photo by unsplash.com user @martijnbaudoin

COMP201

Computer Systems & Programming

Lecture #22 – Cache Memories

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

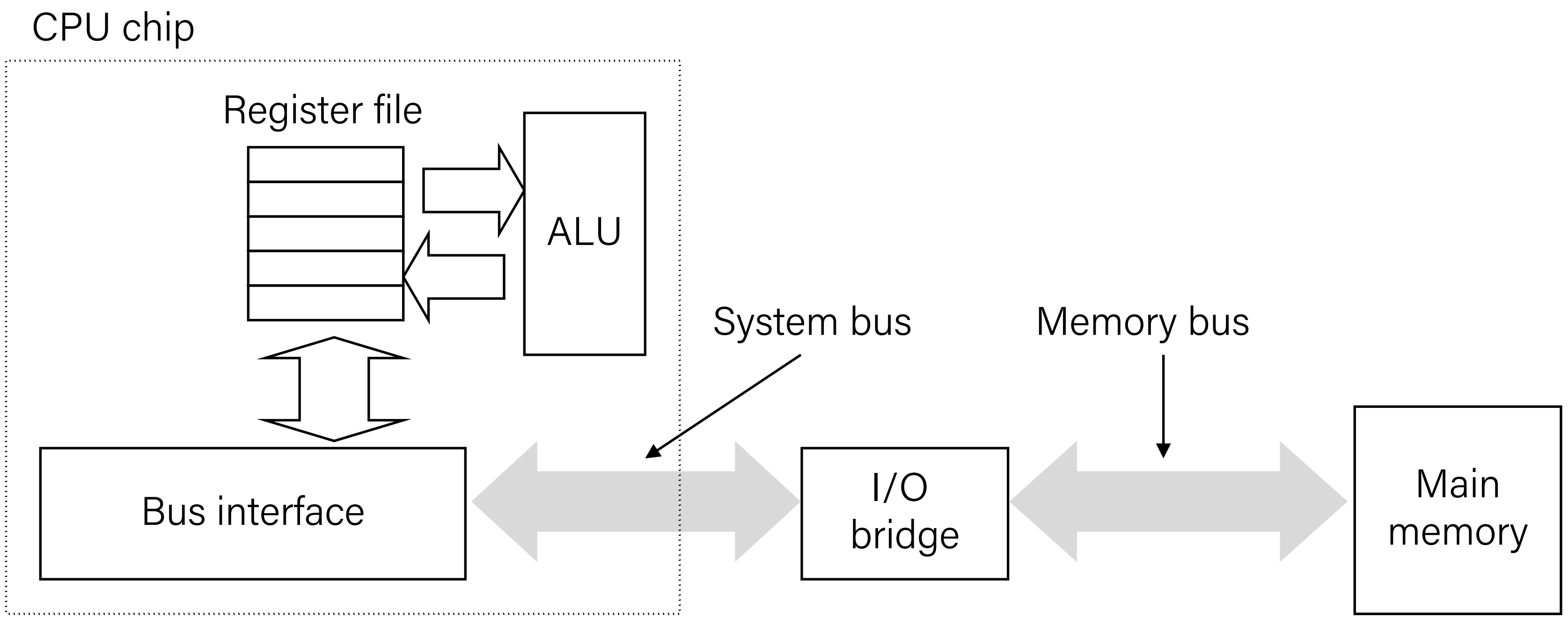
- There will be no labs this week!

Recap

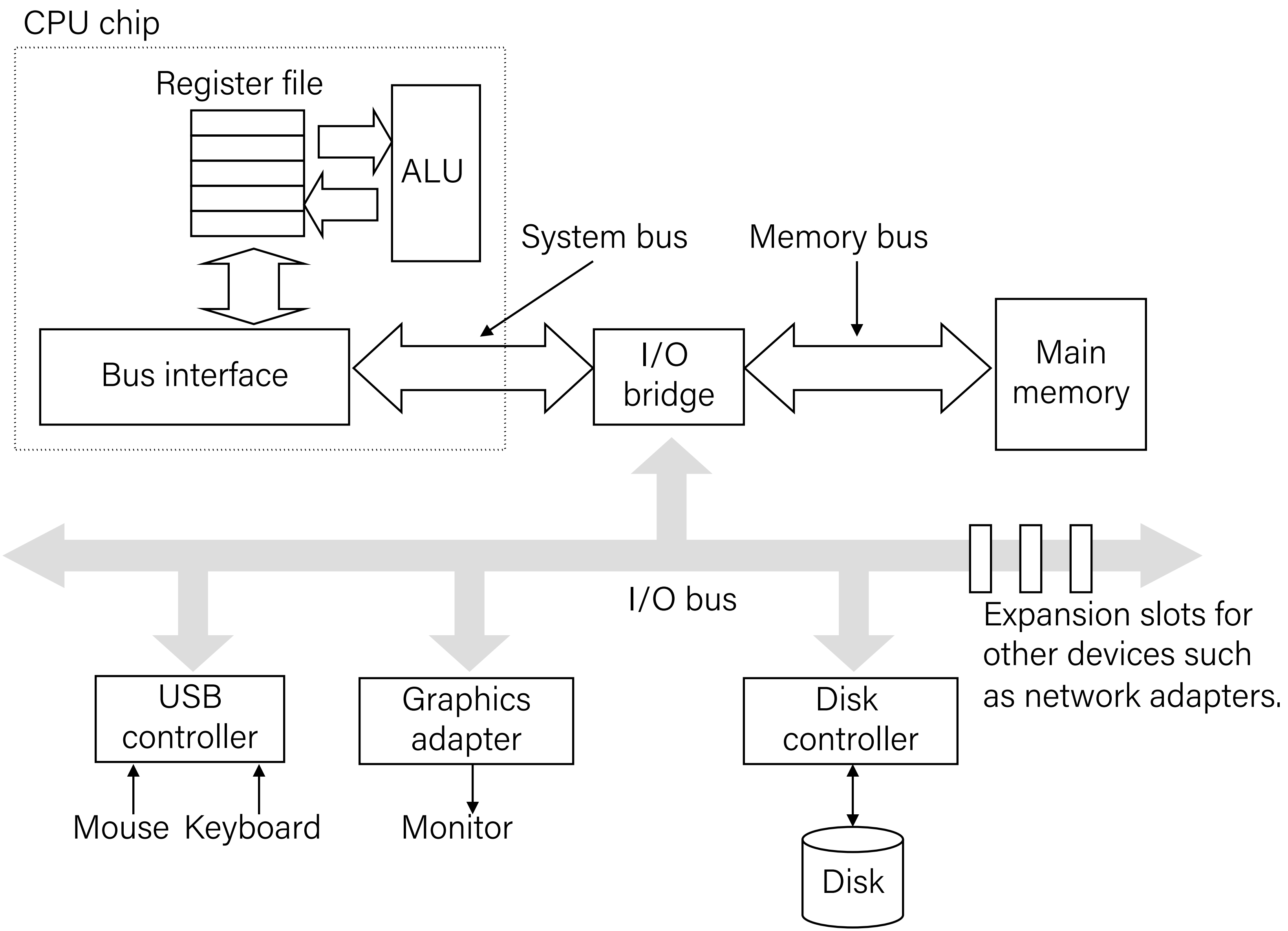
- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

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

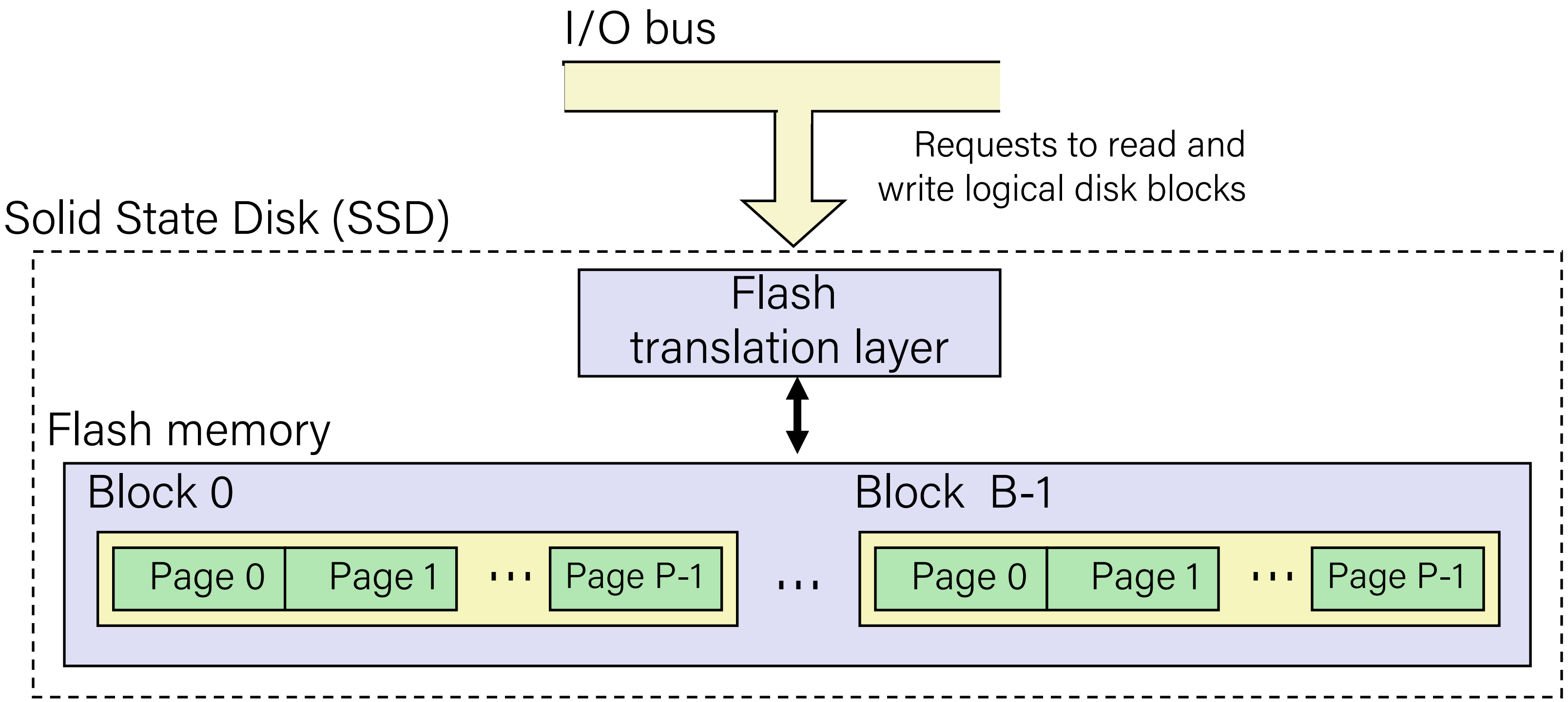

CPU chip
Register file
ALU
System bus
Memory bus
Bus interface
I/O
bridge
Main
memory
I/O bus
Expansion slots for
other devices such
as network adapters.
USB
controller
Graphics
adapter
Disk
controller
Mouse
Keyboard
Monitor
Disk

Recap: Disk Access Time

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

Access time is dominated by seek time and rotational latency

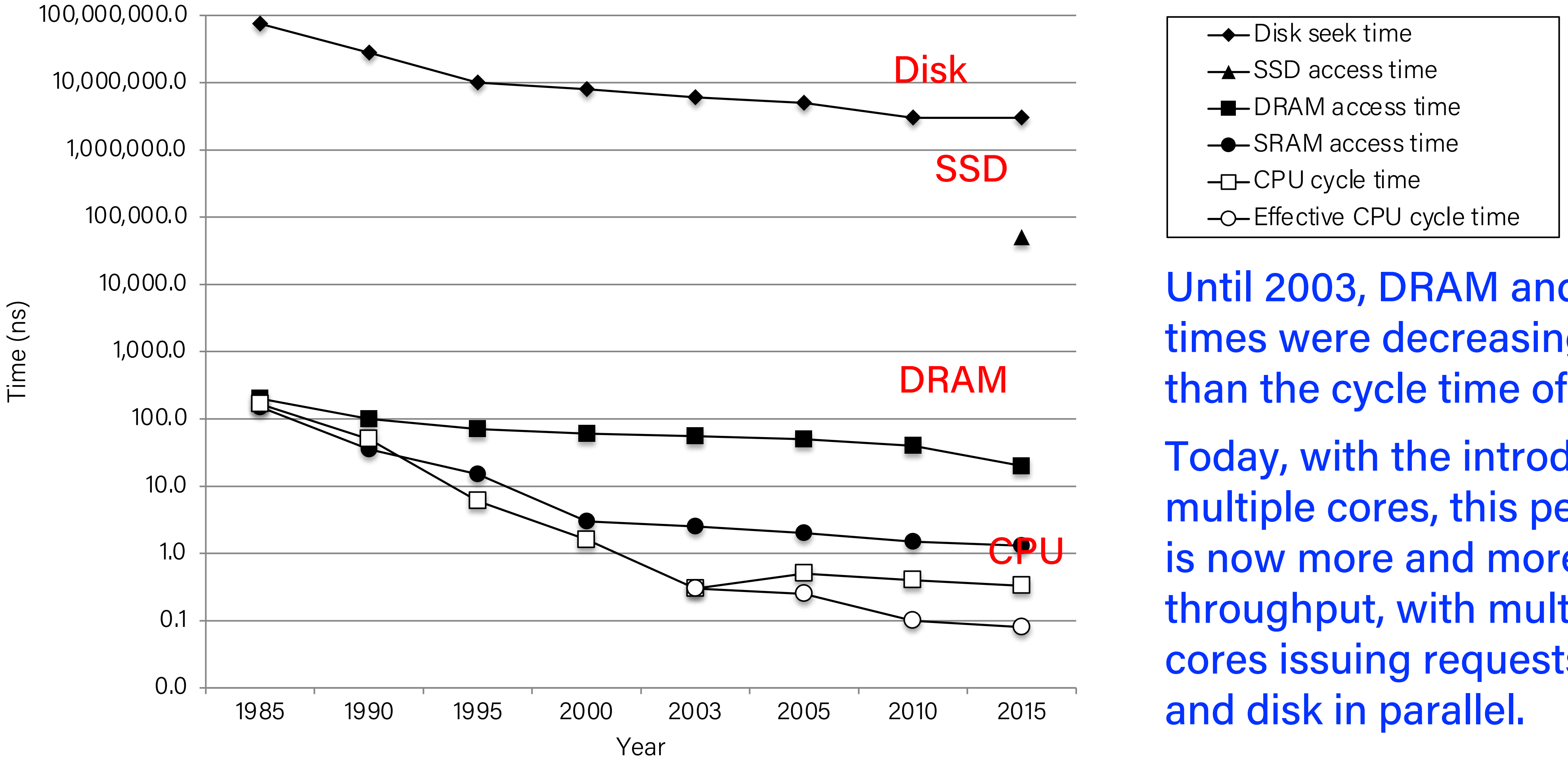
Recap: Solid State Disks (SSDs)



- 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.



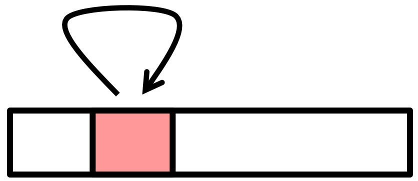
Until 2003, DRAM and disk access times were decreasing more slowly than the cycle time of a processor.

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.

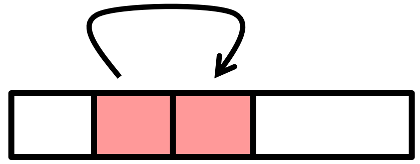
Recap: Locality

- **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



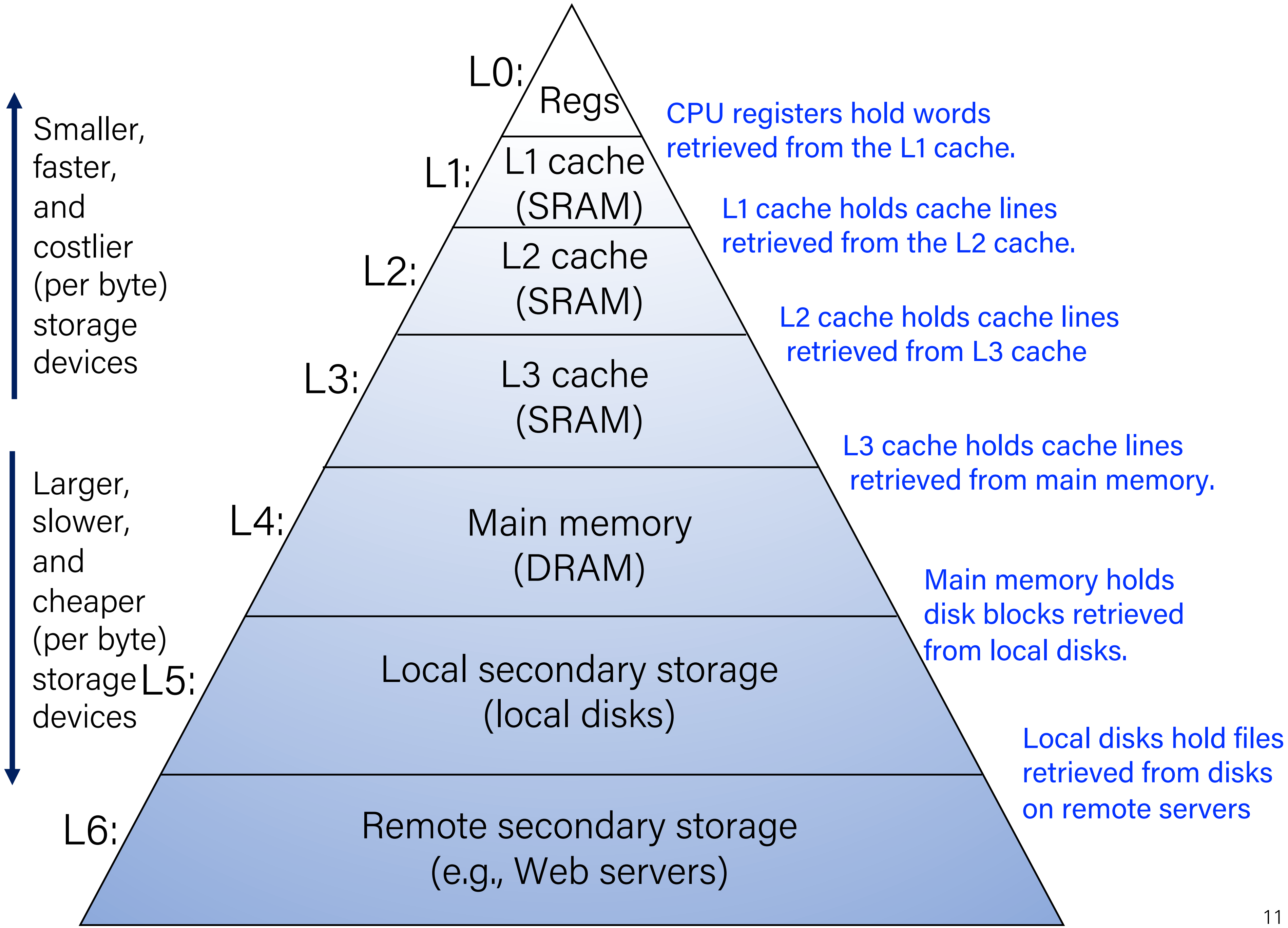
Well-written programs tend to exhibit good locality!

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). Spatial locality
  - Reference variable sum each iteration. Temporal locality
- Instruction references
  - Reference instructions in sequence. Spatial locality
  - Cycle through loop repeatedly. Temporal locality

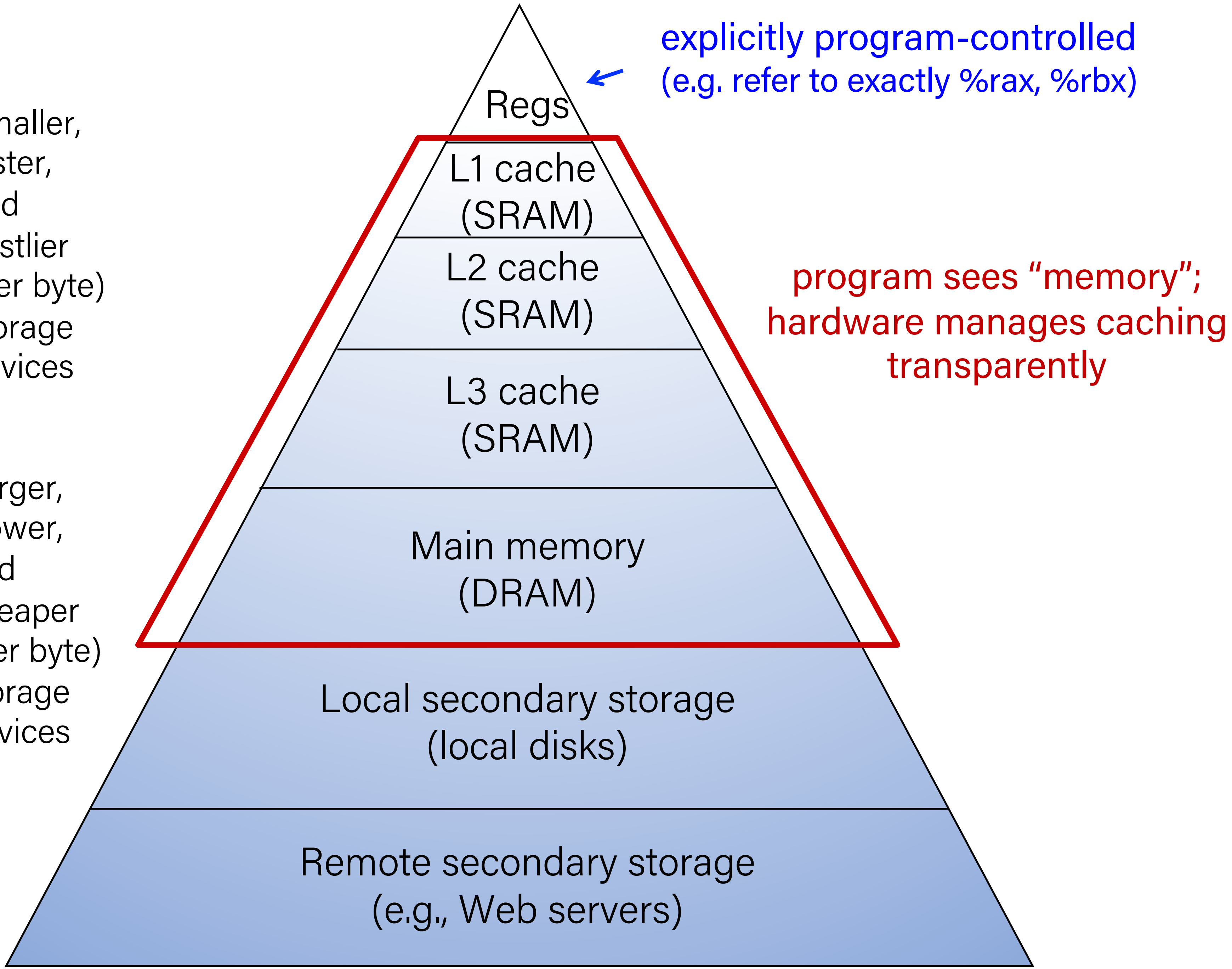
Recap: Memory Hierarchy



Recap: Memory Hierarchy

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

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



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'Hallaroni's CMU 15-213 class

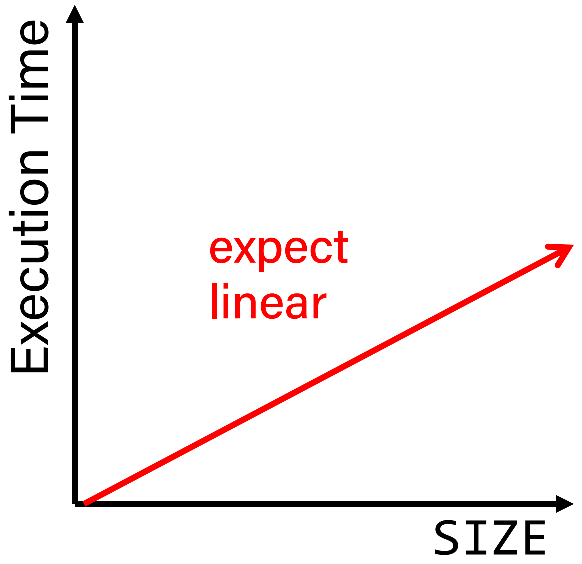
—Porter Jones' UW CSE 351 class

How does execution time grow with SIZE?

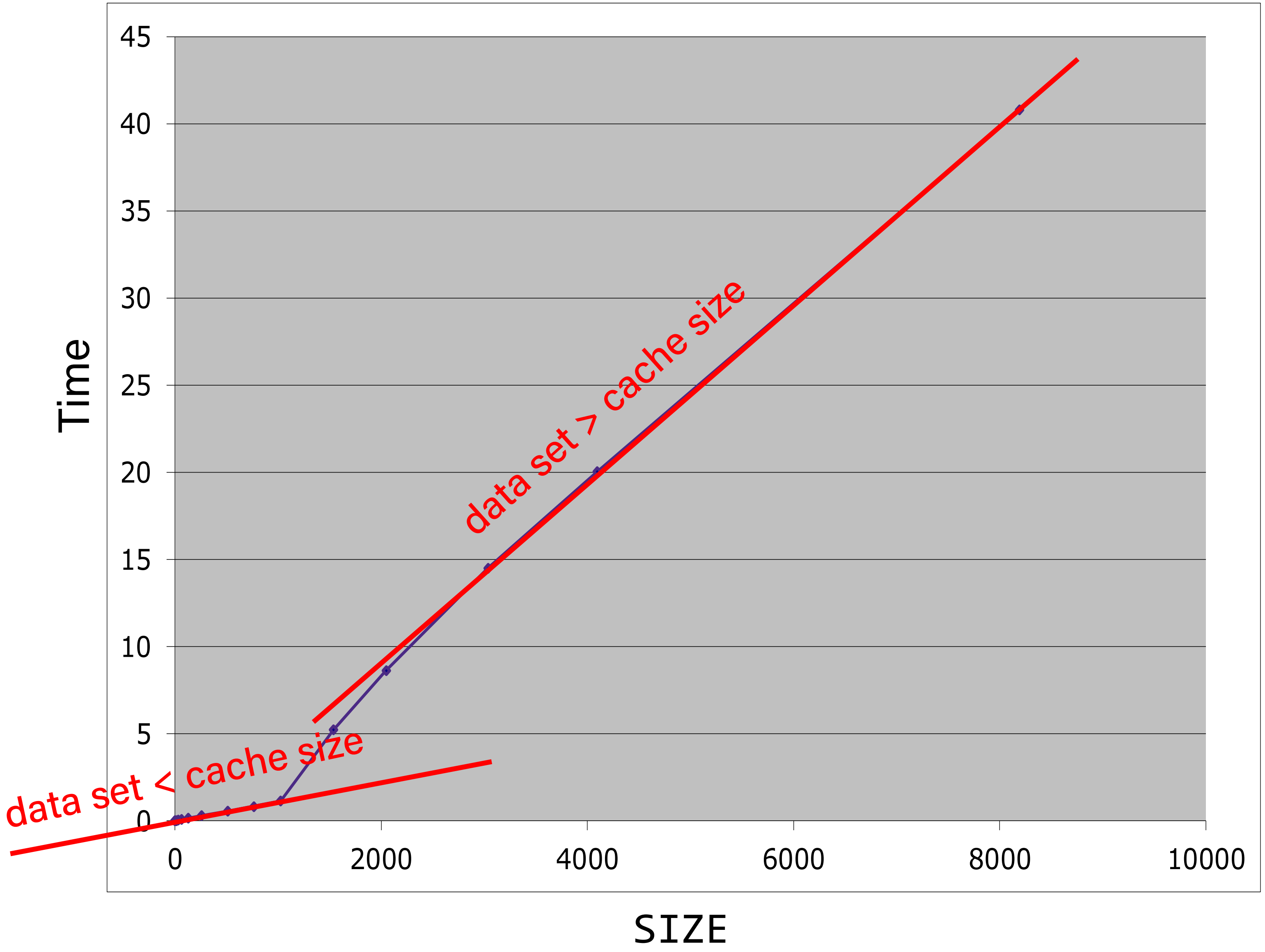
```
int array[SIZE];
int sum = 0;

for (int i = 0; i < 200000; i++) {
  for (int j = 0; j < SIZE; j++) {
    sum += array[j]; ← execute SIZE×200,000 times
  }
}
```

repeat 200,000 times

Plot:

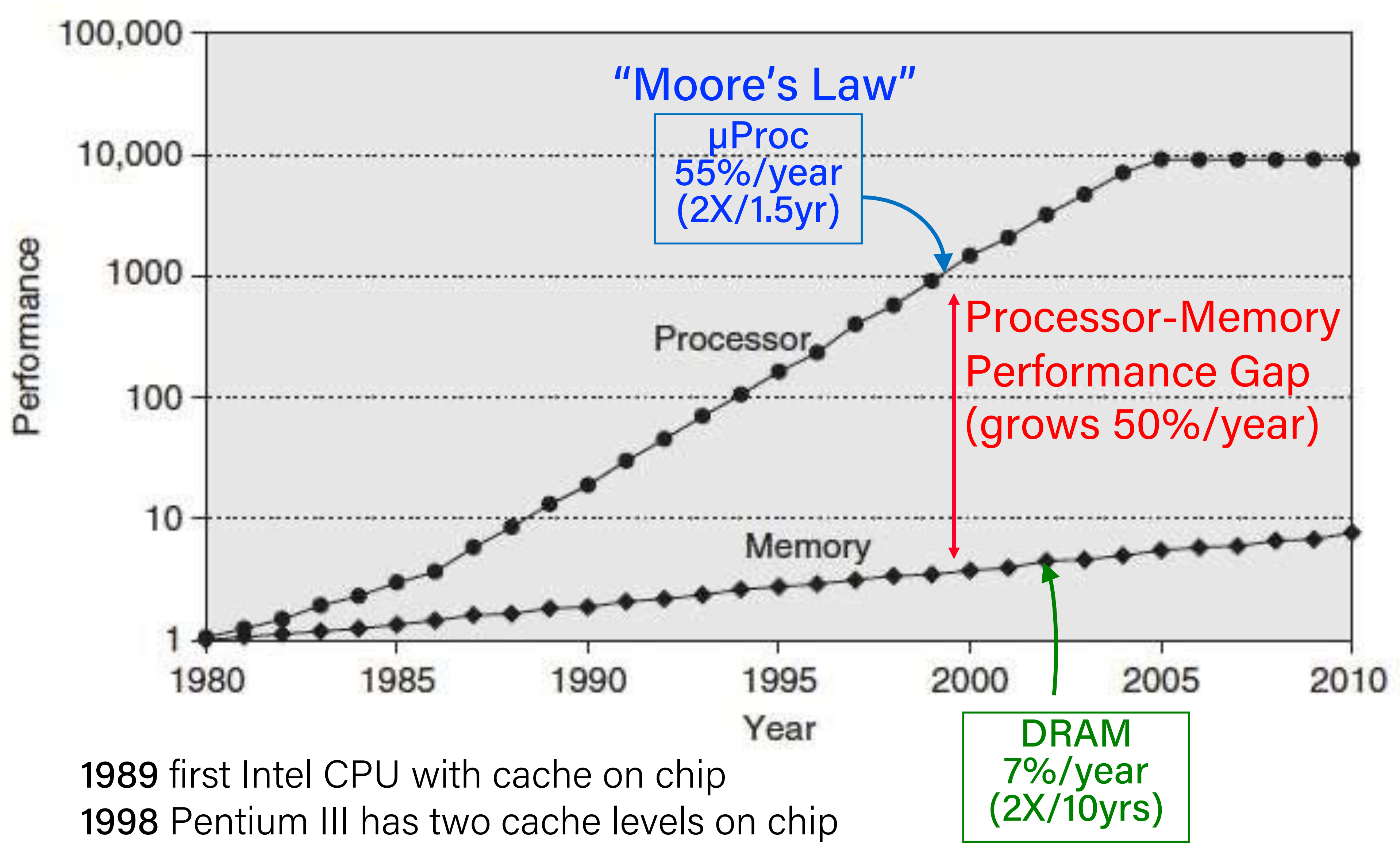


Actual Data



45
40
35
30
25
20
15
10
5
0
Time
0
2000
4000
6000
8000
10000
SIZE
data set > cache size
data set < cache size

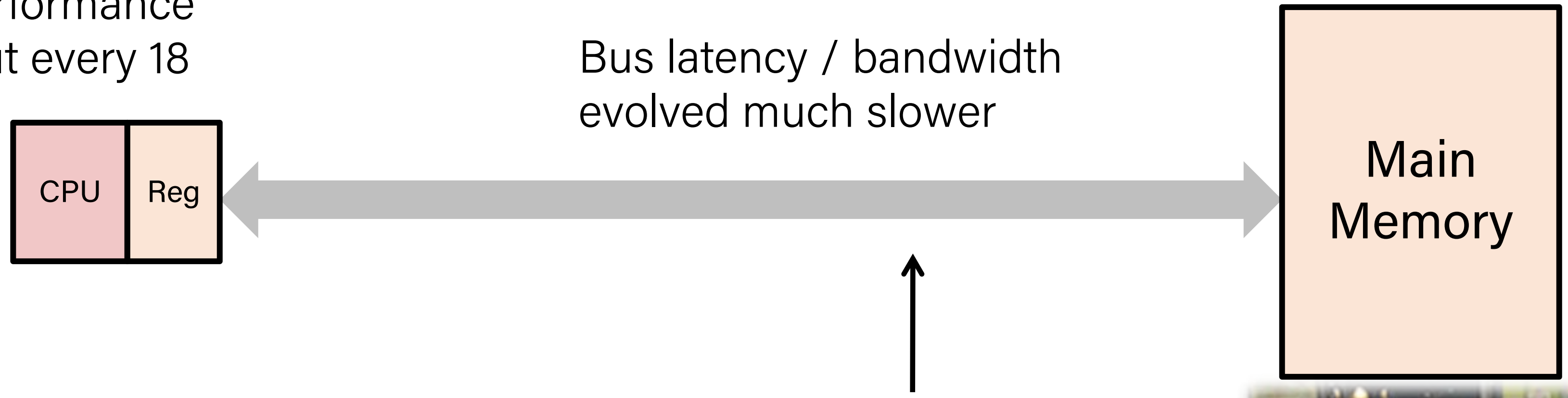
Processor-Memory Gap



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



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

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

cycle: single machine step (fixed-time)

Problem: lots of waiting on memory

Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

CPU | Reg

Cache

Bus latency / bandwidth evolved much slower

Main Memory

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

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

cycle: single machine step (fixed-time)

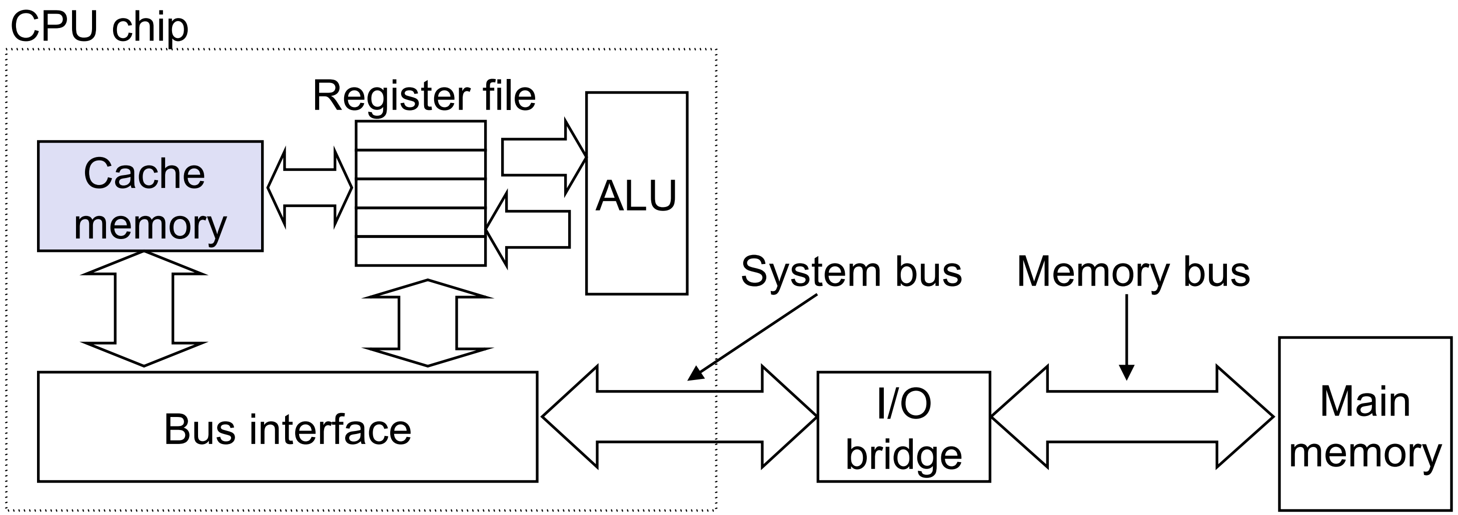
Solution: caches

Lecture Plan

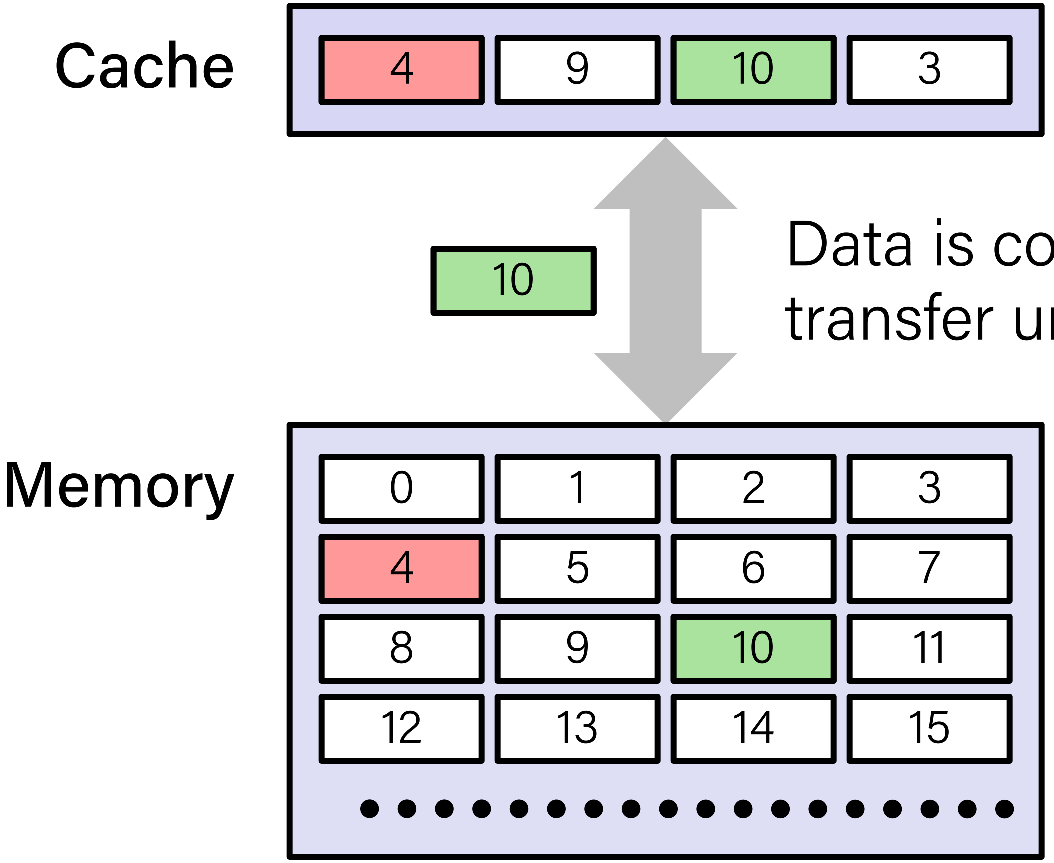
- Cache basics
- Principle of locality
- Cache organization

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

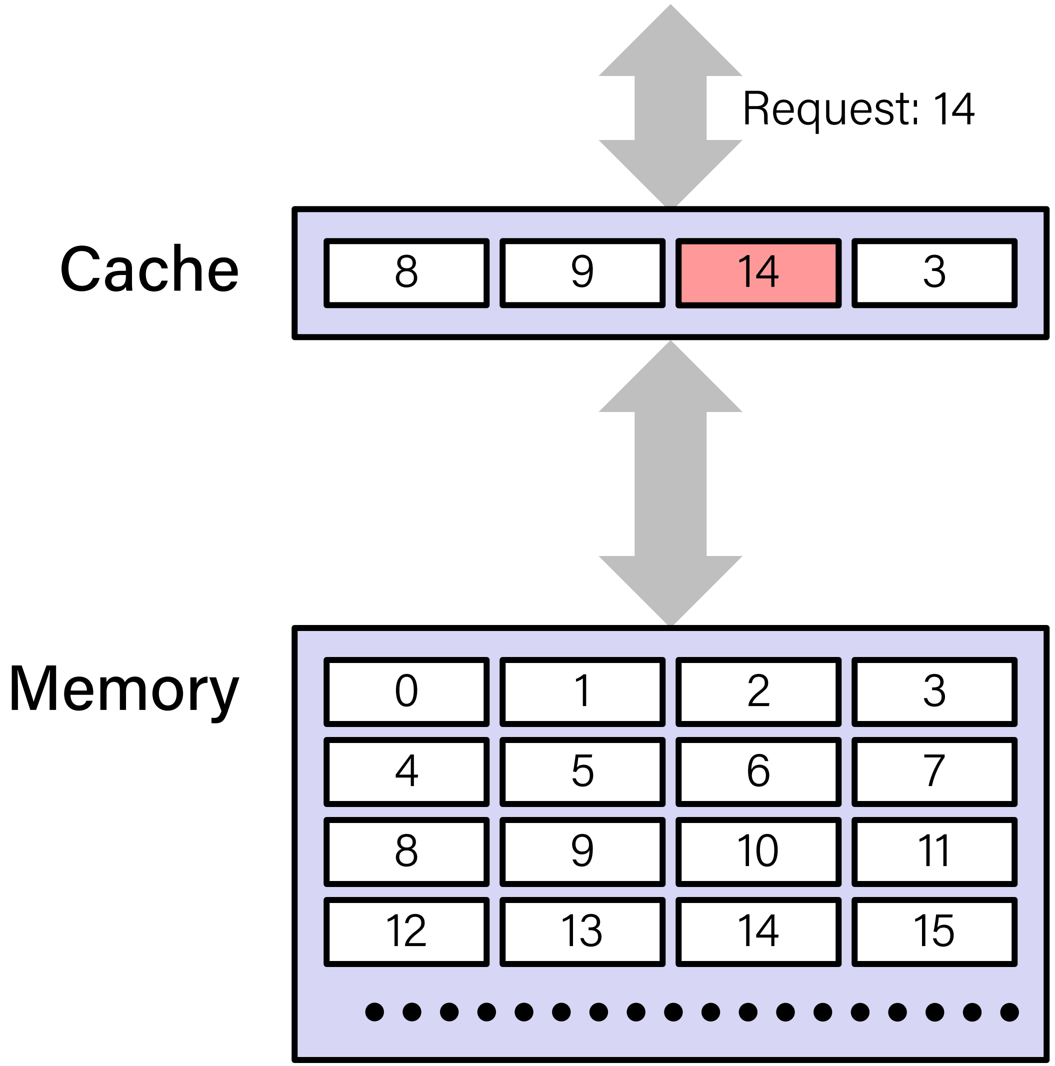


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

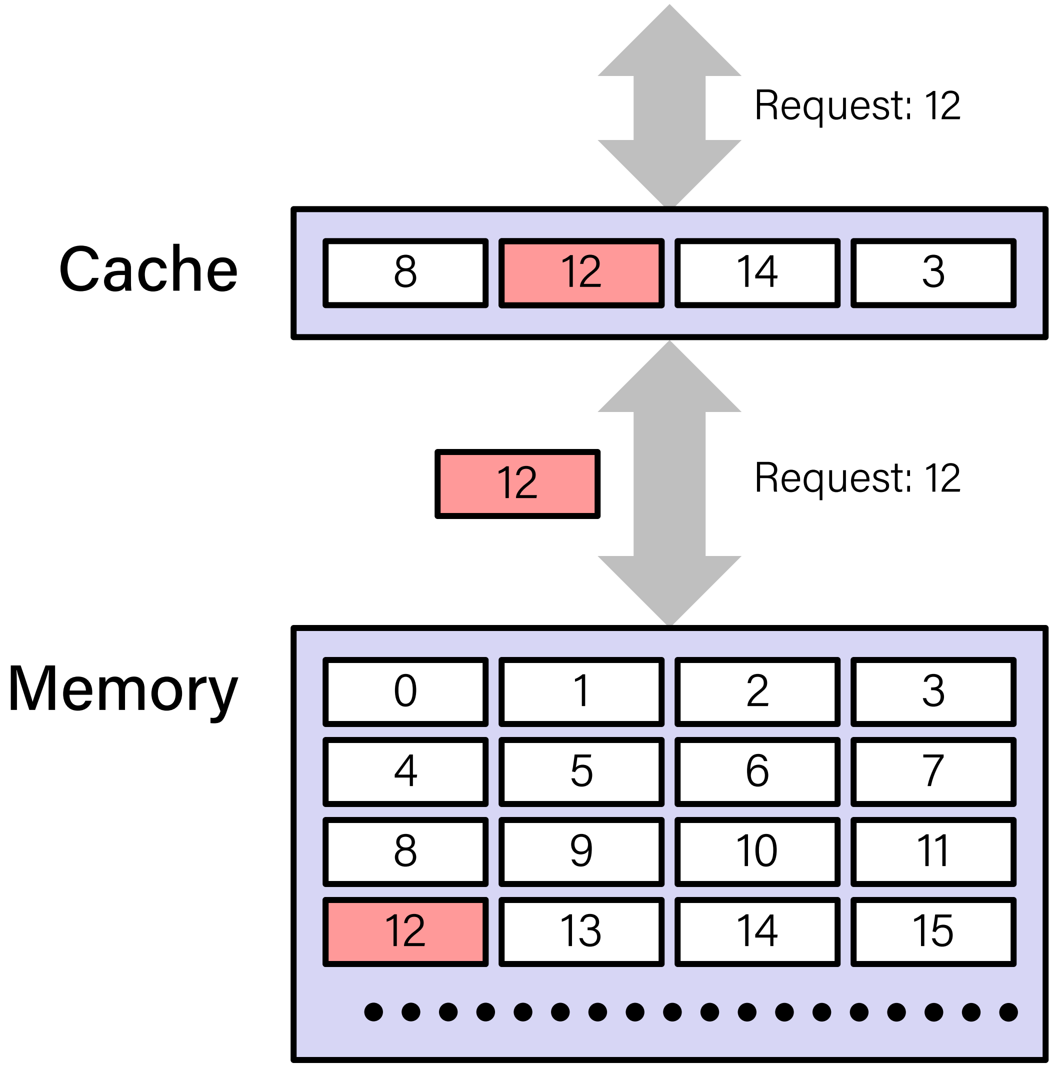
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache: **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 mod 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.

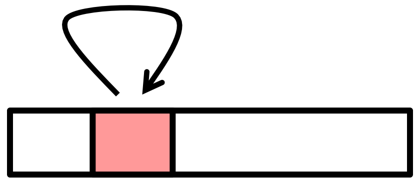
Lecture Plan

- Cache basics
- Principle of locality
- Cache organization

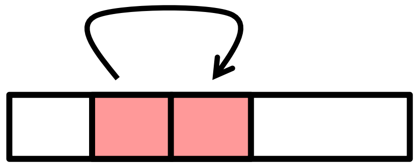
Why Caches Work

- **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; j < M; j++)
        for (j = 0; i < N; 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 = 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[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]
76				92				108			

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]

Layout in Memory

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]

76 92 108

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 [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]

76 92 108

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

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;
}
```

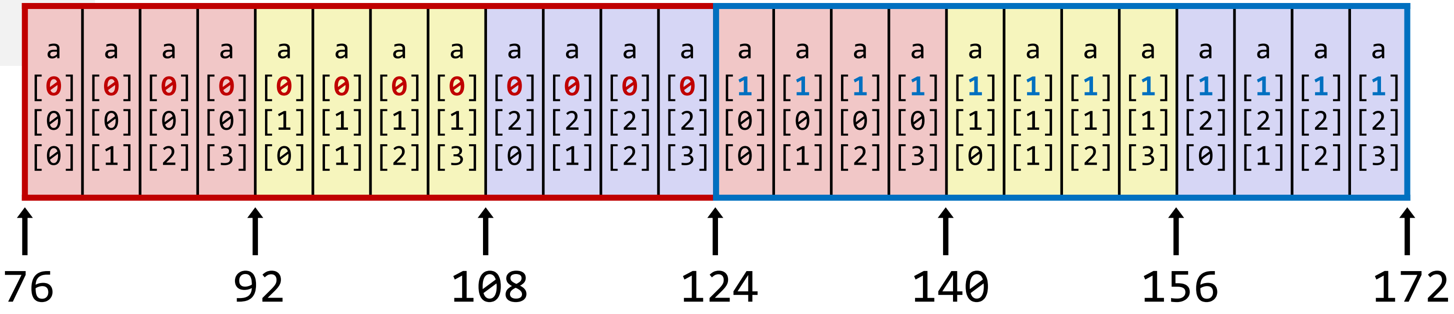
- What is wrong with this code?

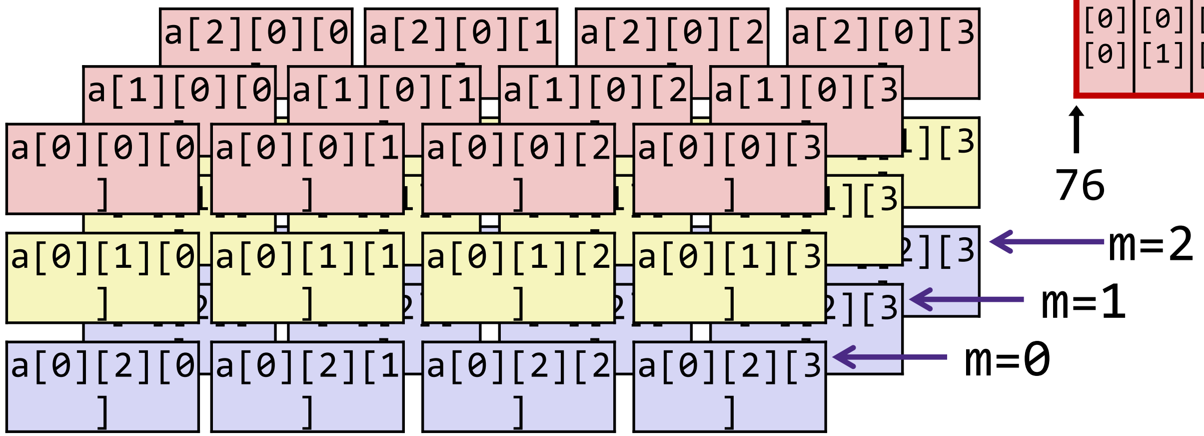
  Access Pattern: stride-N×L

- How can it be fixed?

  Inner loop: i → stride-1
  j → stride-1
  k → stride-N×L

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





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 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 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 × 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
    - (address) mod 2^n = n lowest bits of address
  - – (address) modulo (# of bytes in a block)

	m - b bits	b bits
m-bit address: (refers to byte in memory)	Block Number	Block Offset

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

slido

- 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...	

$$\text{Address: 0b } \underbrace{\underline{0}\ \underline{1}}^{\text{0x }1}\ \underbrace{\underline{0}\ \underline{1}\,/\,\underline{0}\ \underline{1}}^{5}$$

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

Cache Organization

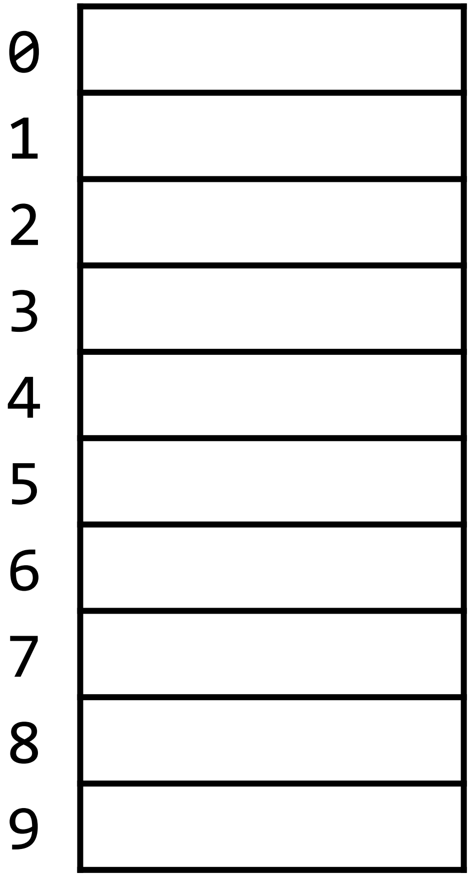
- **Cache Size (C):** amount of *data* the cache can store
  - Cache can only hold so much data (subset of next level)
  - Given in bytes (C) or number of blocks (C/B)
  - <u>Example</u>: C = 32 KiB = 512 blocks if using 64-byte blocks
- Where should data go in the cache?
  - We need a mapping from memory 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

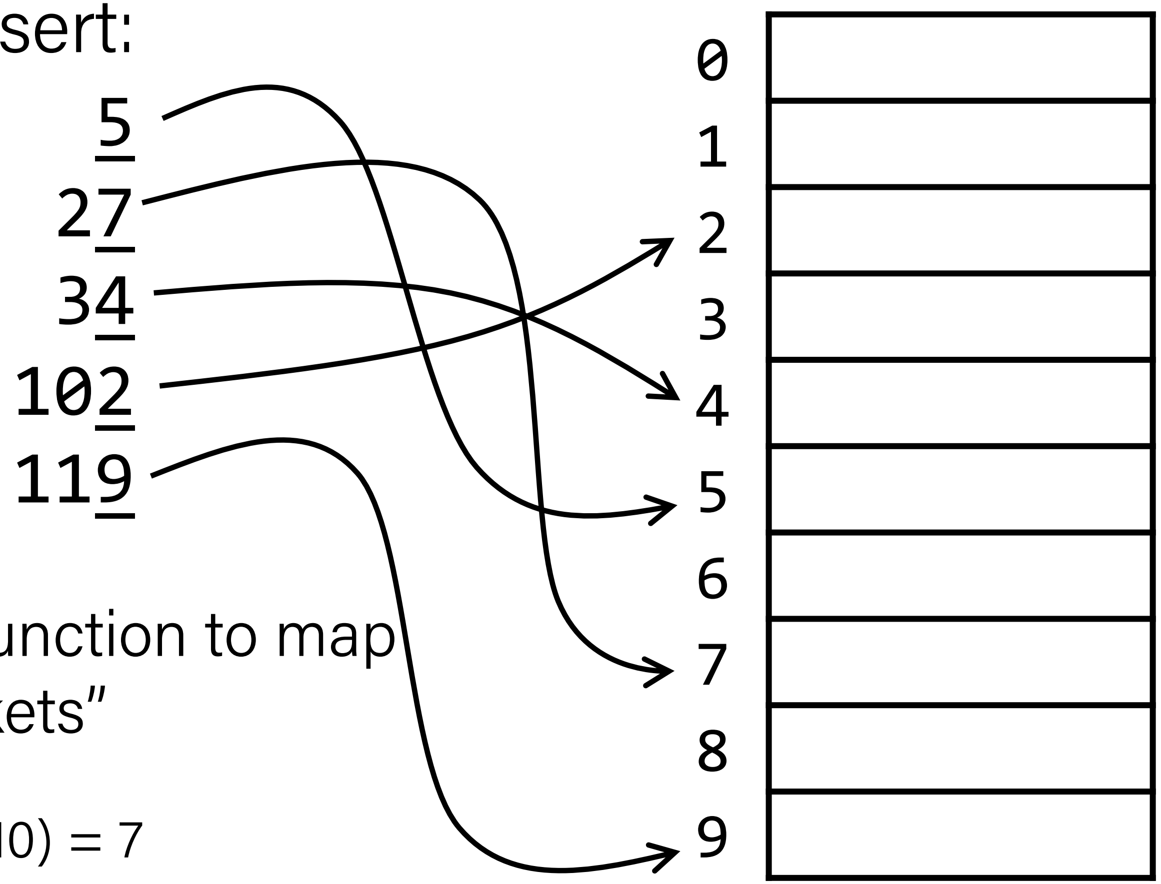
Apply hash function to map data to “buckets”

hash(27) % N (10) = 7



Review: Hash Tables for Fast Lookup

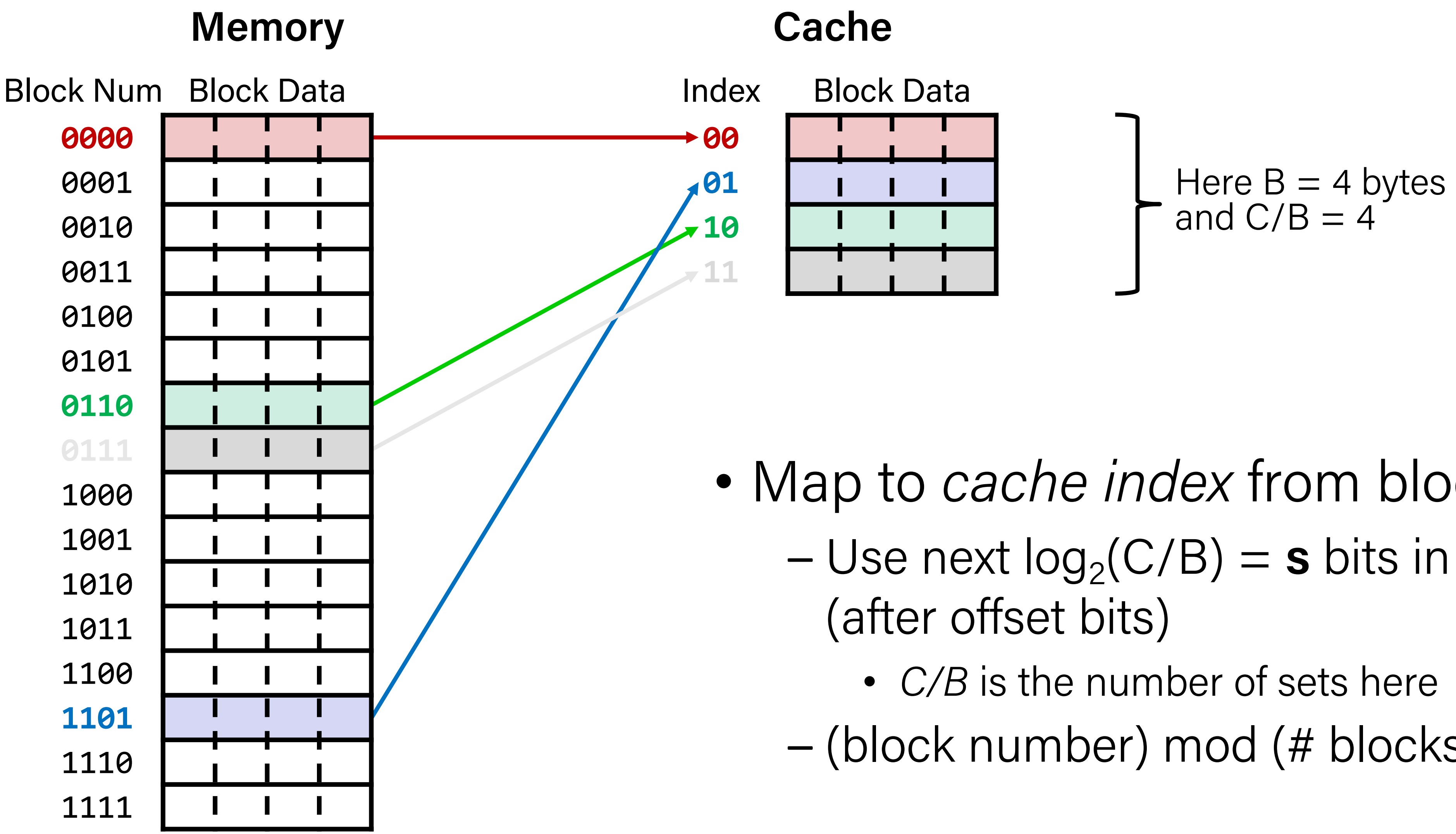
Insert:



Apply hash function to map data to “buckets”

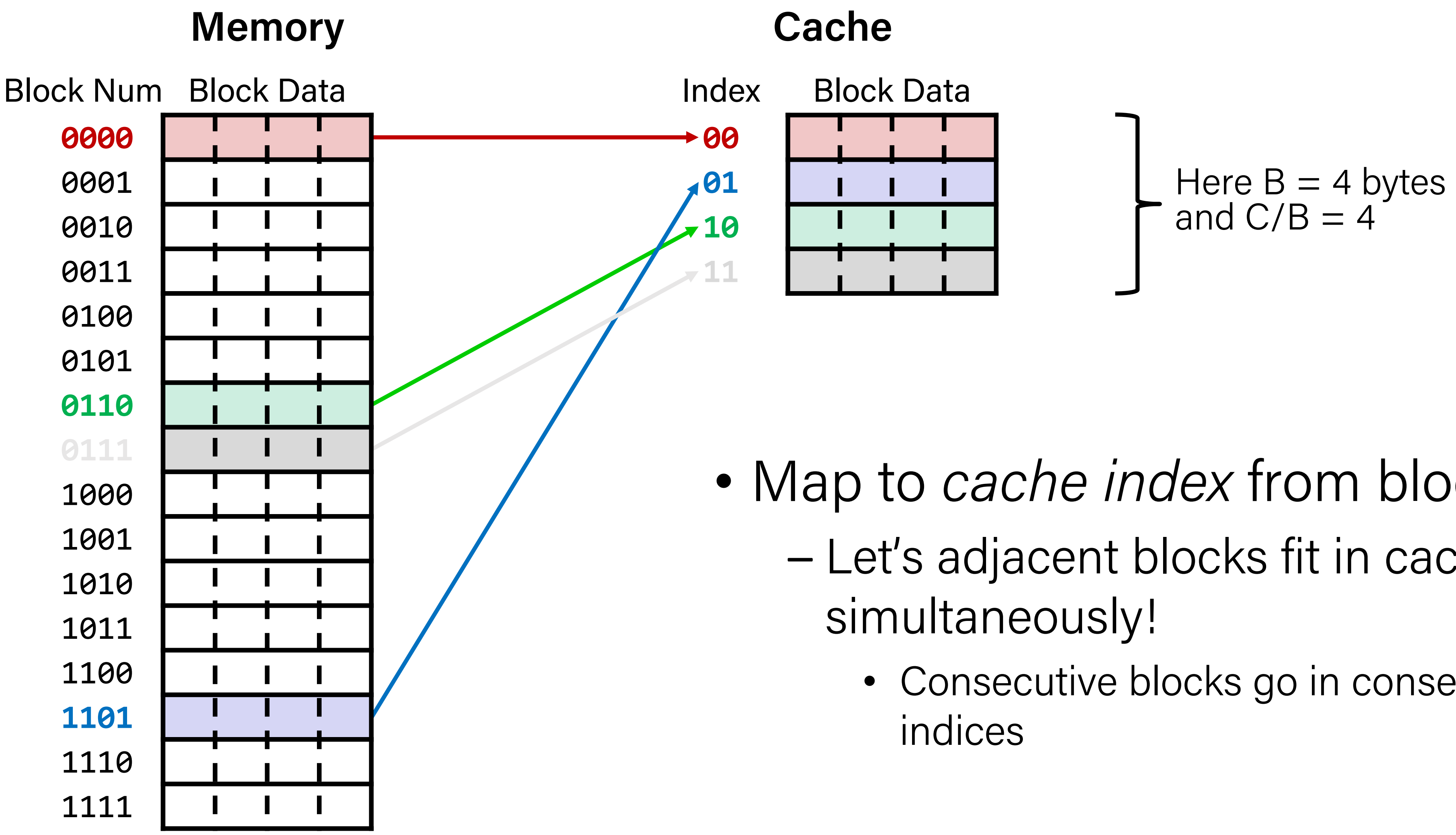
hash(27) % N (10) = 7

Place Data in Cache by Hashing Address



- Map to *cache index* from block number
  - Use next $\log_2(C/B)$ = **s** bits in the address (after offset bits)
    - *C/B* is the number of sets here
  - (block number) mod (# blocks in cache)

Place Data in Cache by Hashing Address



- Map to *cache index* from block number
  - Let's adjacent blocks fit in cache simultaneously!
    - Consecutive blocks go in consecutive cache indices

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?

$$\text{Address: } 0\text{b}\, \underbrace{\overbrace{1\,0}^{0\text{x}\ 2} \mid \overbrace{1\,0}^{\text{index}}}_{\text{block number}} \overbrace{\mid \underbrace{1\,0}_{\text{offset}}}^{\text{A}}$$

addresses w/block number 1010

0b101000 = 0x28

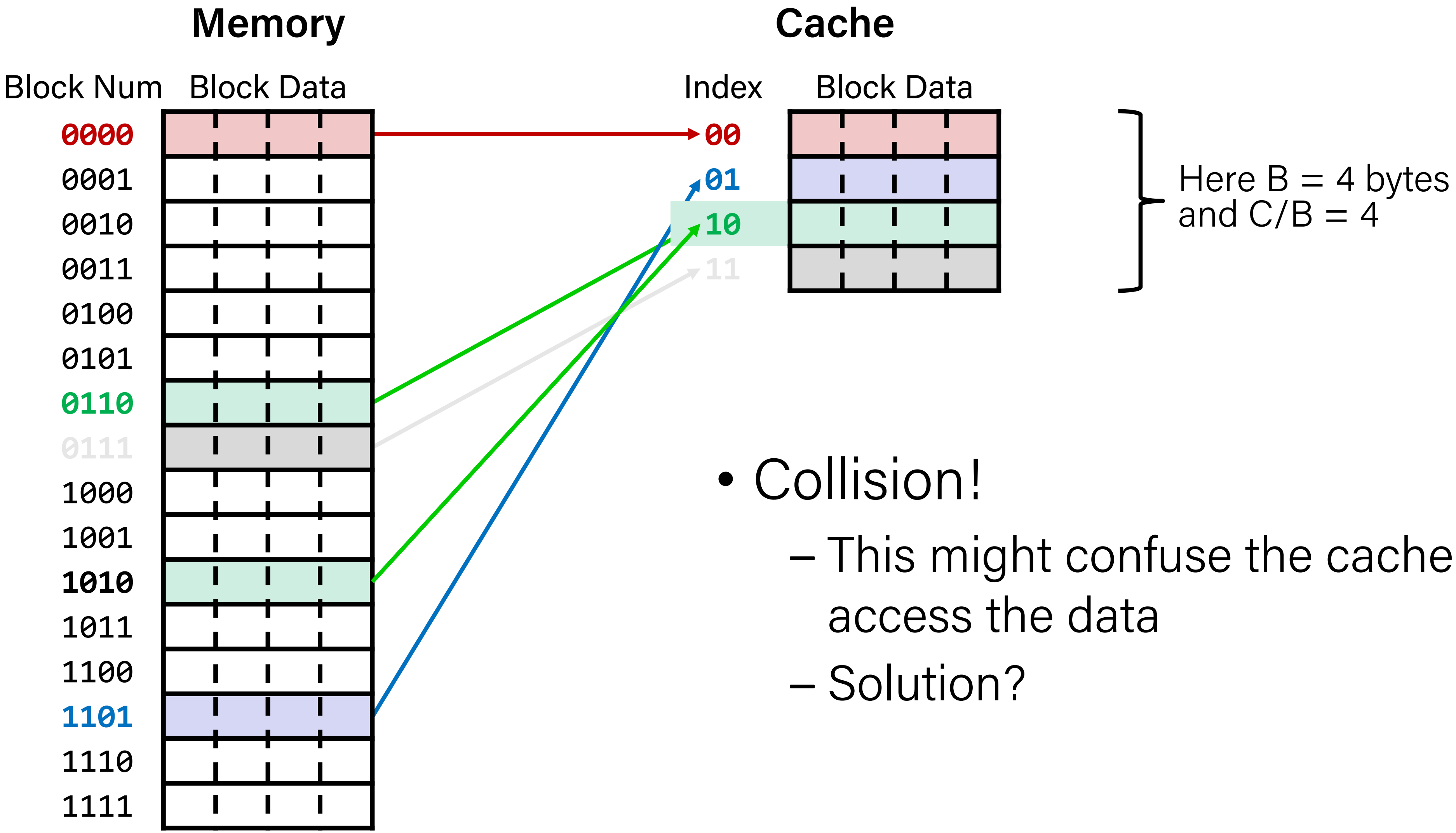
0b101001 = 0x29

0b101010 = 0x2A

0b101011 = 0x2B

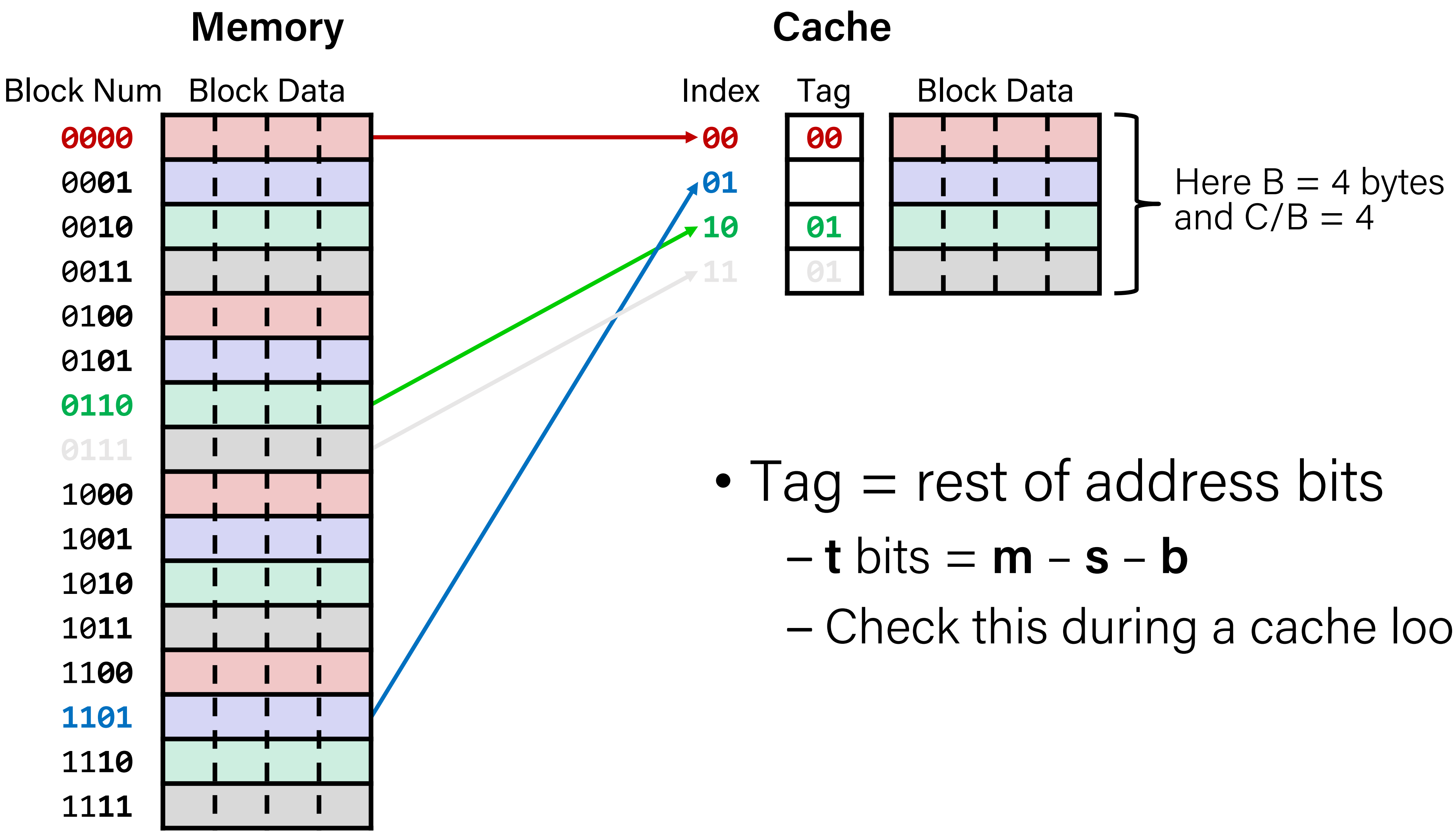
These are loaded into cache!

Place Data in Cache by Hashing Address



- Collision!
  - This might confuse the cache later when we access the data
  - Solution?

Tags Differentiate Blocks in Same Index



- Tag = rest of address bits
  - **t** bits = **m** – **s** – **b**
  - Check this during a cache lookup

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 ≠ their phone number

- TIO address breakdown:

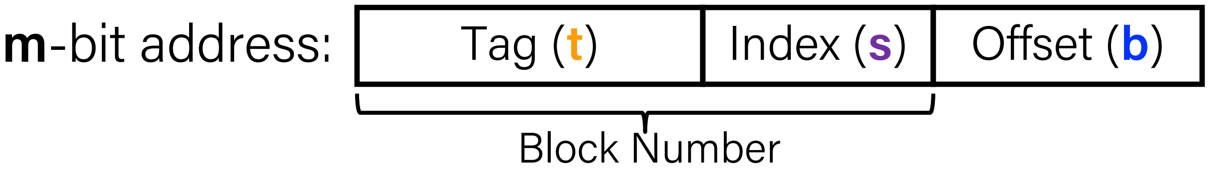
m-bit address: $\underbrace{\boxed{\text{Tag } (\mathbf{t})}\boxed{\text{Index } (\mathbf{s})}}_{\text{Block Number}}\boxed{\text{Offset } (\mathbf{b})}$

  - 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

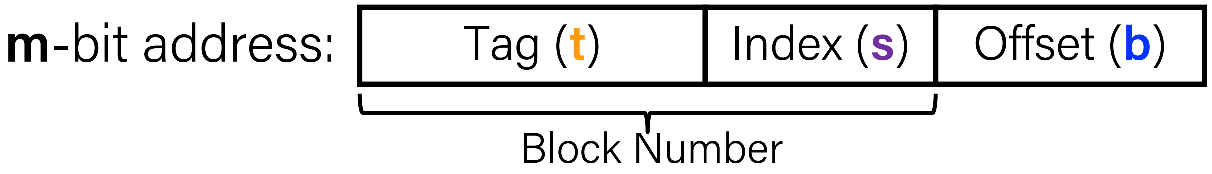
- Using 8-bit addresses.
- Cache Params: block size (B) = 4 bytes, cache size (C) = 32 bytes (which means number of sets is C/B = 8 sets).
  - Offset bits (b) = $\log_2(B)$ = 2 bits
  - Index bits (s) = $\log_2$(number of sets) = 3 bits
  - Tag bits (t) = Rest of the bits in the address = 8 – 2 – 3 = 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
  - Index bits (cache set block maps to): 0x6
  - Offset bits (byte offset within block): 0x2

101 110 10
5 6 2

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...

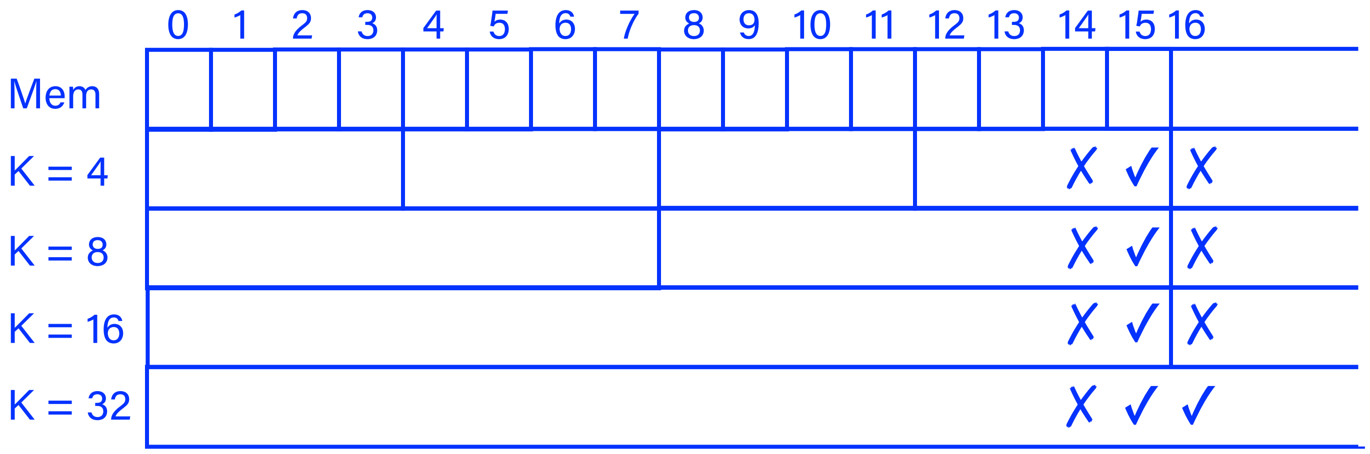
Cache Puzzle

- Based on the following behavior, which of the following block size is <u>NOT</u> possible for our cache?
  - Cache starts *empty*, also known as a **cold cache**
  - Access (addr: hit/miss) stream:
    - (14: miss), (15: hit), (16: miss)

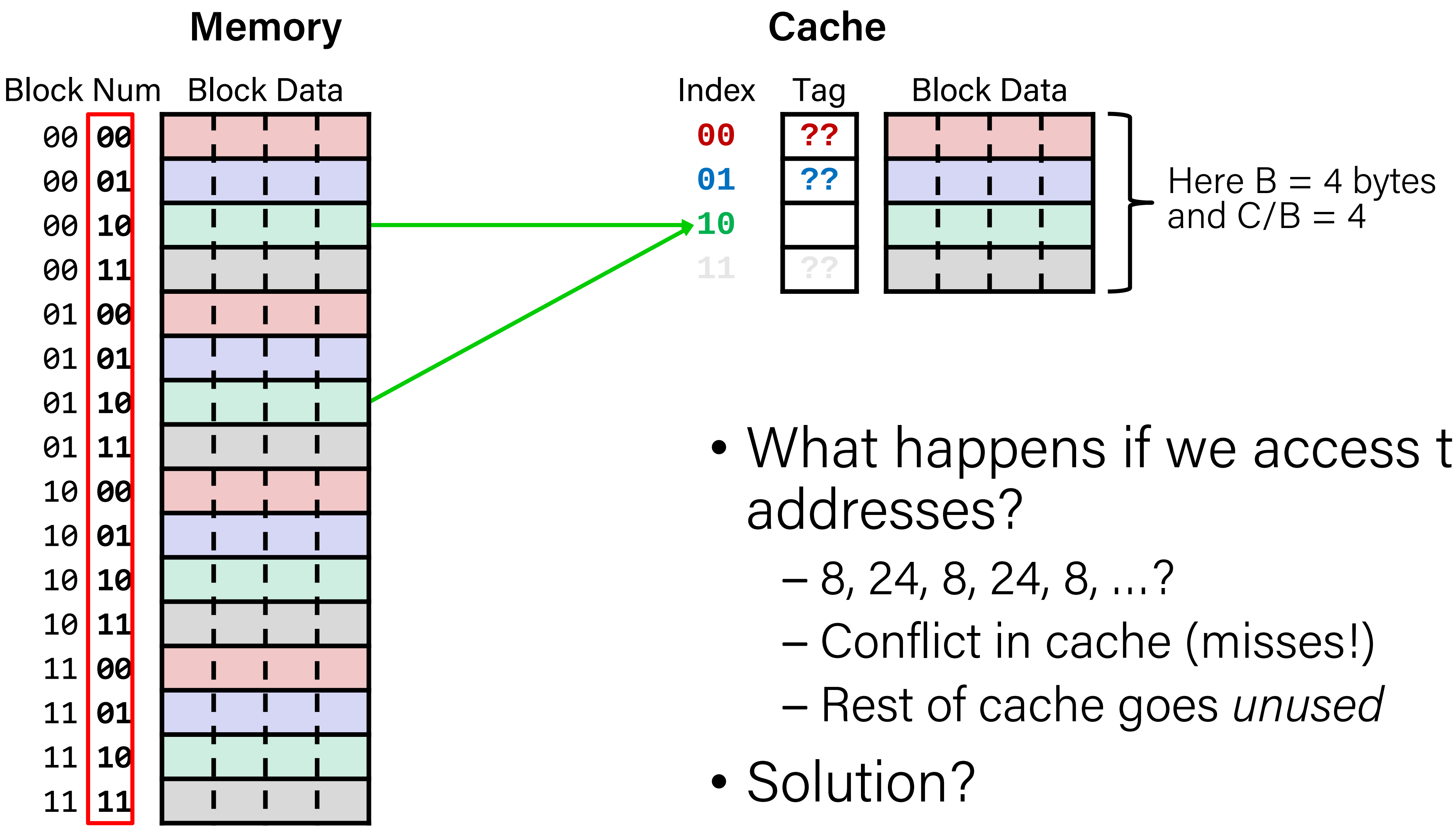
<u>hit</u>: block is already in cache!
<u>miss</u>: block is not in cache, pulls block from memory and puts it in cache

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



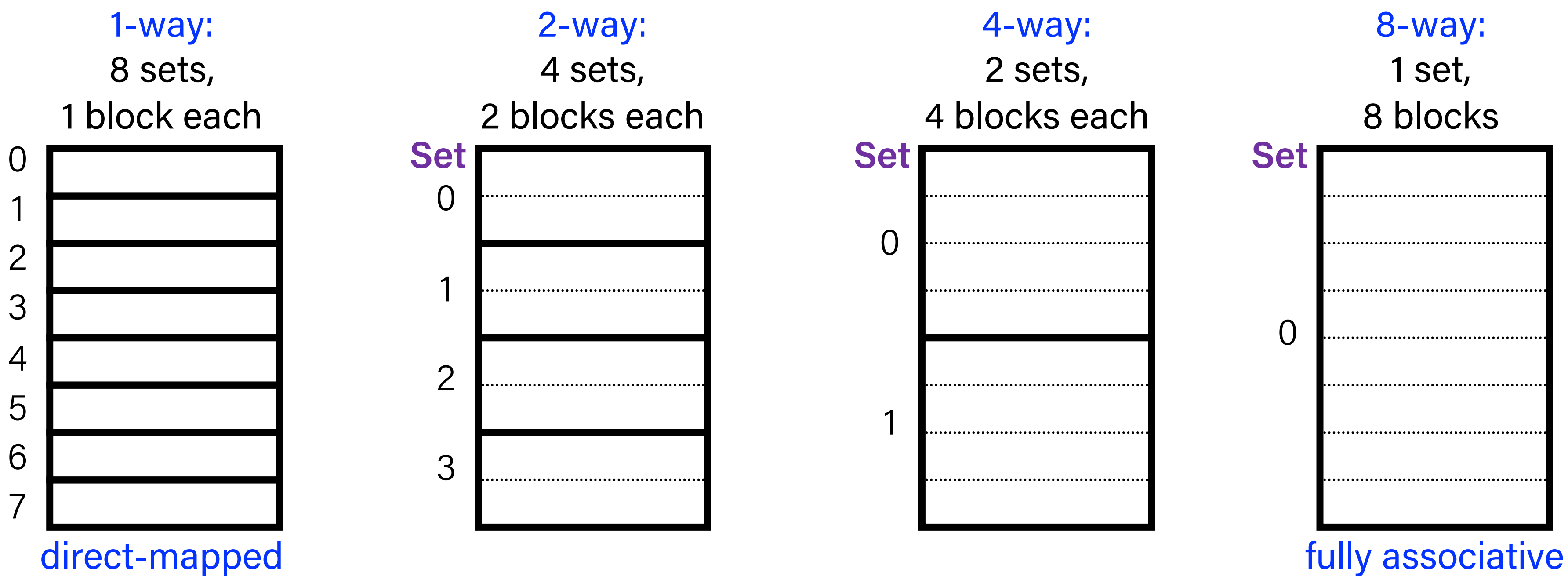
Direct-Mapped Cache Problem



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

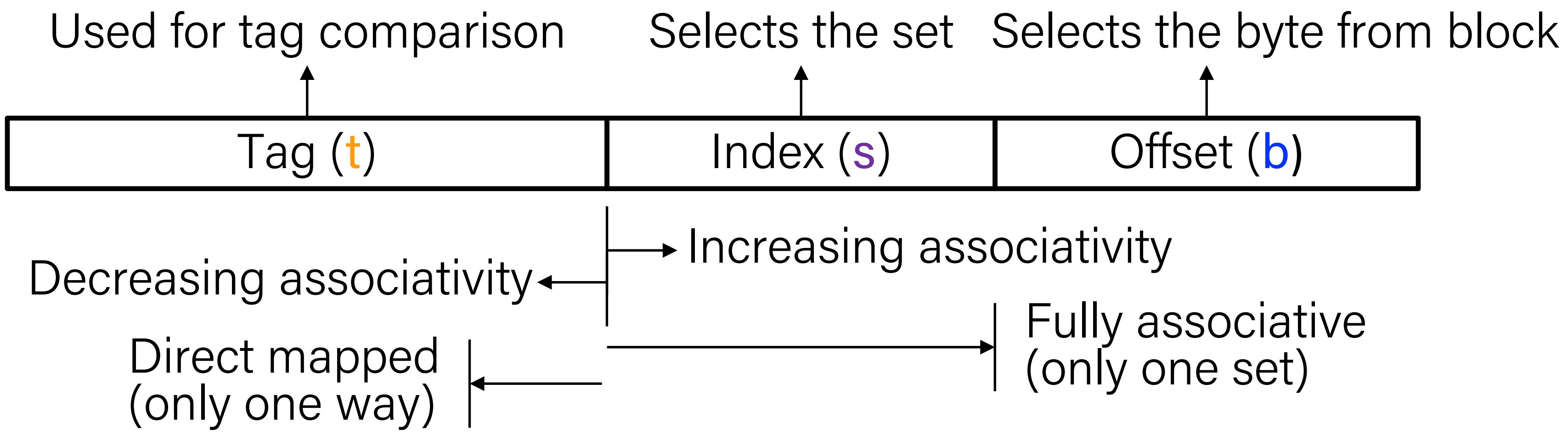
Associativity

- 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**



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** = $\log_2(C/K)$ as we saw previously
    - Fully associative: E = C/K, so **s** = 0 bits



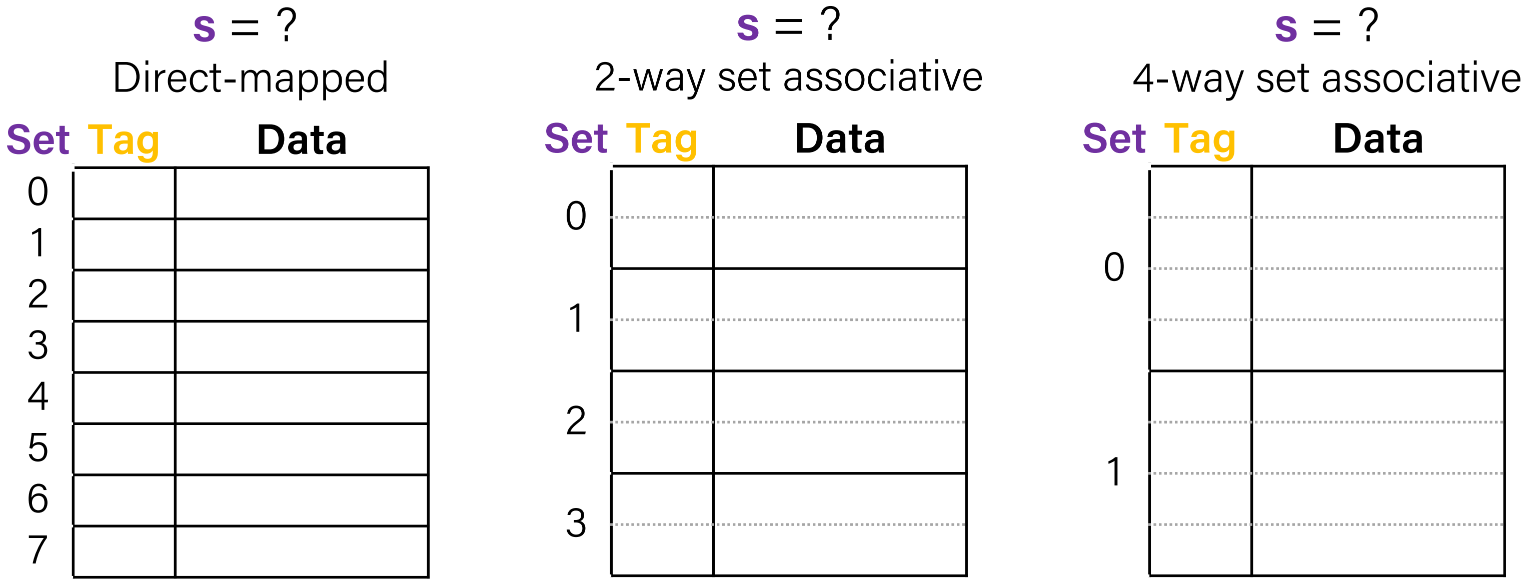
Example Placement

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

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

$t = m - s - b$ $\quad s = \log2(C/B/E)$ $\quad b = \log_2(B)$

m-bit address:	Tag(t)	Index (s)	Offset (b)



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

$t = m - s - b$ $s = \log2(C/B/E)$ $b = \log_2(B)$

m-bit address:

Tag(t)	Index (s)	Offset (b)

$s = \log_2(8) = 3$ bits
Direct-mapped

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

$s = \log_2(8/2) = 2$ bits
2-way set associative

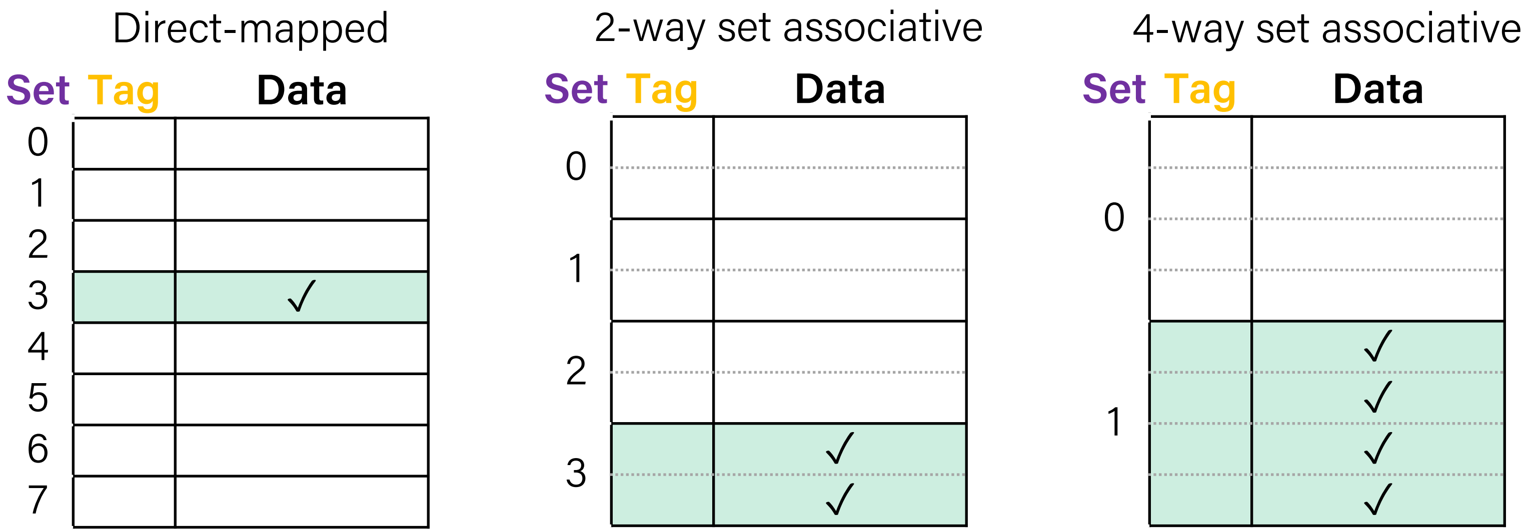
Set	Tag	Data
0		
1		
2		
3		✓
		✓

$s = \log_2(8/4) = 1$ 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*")



Question

slido

- We have a cache of size 2 KiB 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

slido

$(C = 2*2^{10}$ bytes$)$ $(B = 2^7$ 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
  B. 4
  C. 8
  D. 16
  E. We're lost...

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

blocks per set $= E = 16 / 2 = 8$

- If addresses are 16 bits wide, how wide is the Tag field?

Recap

- Cache basics
- Principle of locality
- Cache organization

Next time: More cache memories