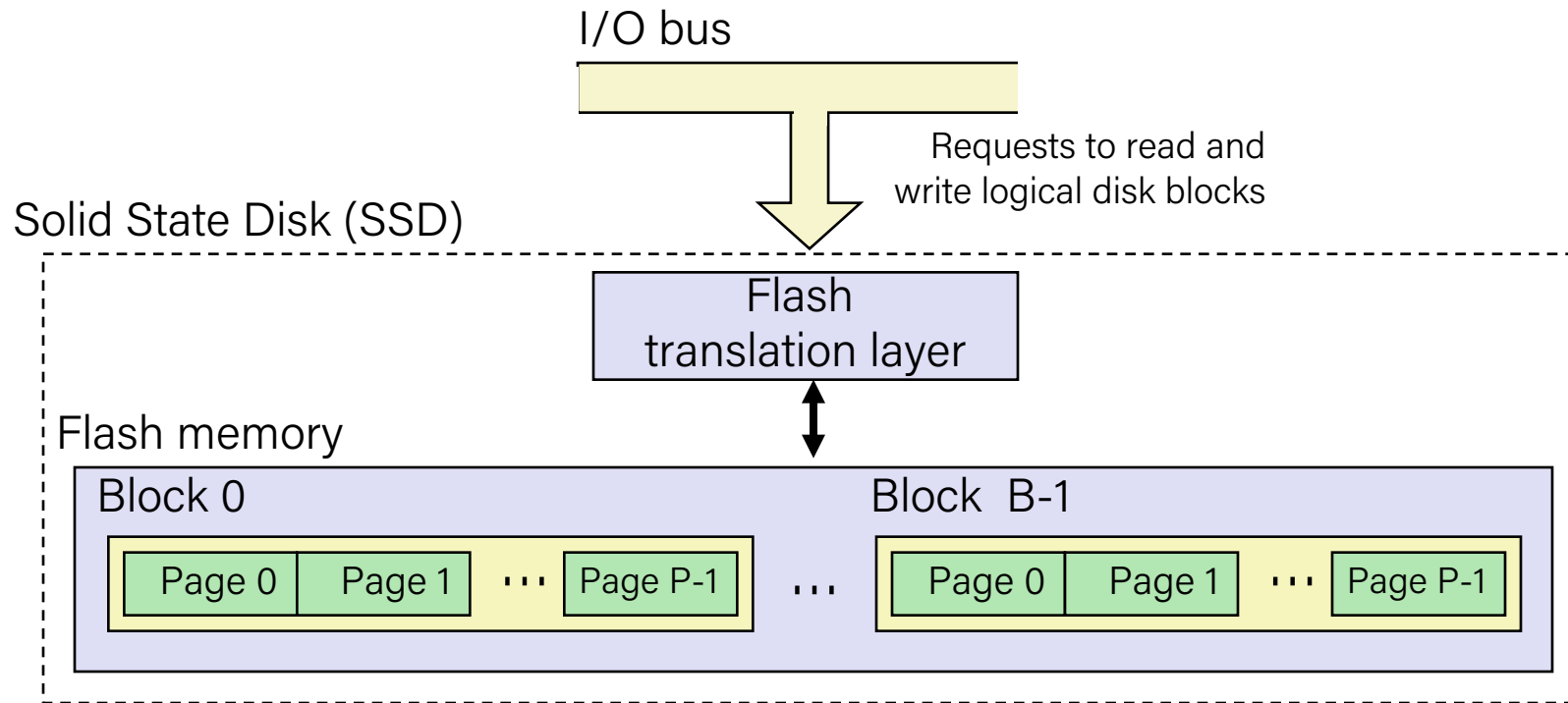


# Recap: Disk Access Time

- **Average time to access some target sector approximated by:**
  - $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$
- **Seek time ( $T_{\text{avg seek}}$ )**
  - Time to position heads over cylinder containing target sector.
  - Typical  $T_{\text{avg seek}}$  is 3—9 ms
- **Rotational latency ( $T_{\text{avg rotation}}$ )**
  - Time waiting for first bit of target sector to pass under r/w head.
  - $T_{\text{avg rotation}} = 1/2 \times 1/\text{RPMs} \times 60 \text{ sec}/1 \text{ min}$
  - Typical  $T_{\text{avg rotation}} = 7200 \text{ RPMs}$
- **Transfer time ( $T_{\text{avg transfer}}$ )**
  - Time to read the bits in the target sector.
  - $T_{\text{avg transfer}} = 1/\text{RPM} \times 1/(\text{avg \# sectors/track}) \times 60 \text{ secs}/1 \text{ min}.$

Access time is dominated by seek time and rotational latency

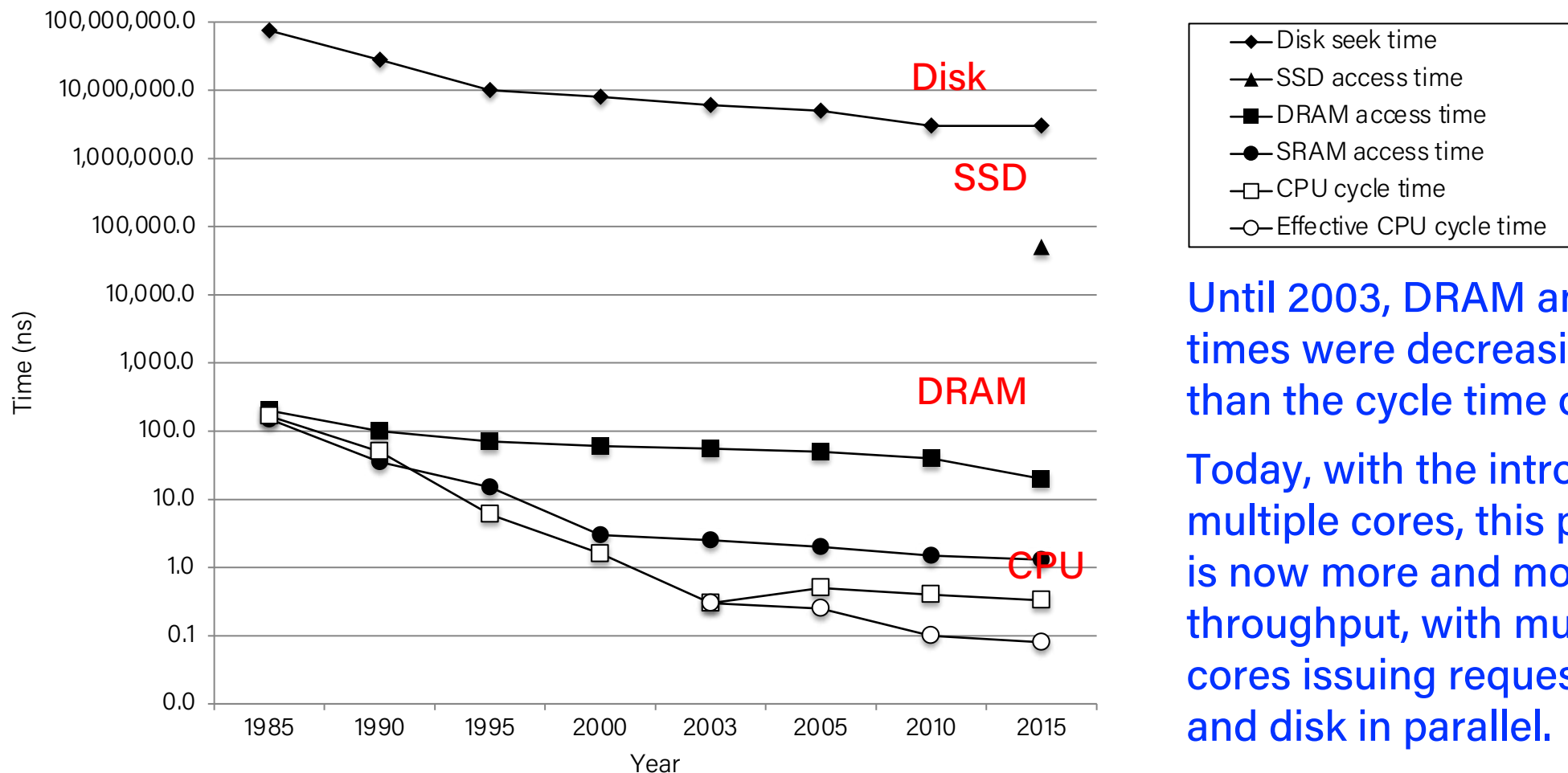
# Recap: Solid State Disks (SSDs)



- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.

# Recap: The CPU-Memory Gap

- The gap widens between DRAM, disk, and CPU speeds.



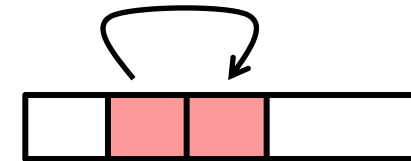
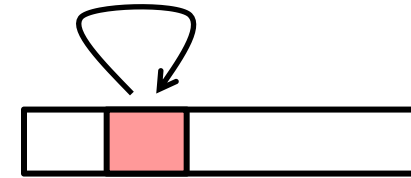
Until 2003, DRAM and disk access times were decreasing more slowly than the cycle time of a processor.

Today, with the introduction of multiple cores, this performance gap is now more and more a function of throughput, with multiple processor cores issuing requests to the DRAM and disk in parallel.



# Recap: Locality

- **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



**Well-written programs  
tend to exhibit good locality!**

# Recap: Locality Example

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

- Data references

- Reference array elements in succession (stride-1 reference pattern).
- Reference variable sum each iteration.

Spatial locality

Temporal locality

- Instruction references

- Reference instructions in sequence.
- Cycle through loop repeatedly.

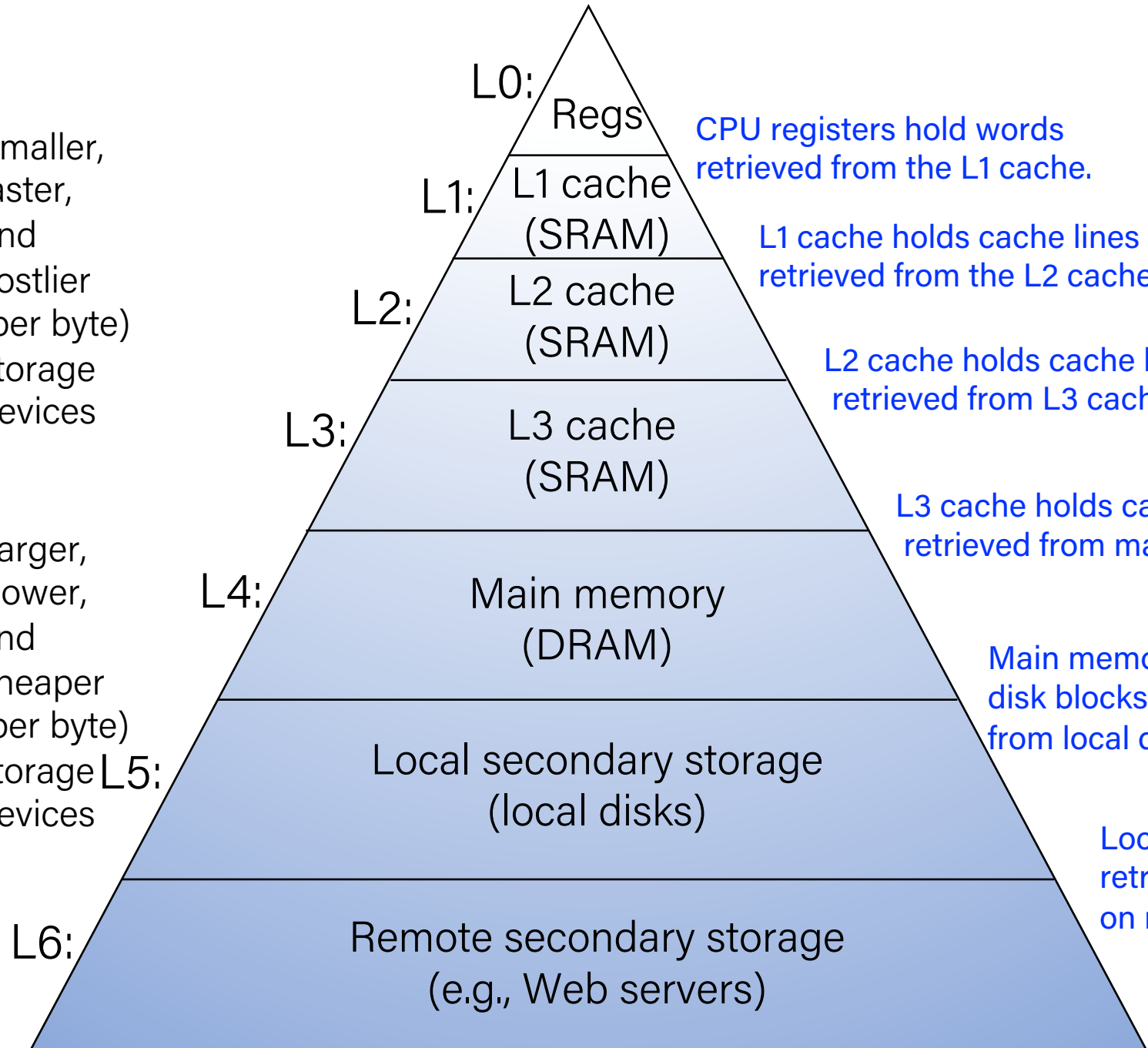
Spatial locality

Temporal locality

# Recap: Memory Hierarchy

↑  
Smaller,  
faster,  
and  
costlier  
(per byte)  
storage  
devices

↓  
Larger,  
slower,  
and  
cheaper  
(per byte)  
storage  
devices



CPU registers hold words  
retrieved from the L1 cache.

L1 cache holds cache lines  
retrieved from the L2 cache.

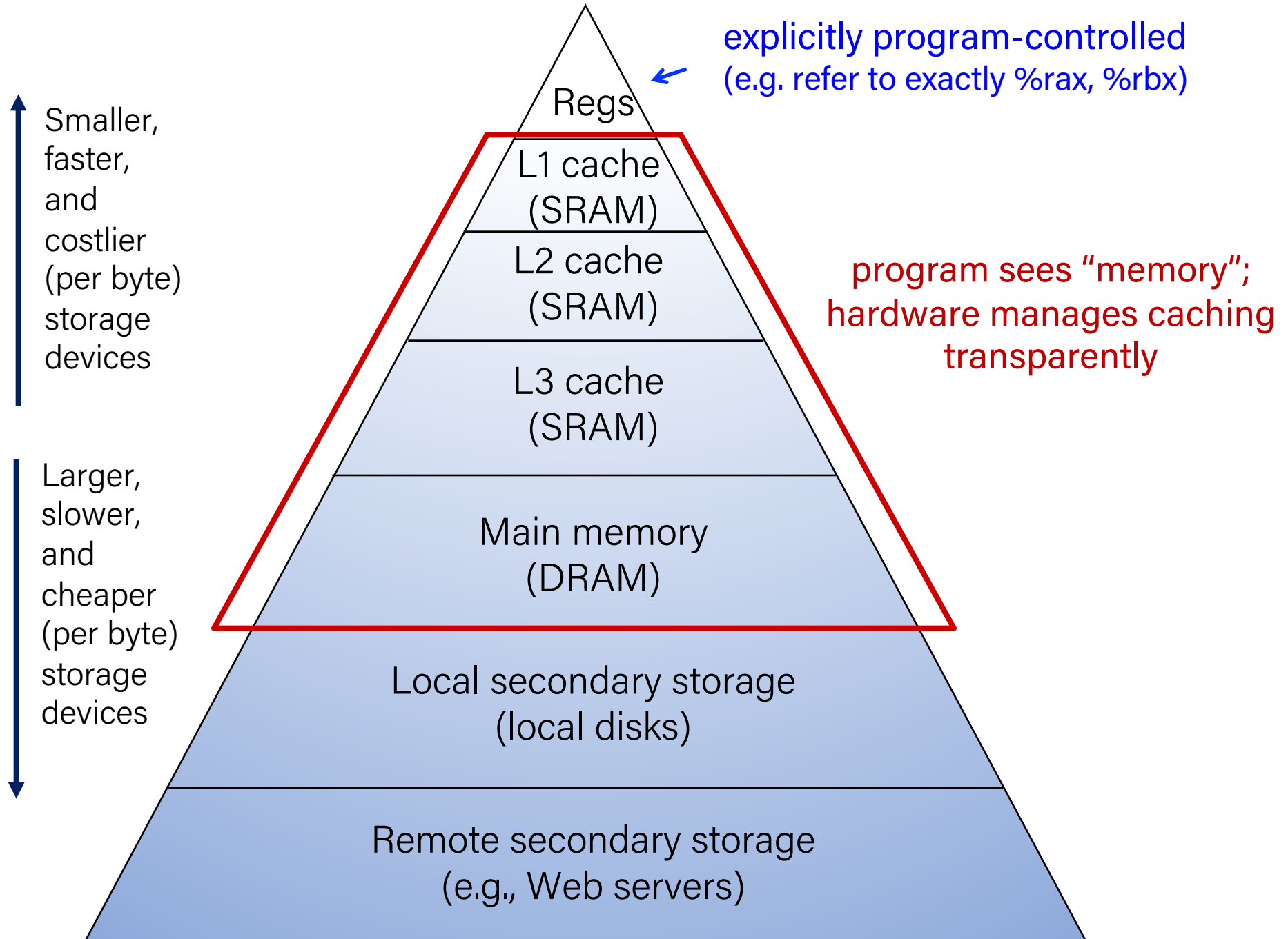
L2 cache holds cache lines  
retrieved from L3 cache

L3 cache holds cache lines  
retrieved from main memory.

Main memory holds  
disk blocks retrieved  
from local disks.

Local disks hold files  
retrieved from disks  
on remote servers

# Recap: Memory Hierarchy



# Recap: Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

# Plan for Today

- Cache basics
- Principle of locality
- Cache organization

**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



# How does execution time grow with SIZE?

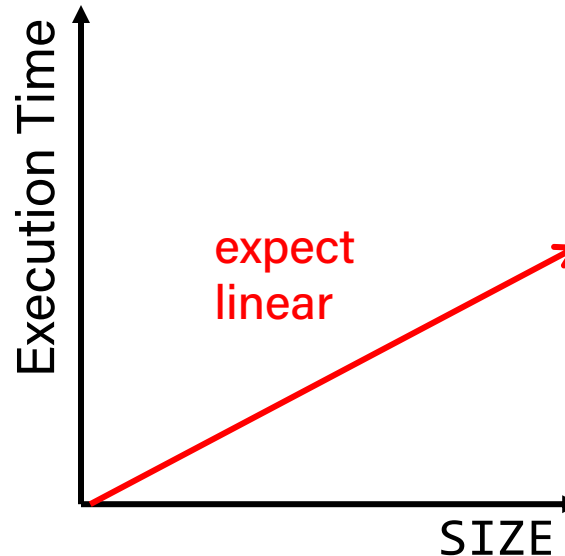
```
int array[SIZE];
int sum = 0;

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

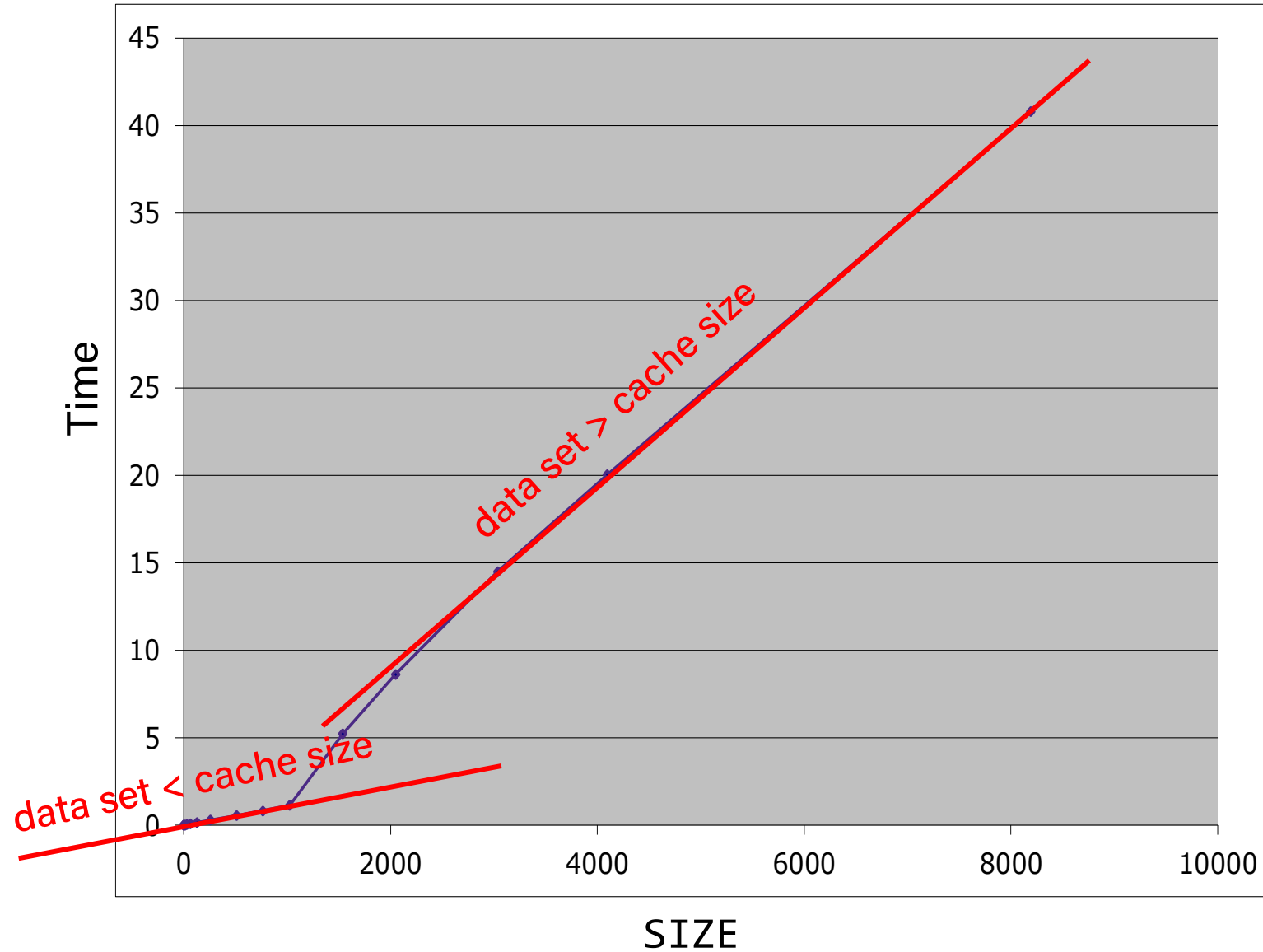
repeat 200,000 times

← execute SIZE×200,000 times

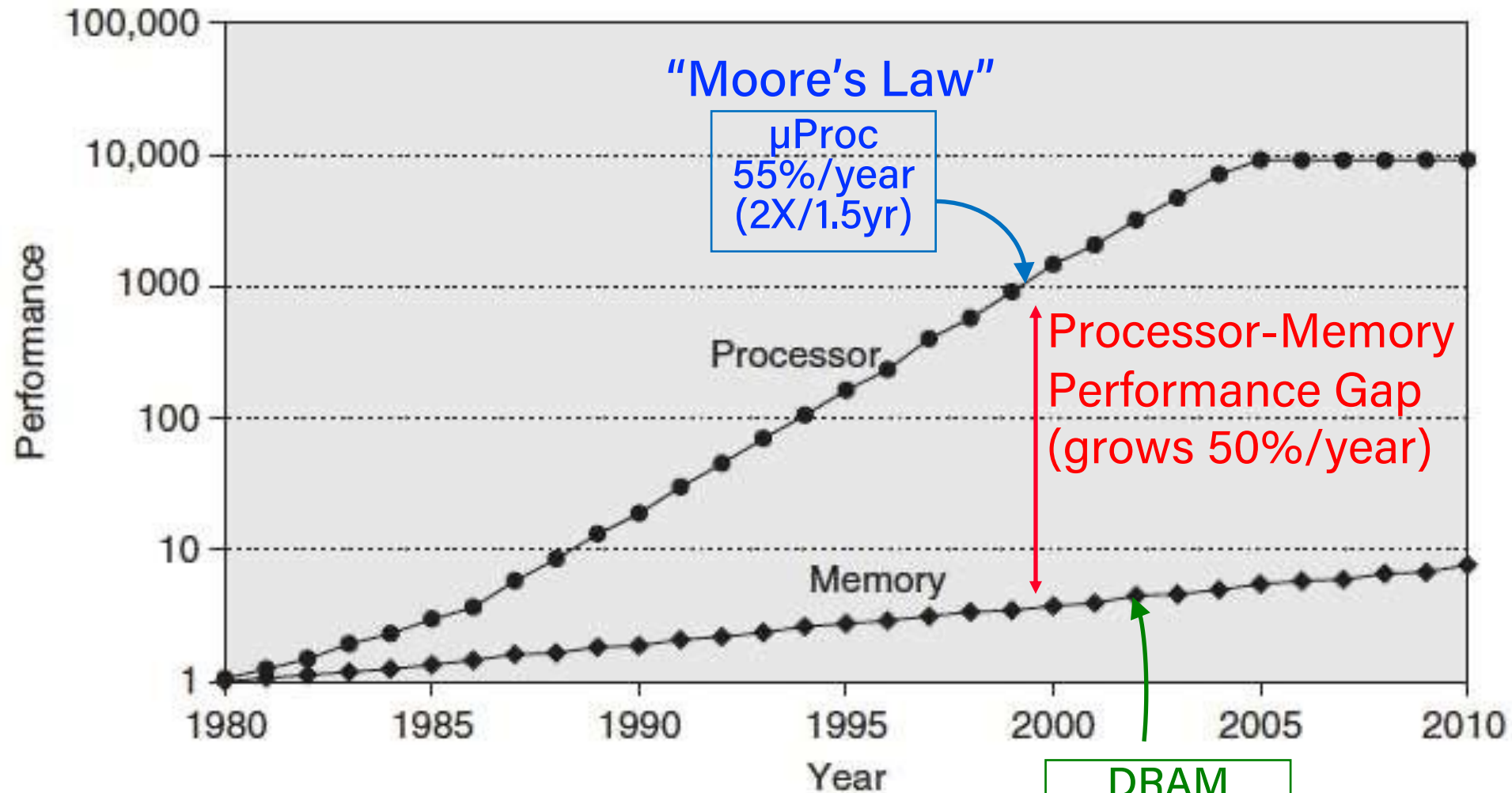
Plot:



# Actual Data



# Processor-Memory Gap

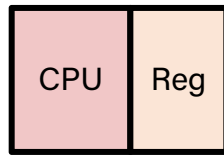


1989 first Intel CPU with cache on chip

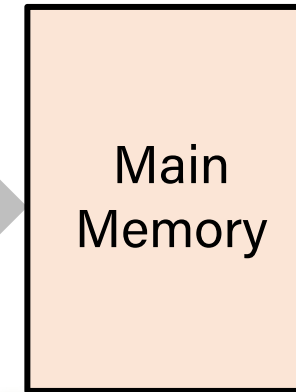
1998 Pentium III has two cache levels on chip

# Problem: Processor-Memory Bottleneck

Processor performance  
doubled about every 18  
months



Bus latency / bandwidth  
evolved much slower



**Core 2 Duo:**  
Can process at least  
256 Bytes/cycle



cycle: single machine step (fixed-time)

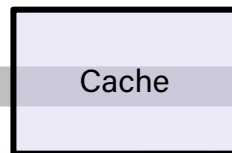
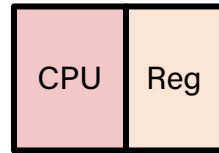
**Core 2 Duo:**  
Bandwidth  
2 Bytes/cycle  
Latency  
100-200 cycles (30-60ns)



**Problem: lots of waiting  
on memory**

# Problem: Processor-Memory Bottleneck

Processor performance  
doubled about every 18  
months

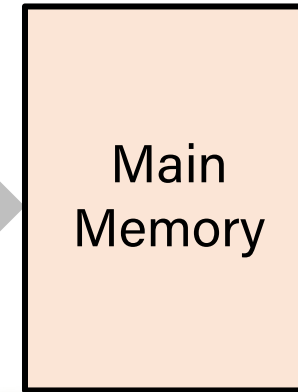


**Core 2 Duo:**  
**Can process** at least  
256 Bytes/cycle



cycle: single machine step (fixed-time)

Bus latency / bandwidth  
evolved much slower



**Core 2 Duo:**  
**Bandwidth**  
2 Bytes/cycle  
**Latency**  
100-200 cycles (30-60ns)



**Solution: caches**

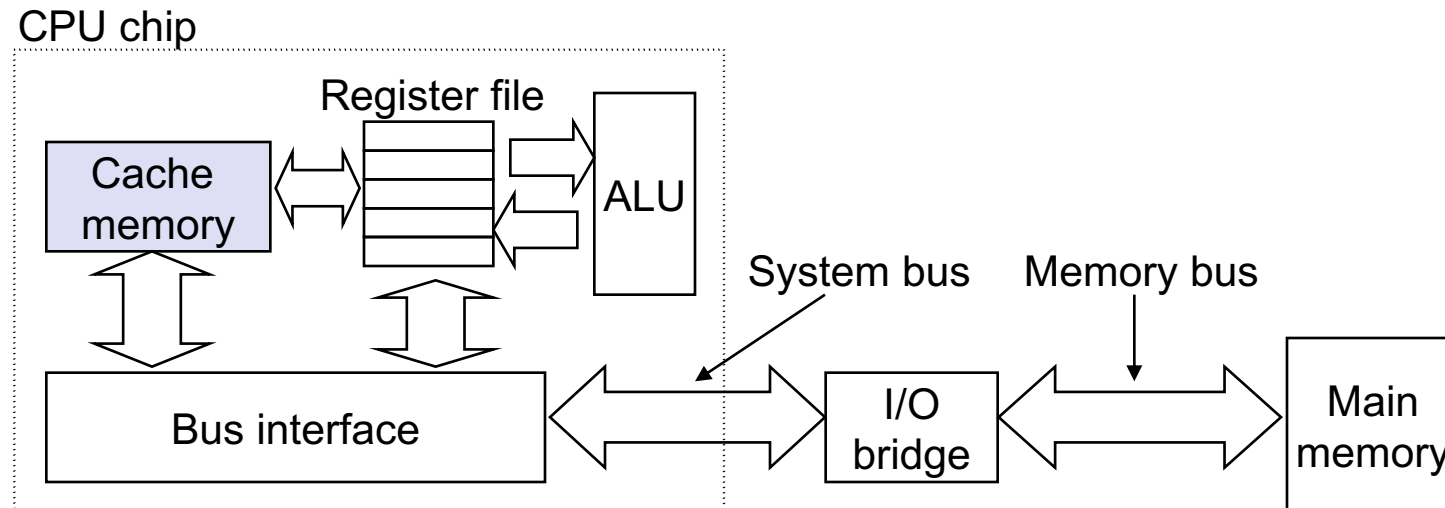
# Lecture Plan

- Cache basics
- Principle of locality
- Cache organization

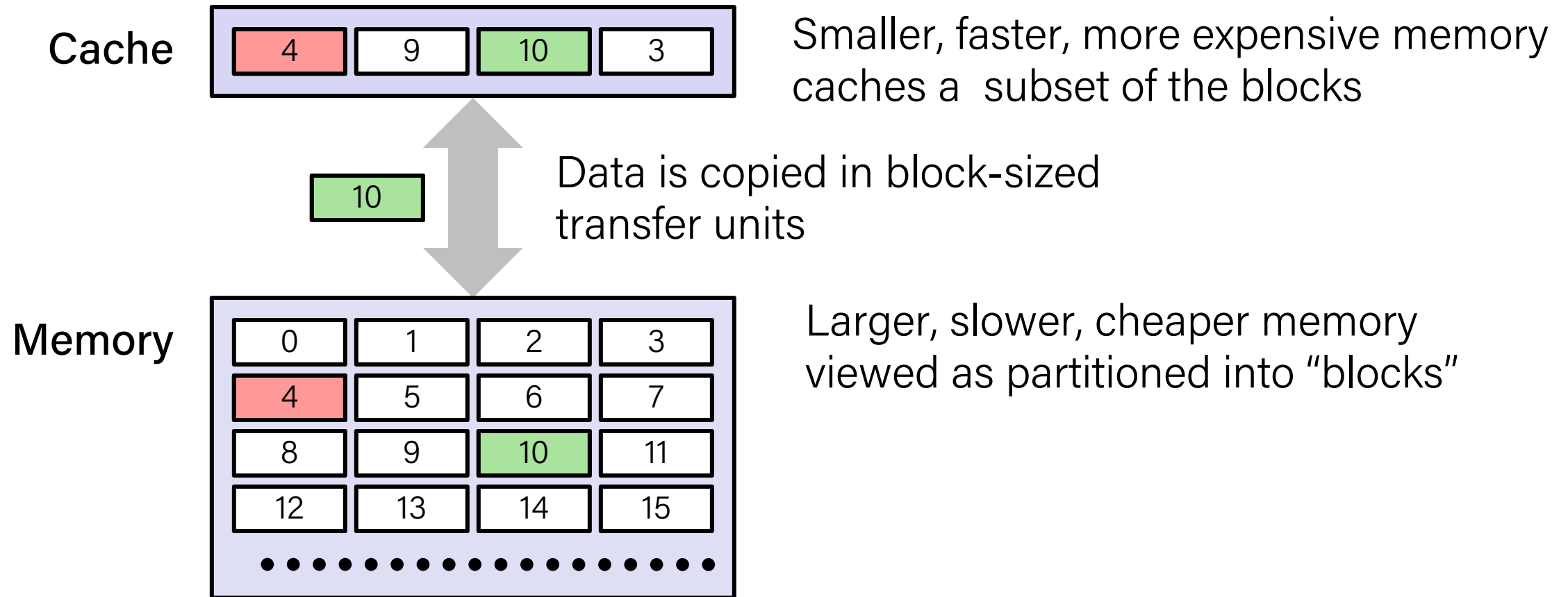


# Cache Memories

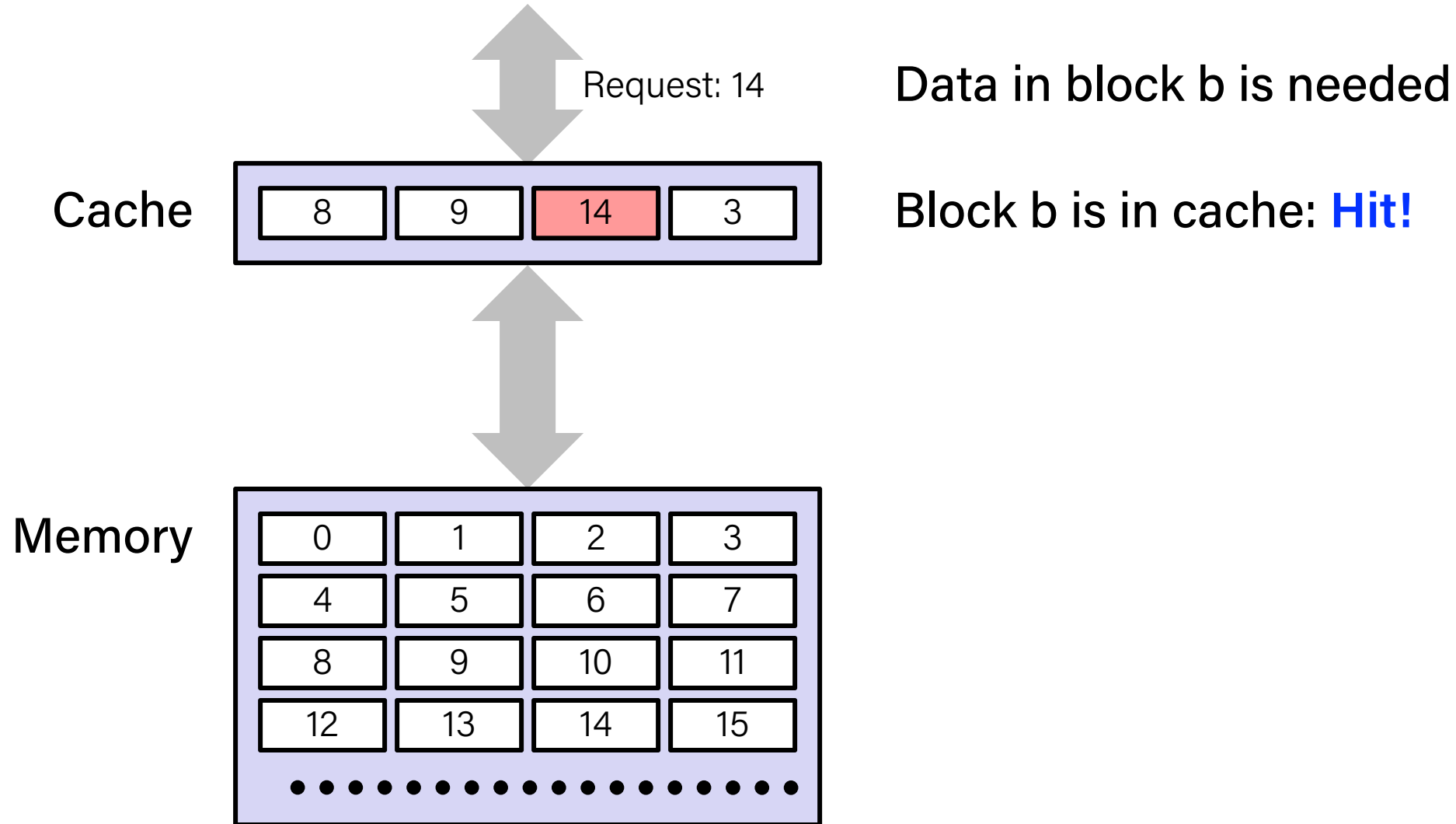
- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



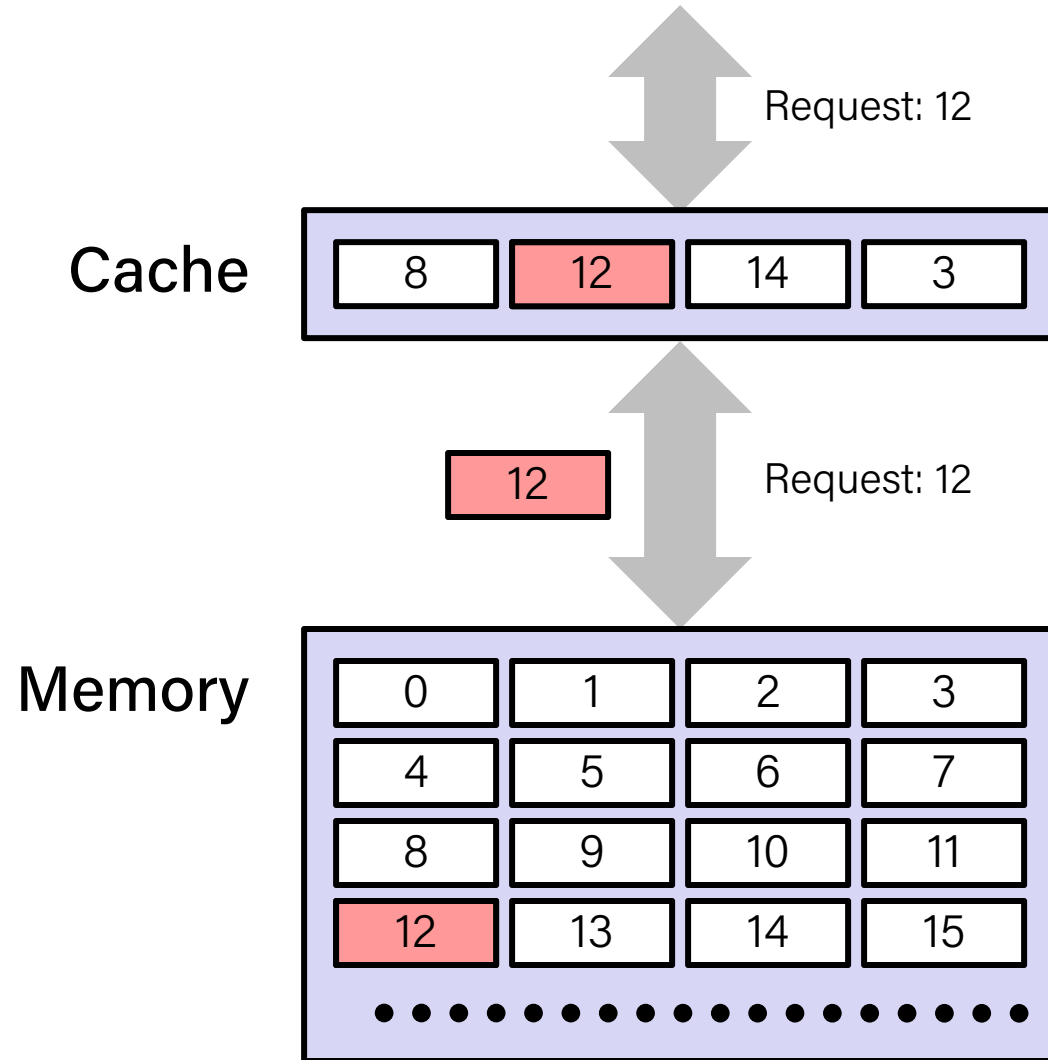
# General Cache Concepts



# General Cache Concepts: Hit



# General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache: **Miss!**

Block b is fetched from memory

Block b is stored in cache

- **Placement policy:** determines where b goes
- **Replacement policy:** determines which block gets evicted (victim)

# Types of Cache Misses

- **Cold (compulsory) miss**

- Cold misses occur because the cache is empty.

- **Conflict miss**

- Most caches limit blocks at level  $k+1$  to a small subset (sometimes a singleton) of the block positions at level  $k$ .
  - E.g. Block  $i$  at level  $k+1$  must be placed in block  $(i \bmod 4)$  at level  $k$ .
- Conflict misses occur when the level  $k$  cache is large enough, but multiple data objects all map to the same level  $k$  block.
  - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

- **Capacity miss**

- Occurs when the set of active cache blocks (working set) is larger than the cache.

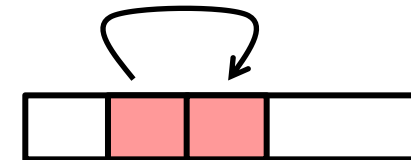
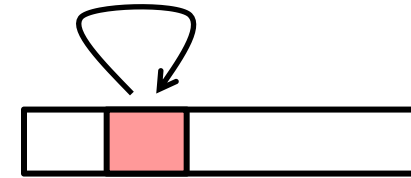
# Lecture Plan

- Cache basics
- Principle of locality
- Cache organization



# 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



# Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.
- **Question:** 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;
}
```

# Locality Example 1

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

# Locality Example 1

- 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;
}
```

M = 3,  
N = 4

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]

Access Pattern:  
stride = 1

1)	a[0][0]
2)	a[0][1]
3)	a[0][2]
4)	a[0][3]
5)	a[1][0]
6)	a[1][1]
7)	a[1][2]
8)	a[1][3]
9)	a[2][0]
10)	a[2][1]
11)	a[2][2]
12)	a[2][3]

Layout in Memory

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

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

↑ 76                      ↑ 92                      ↑ 108

# Locality Example 2

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

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

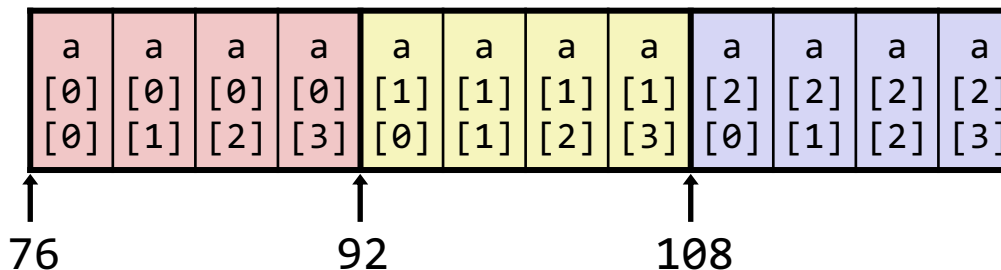
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

M = 3,  
N = 4

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]

Layout in Memory

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



# Locality Example 2

- 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;
}
```

M = 3,  
N = 4

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]

Access Pattern:  
stride = 4

1)	a[0][0]
2)	a[1][0]
3)	a[2][0]
4)	a[0][1]
5)	a[1][1]
6)	a[2][1]
7)	a[0][2]
8)	a[1][2]
9)	a[2][2]
10)	a[0][3]
11)	a[1][3]
12)	a[2][3]

Layout in Memory

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

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

↑ 76                      ↑ 92                      ↑ 108



# Locality Example 3

```
int sum_array_3d(int a[M][N][L])
{
    int i, j, k, sum = 0;

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

    return sum;
}
```

- What is wrong with this code?

**Access Pattern: stride- $N \times L$**

- How can it be fixed?

**Inner loop:  $i \rightarrow \text{stride-1}$**

**$j \rightarrow \text{stride-1}$**

**$k \rightarrow \text{stride-}N \times L$**

Layout in Memory ( $M = 2$ ,  $N = 3$ ,  $L = 4$ )

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

76                      92                      108                      124                      140                      156                      172

$m=2$

$m=1$

$m=0$

# Cache Performance Metrics

- Huge difference between a cache hit and a cache miss
  - Could be 100x speed difference between accessing cache and main memory (measured in *clock cycles*)
- Miss Rate (MR)
  - Fraction of memory references not found in cache (misses / accesses)  
= 1 - Hit Rate
- Hit Time (HT)
  - Time to deliver a block in the cache to the processor
    - Includes time to determine whether the block is in the cache
- Miss Penalty (MP)
  - Additional time required because of a miss

# Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Would you believe 99% hits is twice as good as 97%?
  - Consider:  
cache hit time of 1 cycle  
miss penalty of 100 cycles
  - Average access time:
    - 97% hits:  $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$
    - 99% hits:  $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$
- **This is why “miss rate” is used instead of “hit rate”**

# Can we have more than one cache?

- Why would we want to do that?

- Avoid going to memory!

- Typical performance numbers:

- Miss Rate

- L1 MR = 3-10%

- L2 MR = Quite small (e.g. < 1%), depending on parameters, etc.

- Hit Time

- L1 HT = 4 clock cycles

- L2 HT = 10 clock cycles

- Miss Penalty

- P = 50-200 cycles for missing in L2 & going to main memory

- Trend: increasing!

**(1)Optimize L1 for high HT**

**(2)Optimize L2 for low MR**

# Summary

- Memory Hierarchy
  - Successively higher levels contain “most used” data from lower levels
  - Exploits *temporal and spatial locality*
  - Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics
- Cache Performance
  - Ideal case: found in cache (hit)
  - Bad case: not found in cache (miss), search in next level
  - Average Memory Access Time (AMAT) =  $HT + MR \times MP$ 
    - Hurt by Miss Rate and Miss Penalty

# Lecture Plan

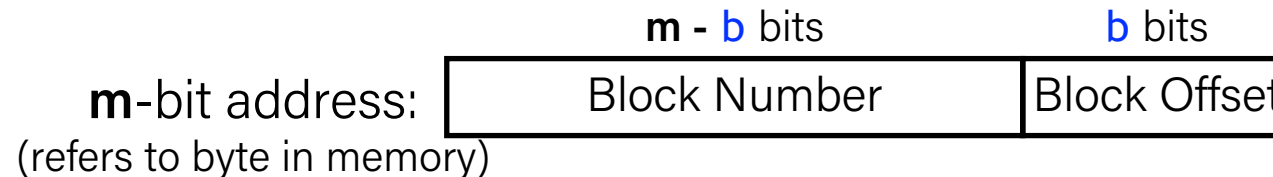
- Cache basics
- Principle of locality
- Cache organization

# 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!

# 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
  - $(\text{address}) \bmod (\# \text{ of bytes in a block})$





# Question

- If we have 6-bit addresses and block size  $B = 4$  bytes, which block and byte does `0x15` refer to?

	Block Num	Block Offset
A.	1	1
B.	1	5
C.	5	1
D.	5	5
E.	We're lost...	

# Question

- If we have 6-bit addresses and block size  $B = 4$  bytes, which block and byte does **0x15** refer to?

	Block Num	Block Offset
A.	1	1
B.	1	5
<b>C.</b>	<b>5</b>	<b>1</b>
D.	5	5
E.	We're lost...	

0x <sup>1</sup> <sup>5</sup>  
Address: 0b 0 1 0 1 / 0 1

Offset width =  $\log_2(B) = \log_2(4) = 2$  bits

# Cache Organization

- **Cache Size (C):** amount of *data* the cache can store
  - Cache can only hold so much data (subset of next level)
  - Given in bytes (C) or number of blocks (C/B)
  - Example:  $C = 32 \text{ KiB} = 512 \text{ blocks}$  if using 64-byte blocks
- Where should data go in the cache?
  - We need a mapping from memory addresses to specific locations in the cache to make checking the cache for an address **fast**
- What is a data structure that provides fast lookup?
  - Hash table!

# Review: Hash Tables for Fast Lookup

Insert:

5

27

34

102

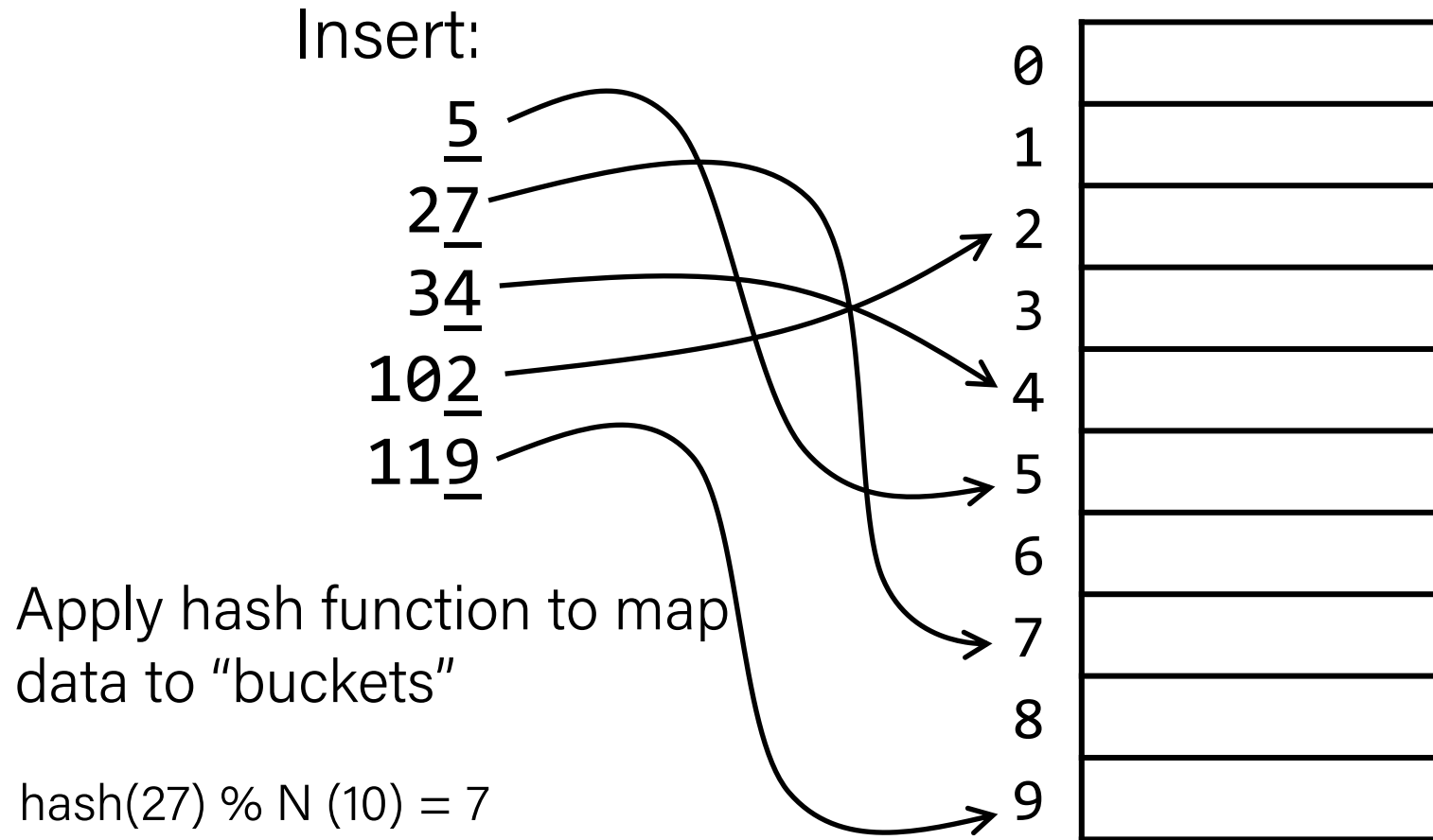
119

Apply hash function to map  
data to "buckets"

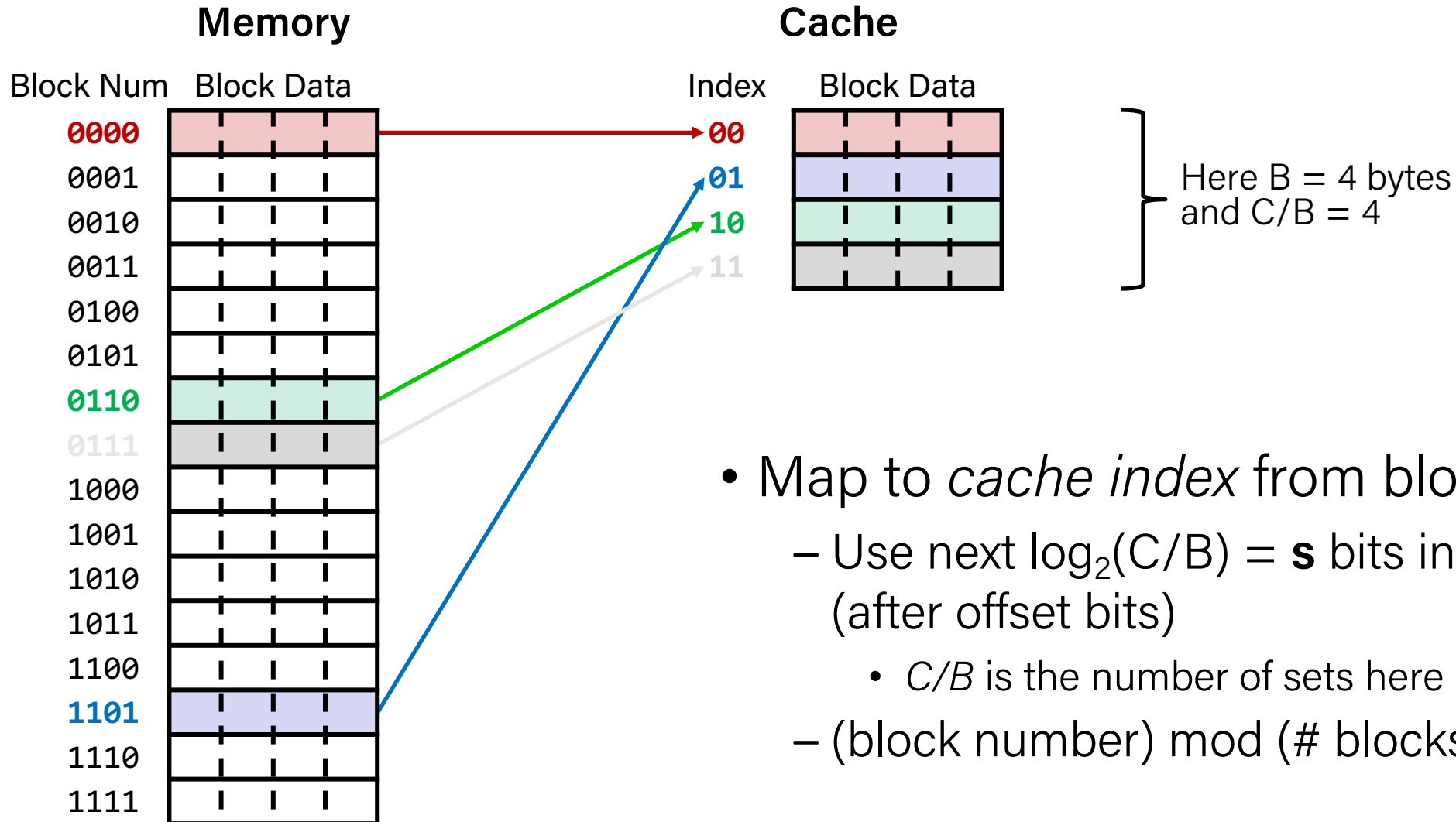
$$\text{hash}(27) \% N(10) = 7$$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

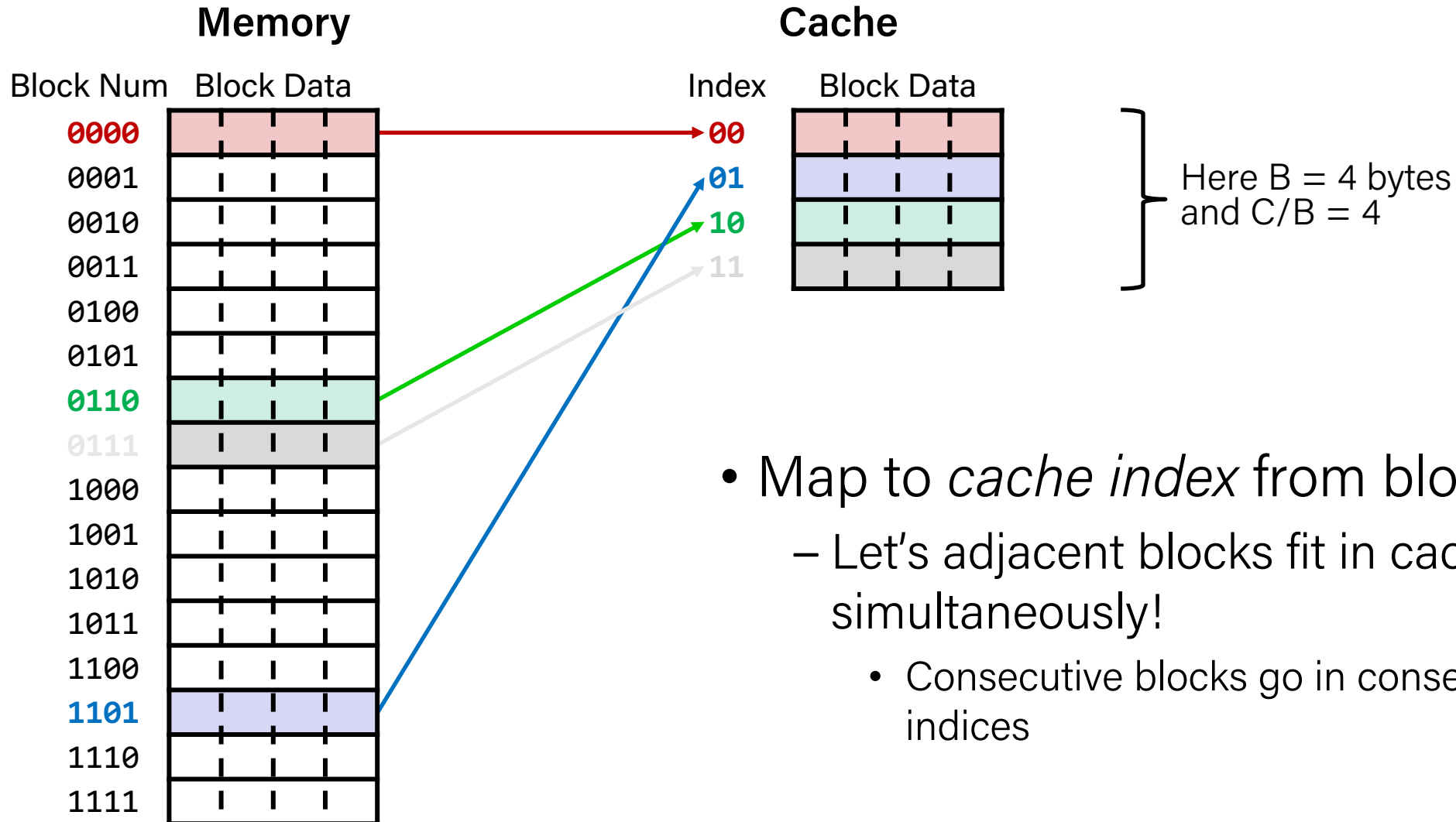
# Review: Hash Tables for Fast Lookup



# Place Data in Cache by Hashing Address



# Place Data in Cache by Hashing Address



# Practice Question

- 6-bit addresses, block size  $B = 4$  bytes, and our cache holds  $S = 4$  blocks.
- A request for address **0x2A** results in a cache miss. Which set index does this block get loaded into and which 3 other addresses are loaded along with it?



# Practice Question

- 6-bit addresses, block size  $B = 4$  bytes, and our cache holds  $S = 4$  blocks.  
 $C = S \times B = 16$  bytes                       $b = \log_2(4) = 2$  bits                       $s = \log_2(4) = 2$  bits
- A request for address **0x2A** results in a cache miss. Which set index does this block get loaded into and which 3 other addresses are loaded along with it?

Address: 0x 2 A

0b 1 0 | 1 0 | 1 0

index offset

block number

addresses w/block number 1010

0b101000 = 0x28

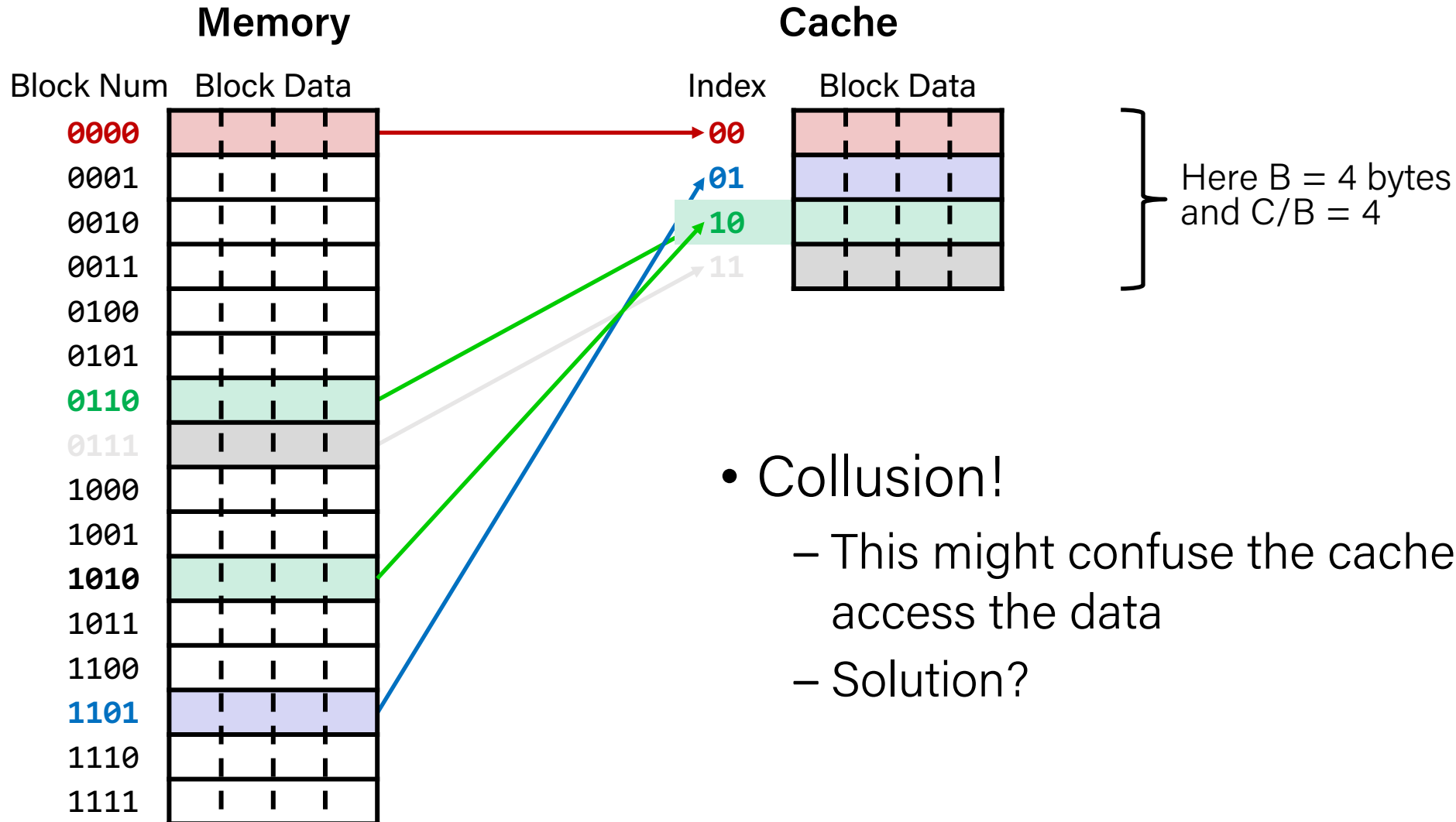
0b101001 = 0x29

0b101010 = 0x2A

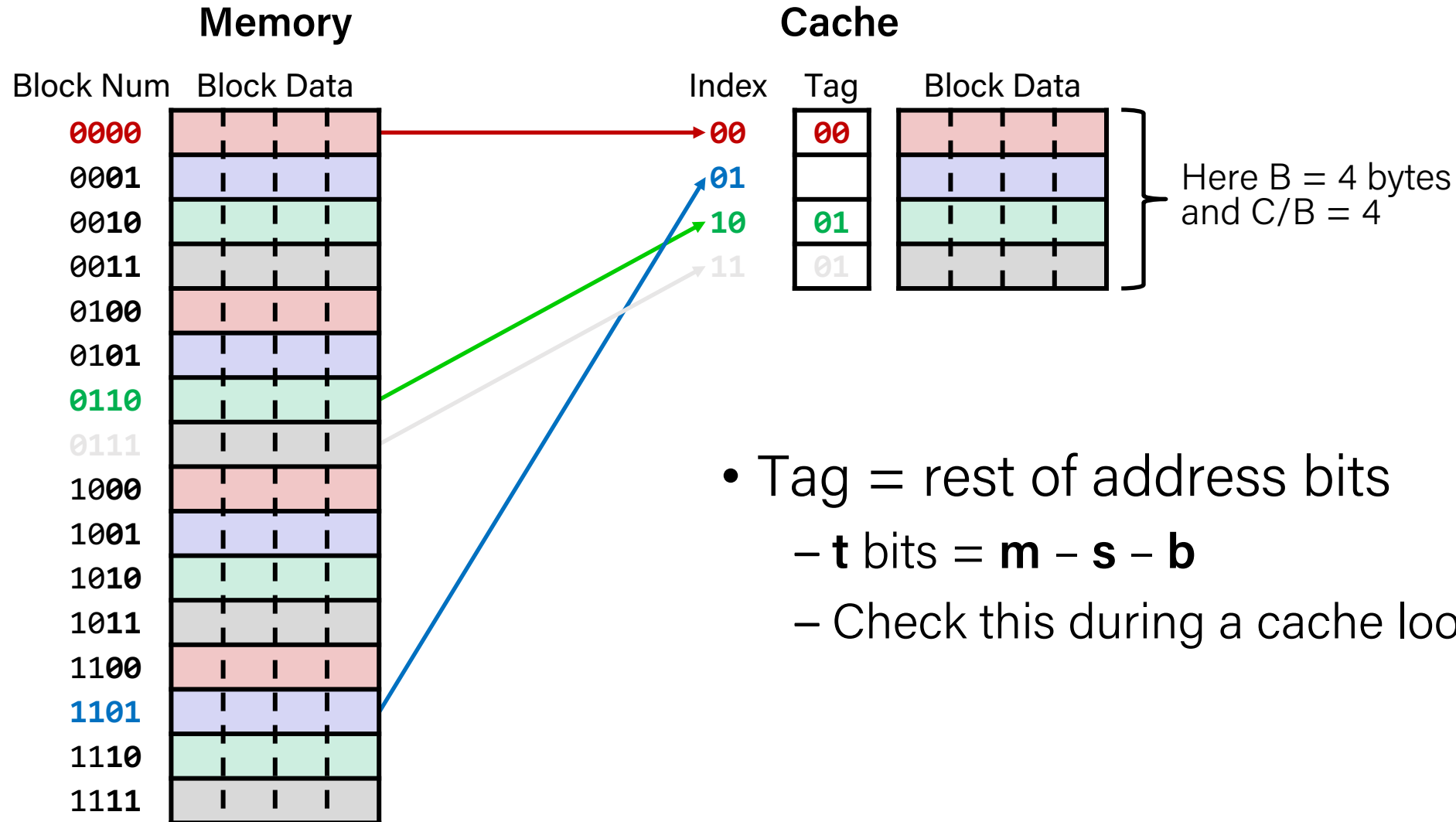
0b101011 = 0x2B

**These are loaded into cache!**

# Place Data in Cache by Hashing Address



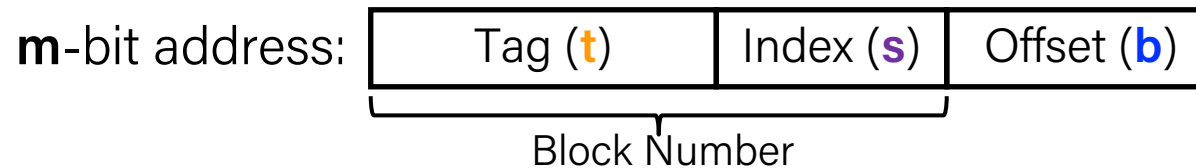
# Tags Differentiate Blocks in Same Index



# 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

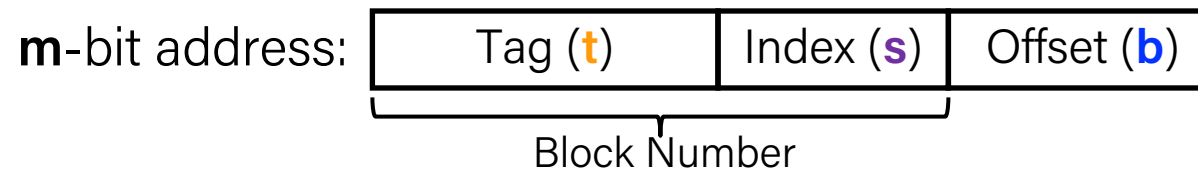
- TIO address breakdown:



- **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

# Checking for a Requested Address Example

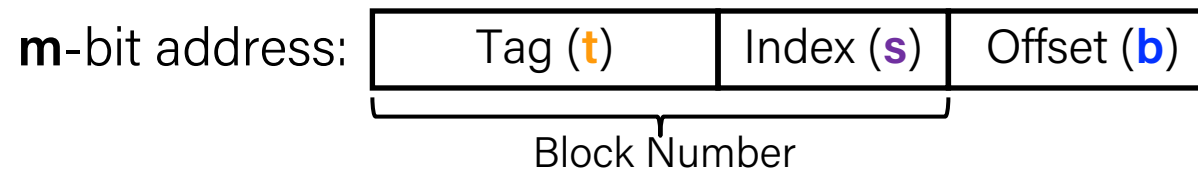
- Using 8-bit addresses.
- Cache Params: block size (B) = 4 bytes, cache size (C) = 32 bytes (which means number of sets is  $C/B = 8$  sets).
  - Offset bits (b) =  $\log_2(B) = 2$  bits
  - Index bits (s) =  $\log_2(\text{number of sets}) = 3$  bits
  - Tag bits (t) = Rest of the bits in the address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



- What are the fields for address 0xBA?
  - Tag bits (unique id for block):
  - Index bits (cache set block maps to):
  - Offset bits (byte offset within block):

# Checking for a Requested Address Example

- 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

# 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

A. 4 bytes

B. 8 bytes

C. 16 bytes

**D. 32 bytes**

E. We're lost...

- 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

	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 ✓ ✓		



# Recap

- Cache basics
- Principle of locality
- Cache organization

**Next time:** More cache memories