

# Computer Architecture, Fall 2019

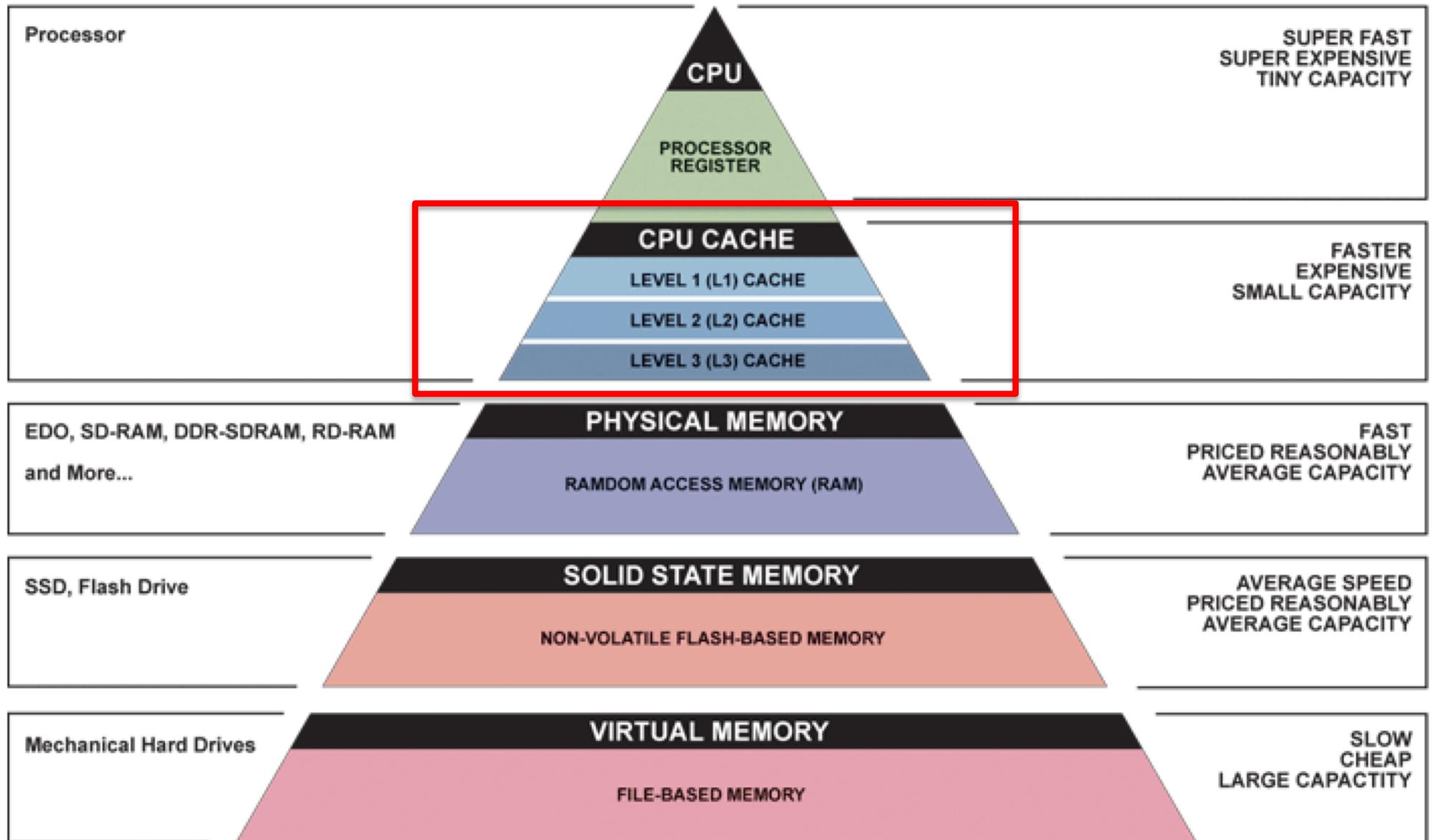
## *Memory Hierarchy, Fully Associative Caches*

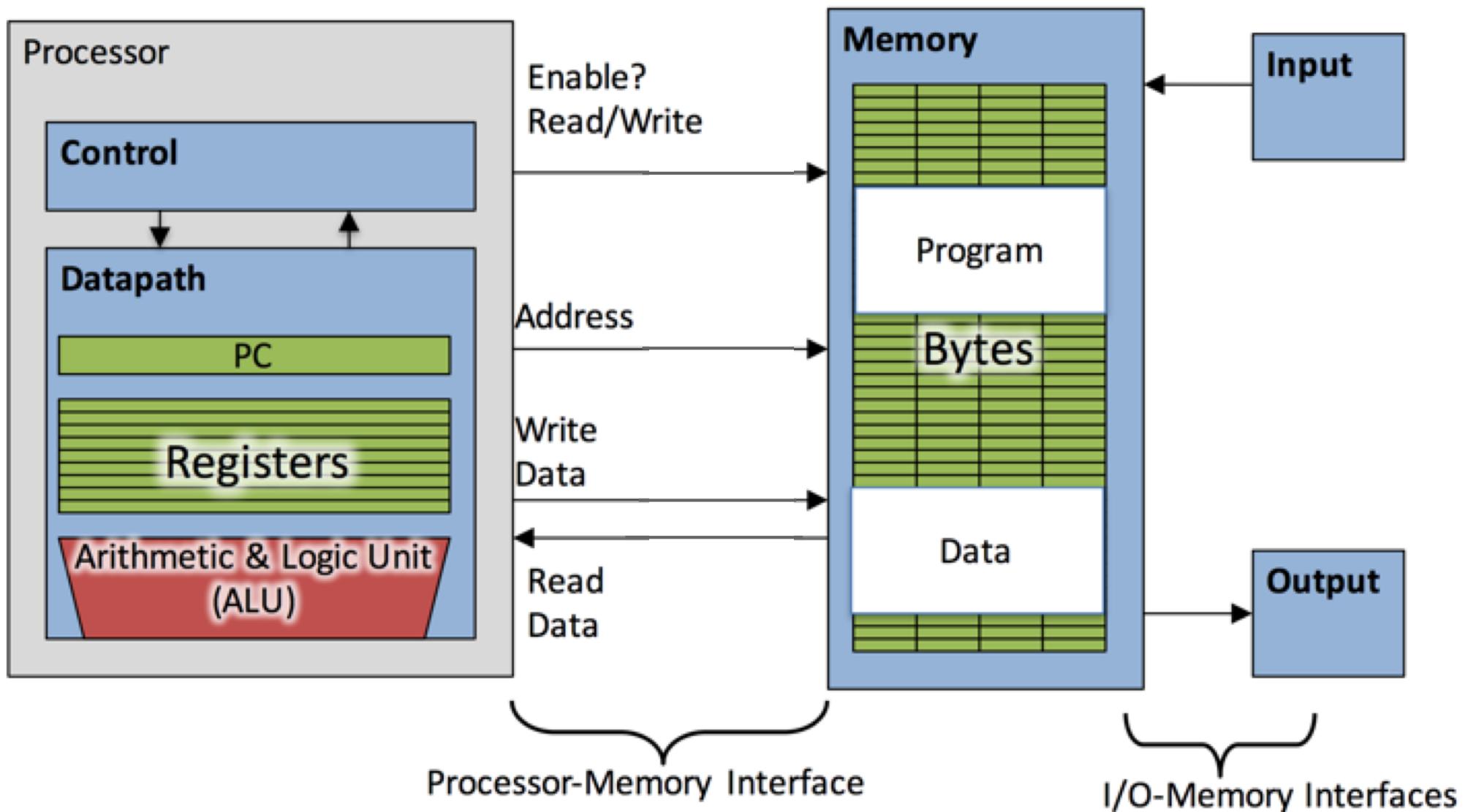


# Review

- Hazards reduce effectiveness of pipelining
  - Cause stalls/bubbles
- Structural Hazards
  - Conflict in use of datapath component
- Data Hazards
  - Need to wait for result of a previous instruction
- Control Hazards
  - Address of next instruction uncertain/unknown
  - Use branch prediction to avoid hazards

# Great Idea #3: Principle of Locality/ Memory Hierarchy





# Agenda

- Memory Hierarchy Overview
- Fully Associative Caches
- Cache Reads and Writes

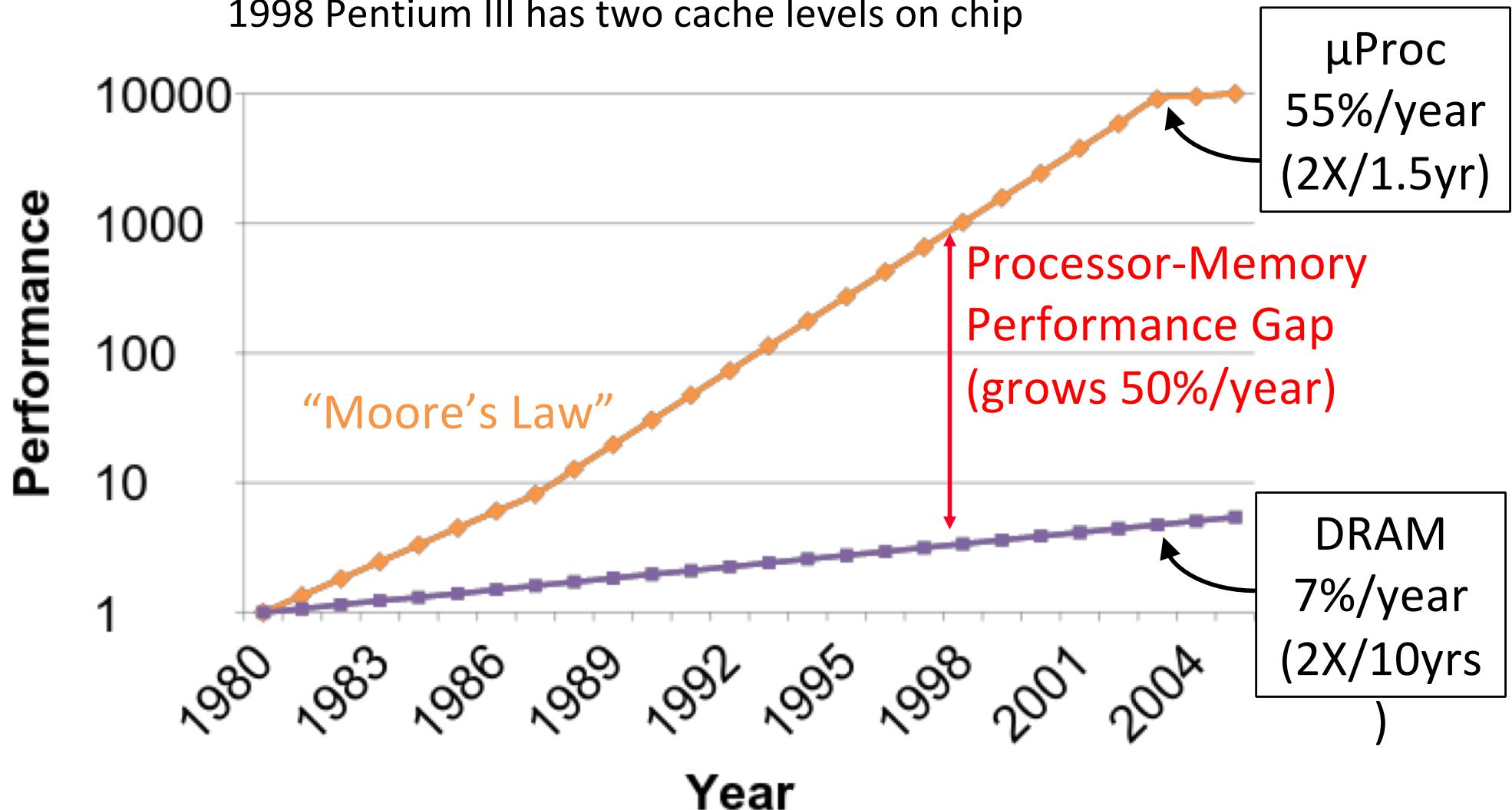
# Storage in a Computer

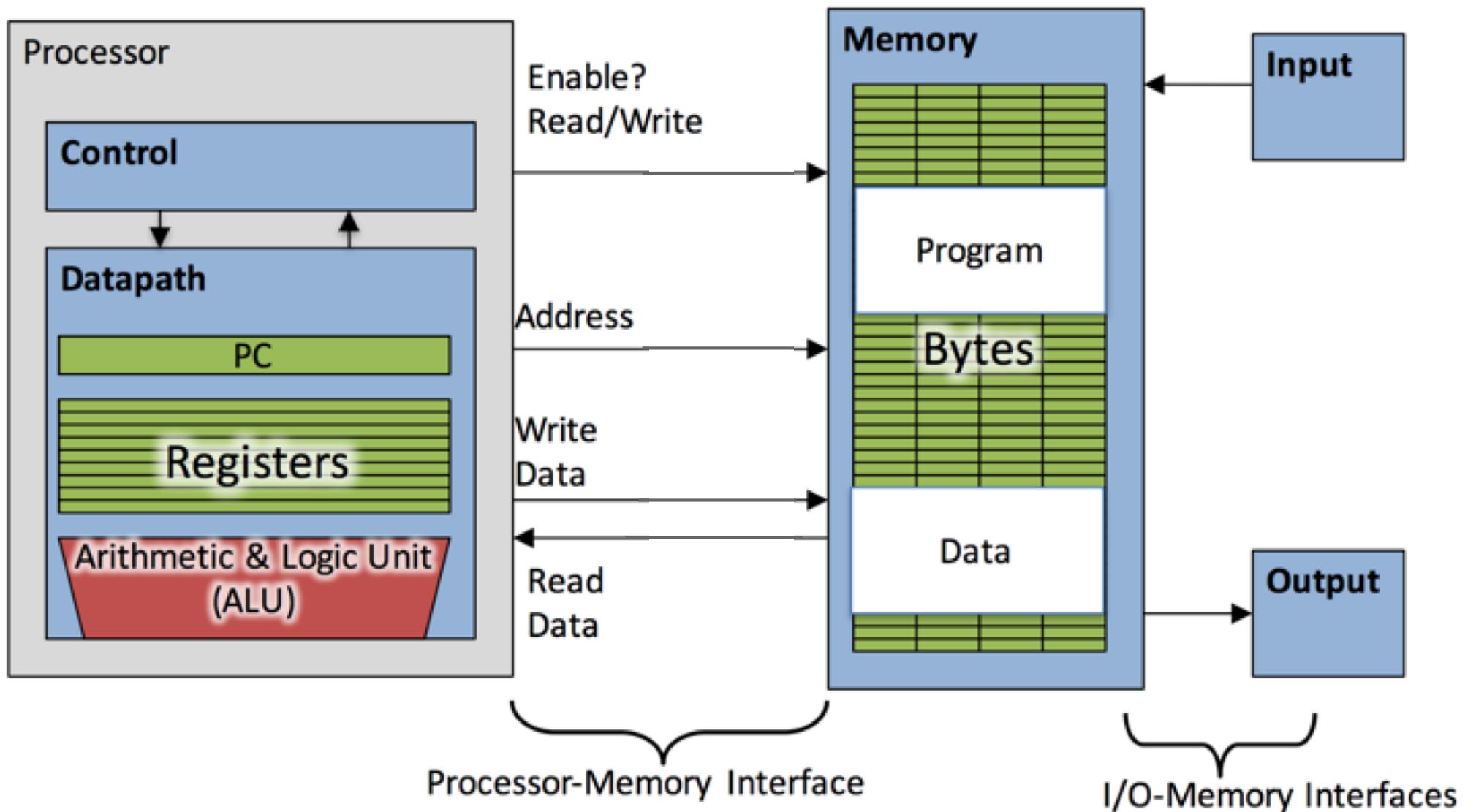
- Processor
  - Holds data in register files (~ 100 B)
  - Registers accessed on sub-nanosecond timescale
- Memory (“main memory”)
  - More capacity than registers (~ GiB)
  - Access time ~ 50-100 ns
- **Hundreds of clock cycles per memory access?!**

# Processor-Memory Gap

1989 first Intel CPU with cache on chip

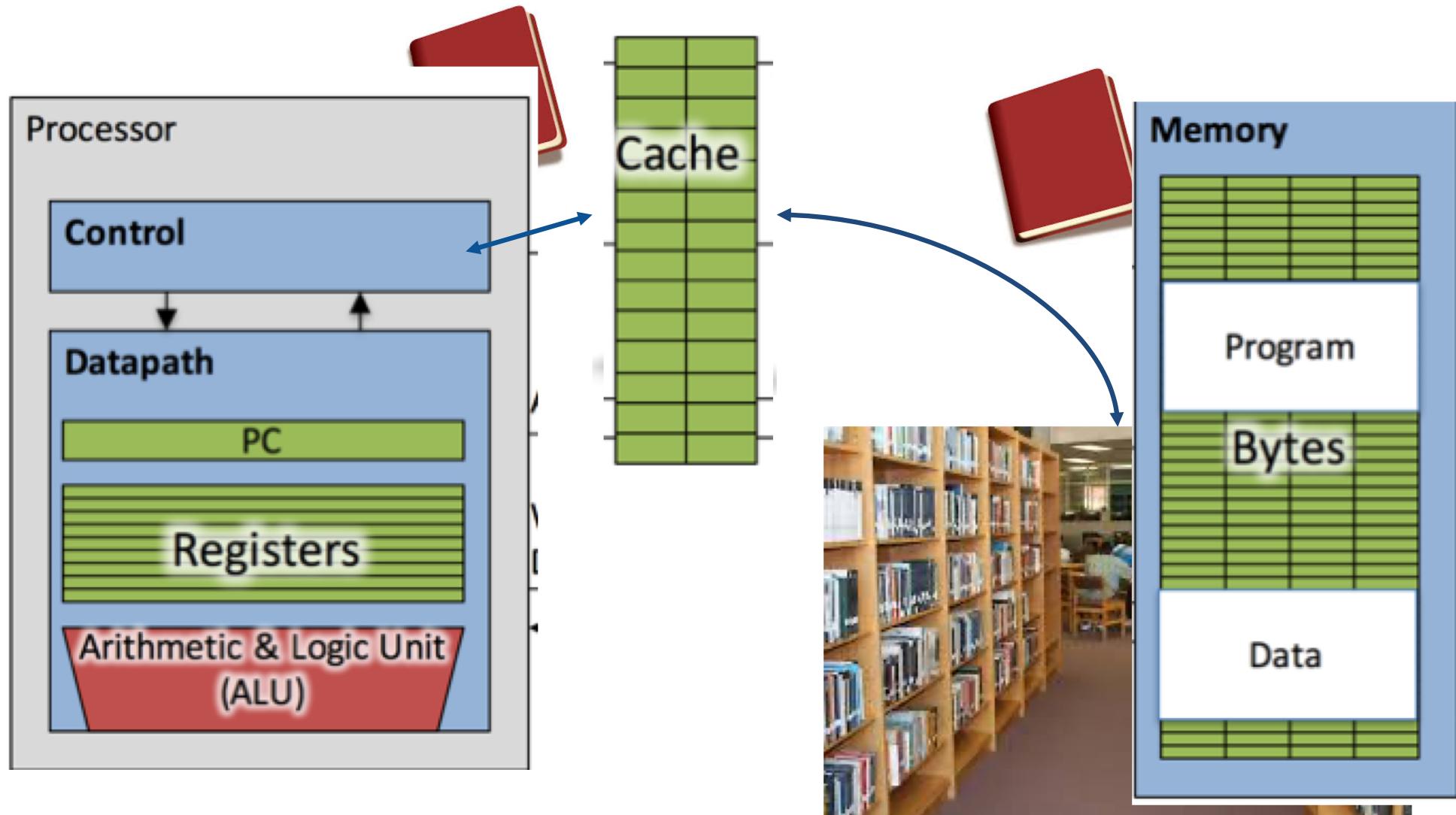
1998 Pentium III has two cache levels on chip





# Library Analogy

- Writing a report on a specific topic
  - e.g. history of the computer (your internet is out)
- While at library, take books from shelves and put them on shelf above your desk
- If need more, go get them and bring back to shelf
  - Don't return earlier books since might still need them
  - Limited space on shelf; which books do we keep?
- You hope these ~10 books on shelf enough to write report
  - Only 0.00001% of the books in libraries!



# Principle of Locality (1/3)

- *Principle of Locality:* Programs access only a small portion of the full address space at any instant of time
  - **Recall:** Address space holds both code and data
  - Loops and sequential instruction execution mean generally localized code access
  - Stack and Heap try to keep your data together
  - Arrays and structs naturally group data you would access together

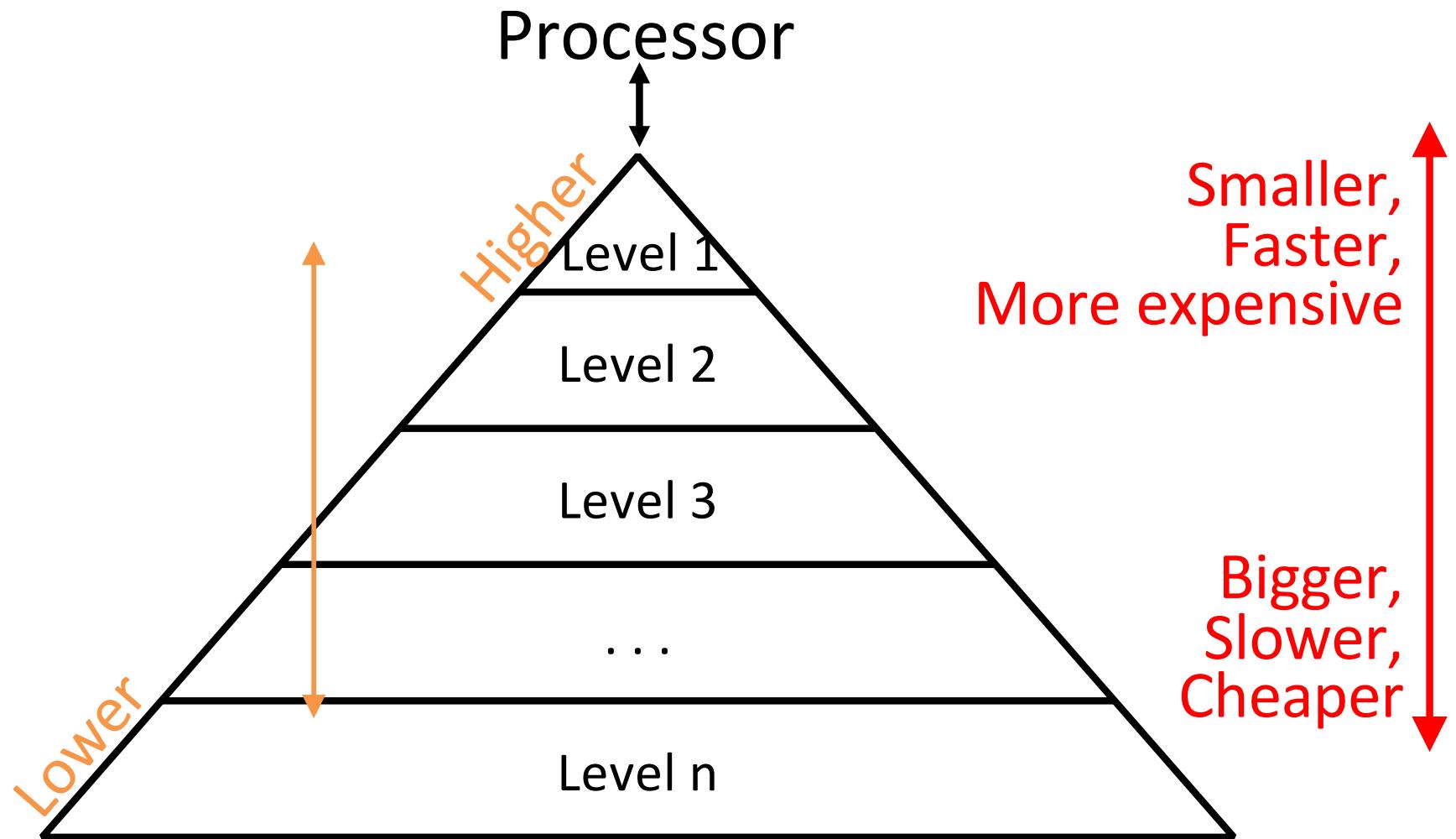
# Principle of Locality (2/3)

- *Temporal Locality* (locality in time)
  - Go back to the same book on desk multiple times
  - If a memory location is referenced then it will tend to be referenced again soon
- *Spatial Locality* (locality in space)
  - When go to shelves, grab many books on computers since related books are stored together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

# Principle of Locality (3/3)

- We exploit the principle of locality in hardware via a *memory hierarchy* where:
  - Levels closer to processor are faster (and more expensive per bit so smaller)
  - Levels farther from processor are larger (and less expensive per bit so slower)
- **Goal:** Create the illusion of memory being almost as fast as fastest memory and almost as large as biggest memory of the hierarchy

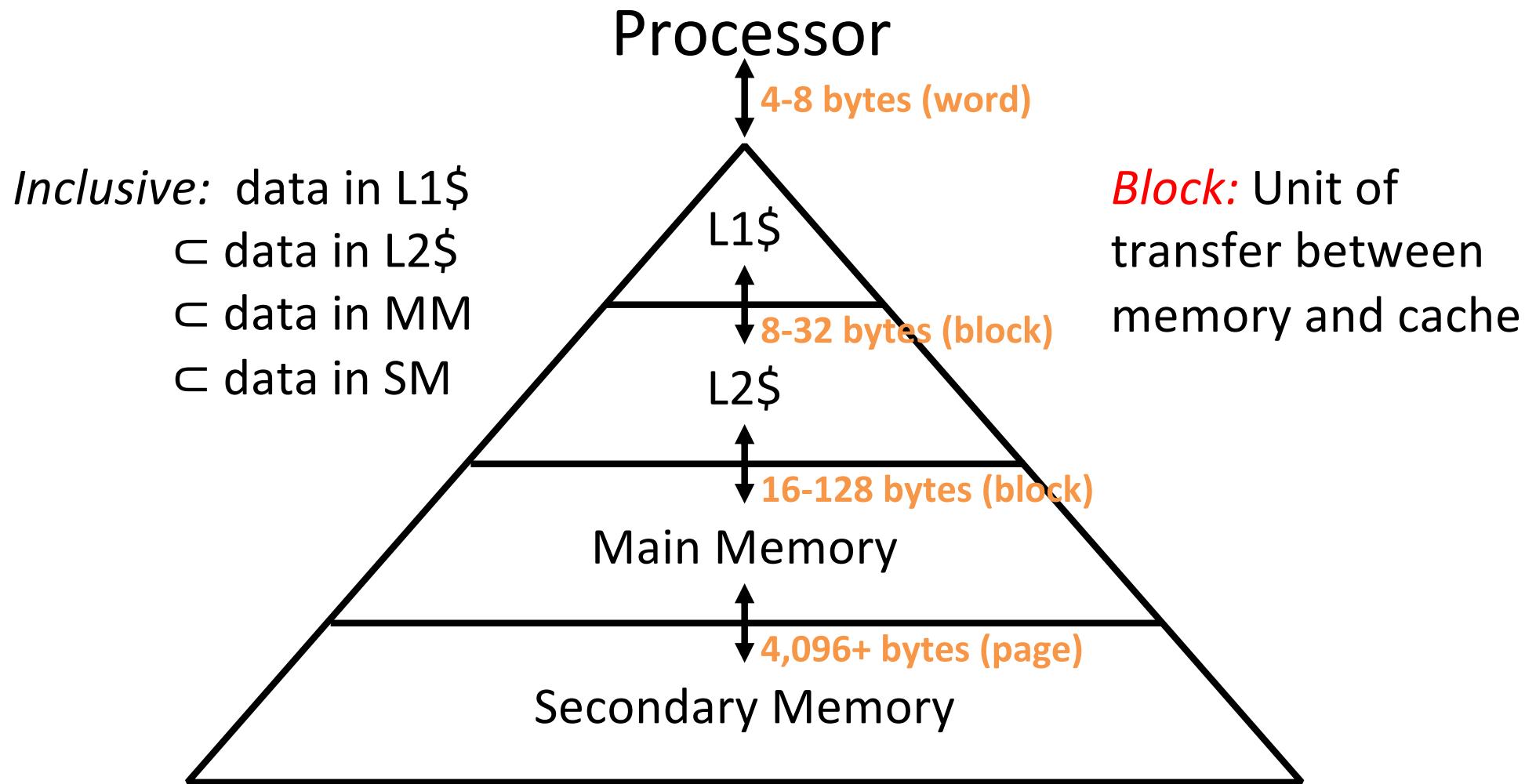
# Memory Hierarchy Schematic



# Cache Concept

- Memory **Cache**—holds a copy of a subset of main memory
  - We often use \$ (“cash”) to abbreviate cache (e.g. D\$ = Data Cache, L1\$ = Level 1 Cache)
- Modern processors have separate caches for instructions and data, as well as several levels of caches implemented in different sizes
- Implemented with same IC processing technology as CPU and integrated on-chip – faster but more expensive than main memory

# Memory Transfer in the Hierarchy



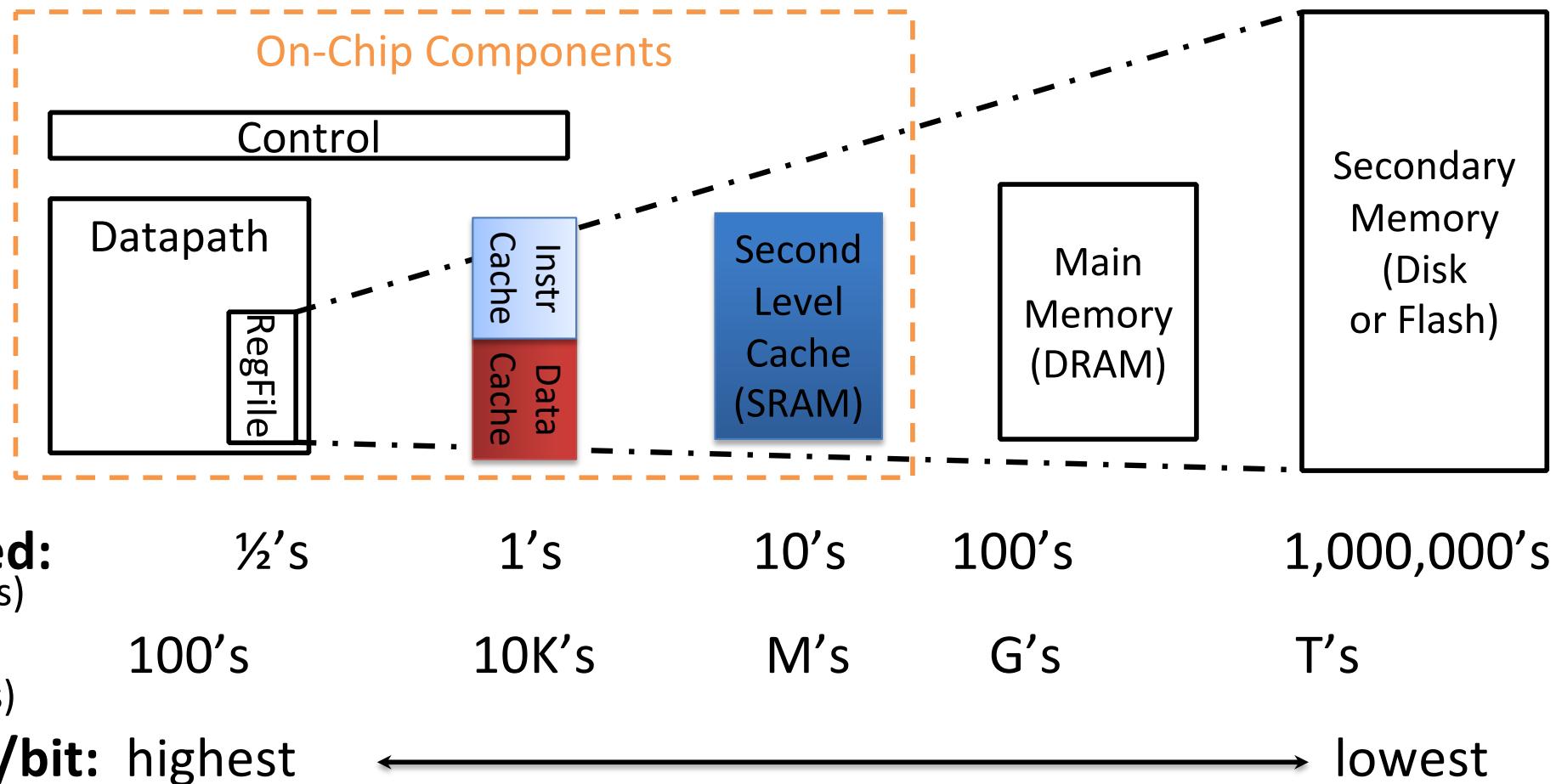
# Managing the Hierarchy

- registers  $\leftrightarrow$  memory
- cache  $\leftrightarrow$  main memory
- main memory  $\leftrightarrow$  disks (secondary storage)

...

here

# Typical Memory Hierarchy



# Memory Hierarchy Technologies

- Caches use static RAM (SRAM)
  - + Fast (typical access times of 0.5 to 2.5 ns)
  - + Higher power, expensive (\$2000 to \$4000 per GB in 2011)
  - + *Static*: content will last as long as power is on
- Main memory uses dynamic RAM (DRAM)
  - + Lower power, cheaper (\$20 to \$40 per GB in 2011)
  - + Slower (typical access times of 50 to 70 ns)
  - + *Dynamic*: needs to be “refreshed” regularly (~ every 8 ms)

# Review So Far

- **Goal:** present the programmer with  $\approx$  as much memory as the *largest* memory at  $\approx$  the speed of the *fastest* memory
- **Approach:** Memory Hierarchy
  - Successively higher levels contain “most used” data from lower levels
  - Exploits *temporal and spatial locality*
  - We will start by studying caches

# Agenda

- Memory Hierarchy Overview
- **Fully Associative Caches**
- Cache Reads and Writes

# Cache Management

- What is the overall organization of blocks we impose on our cache?
  - Where do we put a block of data from memory?
  - How do we know if a block is already in cache?
  - How do we quickly find a block when we need it?
  - When do we replace something in the cache?

# General Notes on Caches (1/4)

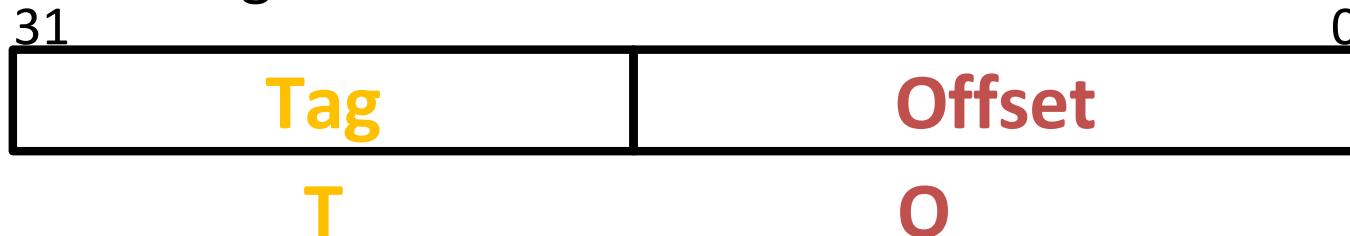
- **Recall:** Memory is *byte-addressed*
  - We haven't specified the size of our "blocks," but will usually be multiple of word size (32-bits)
    - How do we access individual words or bytes within a block? ← **OFFSET**
  - Cache is smaller than memory
    - Can't fit all blocks at once, so multiple blocks in memory must map to the same slot in cache
    - Need some way of identifying which memory block is currently in each cache slot
- INDEX**
- TAG**

# General Notes on Caches (2/4)

- **Recall:** hold subset of memory in a place that's faster to access
  - Return data to you when you request it
  - If cache doesn't have it, then fetches it for you
- Cache must be able to check/identify its current contents
- What does cache initially hold?
  - Garbage! Cache considered “cold”
  - Keep track with **Valid bit**

# General Notes on Caches (3/4)

- Effect of block size (K Bytes):
  - Spatial locality dictates our blocks consist of adjacent bytes, which differ in address by 1
  - **Offset field:** Lowest bits of memory address can be used to index to specific bytes within a block
    - Block size needs to be a power of two (in bytes)
    - One way to compute the offset field is to mask out the Tag and Index bits



# General Notes on Caches (4/4)

- Effect of cache size (C Bytes):
  - “Cache Size” refers to total stored *data*
  - Determines number of blocks the cache can hold ( $C/K$  blocks)
  - **Tag field:** Leftover upper bits of memory address determine *which portion of memory* the block came from (identifier)

## **Question:**

You are given a 256 B cache that holds 8 data blocks. We are working on a 10-bit byte-addressed machine.

Which statement is false?

- (A) Each data block holds 32 bytes of data
- (B) Our tag can be more than 5 bits
- (C) All of main memory could fit into four caches
- (D) Only one valid copy of a block can be in the cache at any point in time

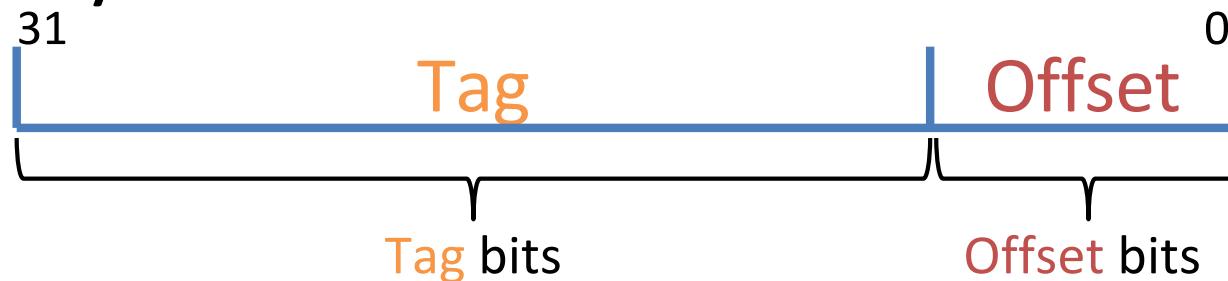
# Fully Associative Caches

- Each memory block can map *anywhere* in the cache (*fully associative*)
  - Most efficient use of space
  - Least efficient to check
- To check a fully associative cache:
  - 1) Look at ALL cache slots in parallel
  - 2) If Valid bit is 0, then ignore
  - 3) If Valid bit is 1 and **Tag** matches, then return that data

# Fully Associative Cache

## Address Breakdown

- Memory address fields:



- Meaning of the field sizes:
  - **Offset** bits  $\leftrightarrow 2^{\text{Offset}}$  bytes per block  
 $= 2^{\text{Offset}-2}$  words per block
  - **Tag** bits =  $A - \text{Offset}$ , where  $A = \#$  of address bits  
( $A = 32$  here)

## **Question:**

You are given a 32 bit byte addressed machine. The offset field is 5 bits, but you decide to use only 26 bits to represent the tag. Which of the following statements is true?

- (A) Each block can contain more data than if you used all 27 bits.**
- (B) More bits will need to be stored by the cache**
- (C) Fewer blocks can be stored in the cache than if you used all 27 bits.**
- (D) You cannot properly identify what is data block is in the cache.**

# Caching Terminology (1/2)

- When reading memory, 2 things can happen:
  - Cache hit:  
Cache holds a valid copy of the block, so return the desired data
  - Cache miss:  
Cache does not have desired block, so fetch from memory and put in empty (invalid) slot
    - If cache is full you must discard one valid block and replace it with desired data

# Block Replacement Policies

- Which block do you replace?
  - Use a *cache block replacement policy*
  - There are many (most are intuitively named), but we will just cover a few in this class
- Of note:
  - Random Replacement
  - Least Recently Used (LRU): requires some “management bits”

# Understanding LRU

- The BEST replacement policy we could have is to replace data we will use farthest in the future
  - Either never use again or use later than everything else
- Unfortunately this isn't possible
- Instead we approximate using LRU
  - Replace block we used least recently first in the hopes we will use it again later than all other blocks

# Caching Terminology (2/2)

- How effective is your cache?
  - Want to max cache hits and min cache misses
  - **Hit rate (HR):** Percentage of memory accesses in a program or set of instructions that result in a cache hit
  - **Miss rate (MR):** Like hit rate, but for cache misses  
$$MR = 1 - HR$$
- How fast is your cache?
  - **Hit time (HT):** Time to access cache (including **Tag** comparison)
  - **Miss penalty (MP):** Time to replace a block in the cache from a lower level in the memory hierarchy

# Fully Associative Cache Implementation

- What's actually in the cache?
  - Each cache slot contains the actual data block ( $8 \times K = 8 \times 2^{\text{Offset}}$  bits)
  - **Tag** field of address as identifier (**Tag** bits)
  - **Valid** bit (1 bit): Whether cache slot was filled in
  - Any necessary replacement management bits (“LRU bits” – variable # of bits stored in each row)
- Total bits in cache
  - = # slots  $\times$  (8 $\times$ K + **Tag** + 1 + ?)
  - = (C/K)  $\times$  (8 $\times$ 2<sup>Offset</sup> + **Tag** + 1 + ?) bits

# FA Cache Examples (1/4)

- Cache parameters:
  - Fully associative, address space of 64B, block size of 1 word, cache size of 4 words, LRU (2 bits)

- Address Breakdown:      Memory Addresses: 
  - 1 word = 4 bytes, so **Offset** =  $\log_2(4) = 2$
  - $A = \log_2(64) = 6$  bits, so **Tag** =  $6 - 2 = 4$

- Bits in cache
  - =  $(4/1) \times (8 \times 2^2 + 4 + 1 + 2) = 156$  bits

# FA Cache Examples (1/4)

- Cache parameters:
  - Fully associative, address space of 64B, block size of 1 word, cache size of 4 words, LRU (2 bits)
  - Offset – 2 bits, Tag – 4 bits

	Slot	V	Tag	Offset				Block Size	LRU Bits: 00: Used Last 11: Next to remove
				00	01	10	11	LRU	
	0	X	XXXX	0x??	0x??	0x??	0x??	01	
	1	X	XXXX	0x??	0x??	0x??	0x??	10	
	2	X	XXXX	0x??	0x??	0x??	0x??	00	
	3	X	XXXX	0x??	0x??	0x??	0x??	11	Cache Size

- 39 bits per slot, 156 bits to implement with LRU

# FA Cache Examples (2/4)

1) Consider the sequence of memory address accesses

Starting with a cold cache:

0 2 4 10 20 16 0 2

**000000**

**0 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000100**

**4 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0001	M[4]	M[5]	M[6]	M[7]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000010**

**2 hit**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**001010**

**10 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0001	M[4]	M[5]	M[6]	M[7]
1	0010	M[8]	M[9]	M[10]	M[11]
0	0000	0x??	0x??	0x??	0x??

# FA Cache Examples (2/4)

1) Consider the sequence of memory address accesses

Starting with a cold cache:

**010100**  
**20 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0001	M[4]	M[5]	M[6]	M[7]
1	0010	M[8]	M[9]	M[10]	M[11]
1	0101	M[20]	M[21]	M[22]	M[23]

**000000**  
**0 miss**

1	0100	M[16]	M[17]	M[18]	M[19]
1	0001	M[4]	M[5]	M[6]	M[7]
1	0010	M[8]	M[9]	M[10]	M[11]
1	0101	M[20]	M[21]	M[22]	M[23]

0    2    4    10    20    16    0    2  
M   H   M   M   M   M   M   H

**010000**  
**16 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0001	M[4]	M[5]	M[6]	M[7]
1	0010	M[8]	M[9]	M[10]	M[11]
1	0101	M[20]	M[21]	M[22]	M[23]

**000010**  
**2 hit**

1	0100	M[16]	M[17]	M[18]	M[19]
1	0000	M[0]	M[1]	M[2]	M[3]
1	0010	M[8]	M[9]	M[10]	M[11]
1	0101	M[20]	M[21]	M[22]	M[23]

- 8 requests, 6 misses – HR of 25%

# FA Cache Examples (3/4)

## 2) Same requests, but reordered

Starting with a cold cache:

0    2    2    0    16    20    10    4

**000000**  
**0 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000010**  
**2 hit**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000010**  
**2 hit**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000000**  
**0 hit**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

# FA Cache Examples (3/4)

## 2) Same requests, but reordered

Starting with a cold cache:

0    2    2    0    16    20    10    4  
M    H    H    H

**010000**  
**16 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0100	M[16]	M[17]	M[18]	M[19]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**001010**  
**10 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0100	M[16]	M[17]	M[18]	M[19]
1	0101	M[20]	M[21]	M[22]	M[23]
1	0010	M[8]	M[9]	M[10]	M[11]

**010100**  
**20 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0100	M[16]	M[17]	M[18]	M[19]
1	0101	M[20]	M[21]	M[22]	M[23]
0	0000	0x??	0x??	0x??	0x??

**000100**  
**4 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0100	M[16]	M[17]	M[18]	M[19]
1	0101	M[20]	M[21]	M[22]	M[23]
1	0010	M[8]	M[9]	M[10]	M[11]

- 8 requests, 5 misses – ordering matters!

# FA Cache Examples (4/4)

## 3) Original sequence, but double block size

Starting with a cold cache: 0 2 4 10 20 16 0 2

000000										
0	miss	1 000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
		0 000	0x??							

000010										
2	hit	1 000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
		0 000	0x??							

000100										
4	hit	1 000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
		0 000	0x??							

001010										
10	miss	1 000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
		1 001	M[8]	M[9]	M[10]	M[11]	M[12]	M[13]	M[14]	M[15]

# FA Cache Examples (4/4)

## 3) Original sequence, but double block size

		Starting with a cold cache:		0	2	4	8	20	16	0	2
<b>010100</b>				M	H	H	M				
<b>20</b>		1   000   M[0]   M[1]   M[2]   M[3]   M[4]   M[5]   M[6]   M[7]									
<b>miss</b>		1   001   M[8]   M[9]   M[10]   M[11]   M[12]   M[13]   M[14]   M[15]									
<b>010000</b>											
<b>16</b>		1   010   M[16]   M[17]   M[18]   M[19]   M[20]   M[21]   M[22]   M[23]									
<b>hit</b>		1   001   M[8]   M[9]   M[10]   M[11]   M[12]   M[13]   M[14]   M[15]									
<b>000000</b>											
<b>0</b>		1   010   M[16]   M[17]   M[18]   M[19]   M[20]   M[21]   M[22]   M[23]									
<b>miss</b>		1   001   M[8]   M[9]   M[10]   M[11]   M[12]   M[13]   M[14]   M[15]									
<b>000010</b>											
<b>2</b>		1   010   M[16]   M[17]   M[18]   M[19]   M[20]   M[21]   M[22]   M[23]									
<b>hit</b>		1   000   M[0]   M[1]   M[2]   M[3]   M[4]   M[5]   M[6]   M[7]									

- 8 requests, 4 misses – cache parameters matter!

## Question:

Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

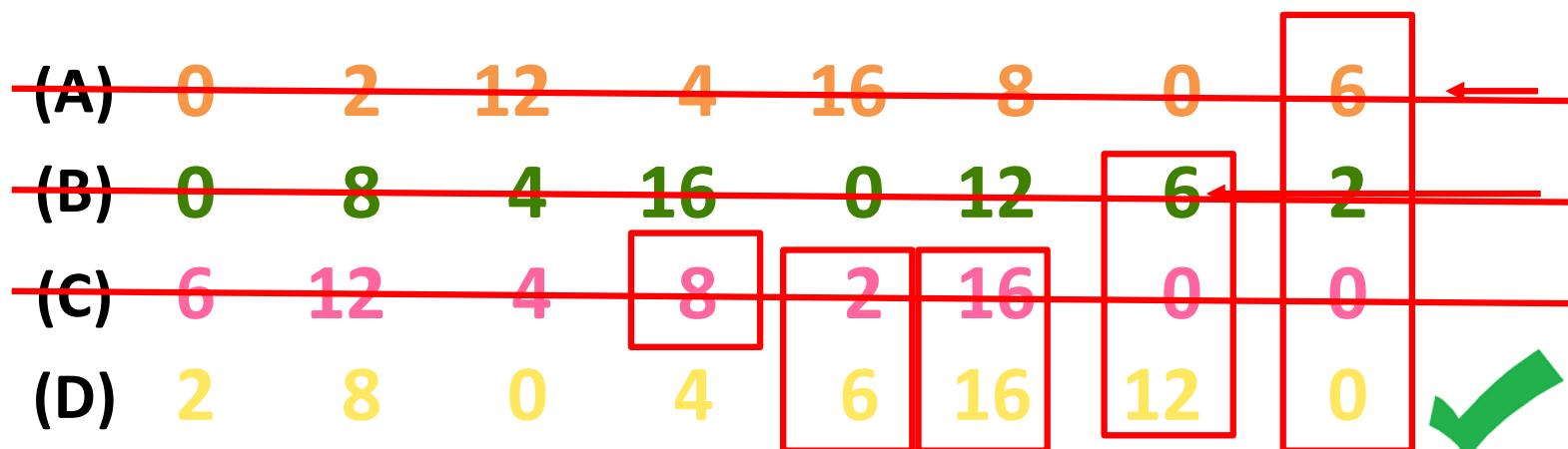
							LRU
0	1	0000	M[0]	M[1]	M[2]	M[3]	XX
1	1	0011	M[12]	M[13]	M[14]	M[15]	XX
2	1	0001	M[4]	M[5]	M[6]	M[7]	11
3	1	0100	M[16]	M[17]	M[18]	M[19]	XX

- (A) 0      2      12      4      16      8      0      6
- (B) 0      8      4      16      0      12      6      2
- (C) 6      12      4      8      2      16      0      0
- (D) 2      8      0      4      6      16      12      0

## Question:

Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

							LRU
0	1	0000	M[0]	M[1]	M[2]	M[3]	XX
1	1	0011	M[12]	M[13]	M[14]	M[15]	XX
2	1	0001	M[4]	M[5]	M[6]	M[7]	11
3	1	0100	M[16]	M[17]	M[18]	M[19]	XX



## Question:

Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

							LRU
0	1	0000	M[0]	M[1]	M[2]	M[3]	XX
1	1	0011	M[12]	M[13]	M[14]	M[15]	XX
2	1	0001	M[4]	M[5]	M[6]	M[7]	11
3	1	0100	M[16]	M[17]	M[18]	M[19]	XX

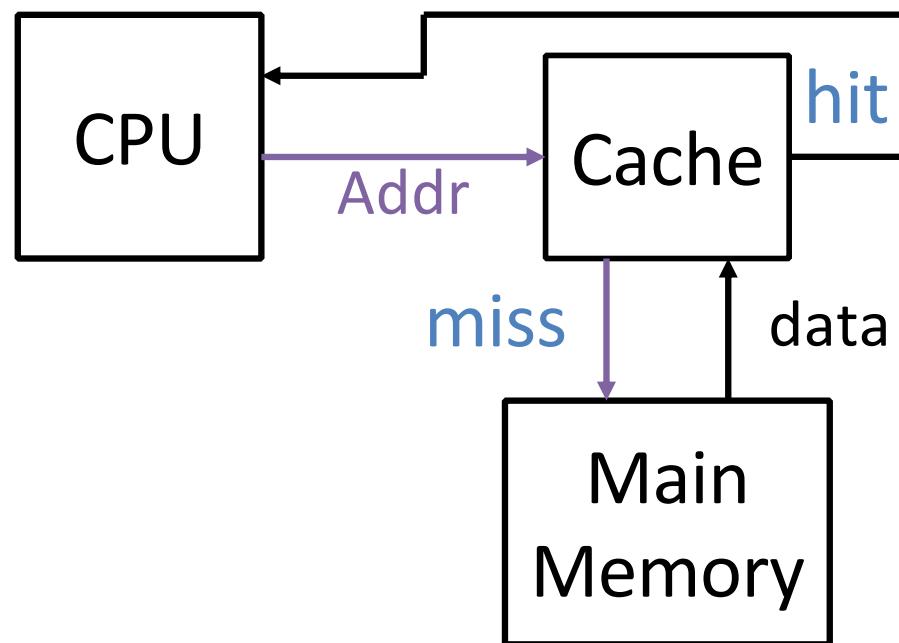
- (A) 0      2      12      4      16      8      0      6
- (B) 0      8      4      16      0      12      6      2
- (C) 6      12      4      8      2      16      0      0
- (D) 2      8      0      4      6      16      12      0

# Agenda

- Memory Hierarchy Overview
- Fully Associative Caches
- Cache Reads and Writes

# Memory Accesses

- The picture so far:



- Cache is separate from memory
  - Possible to hold different data?

# Cache Reads and Writes

- Want to handle reads and writes quickly while maintaining consistency between cache and memory (i.e. both know about all updates)
  - Policies for cache hits and misses are independent
- Here we assume the use of separate instruction and data caches (I\$ and D\$)
  - Read from both
  - Write only to D\$ (assume no self-modifying code)

# Handling Cache Hits

- Read hits (I\$ and D\$)
  - Fastest possible scenario, so want more of these
- Write hits (D\$)
  - 1) **Write-Through Policy:** Always write data to cache and to memory (*through* cache)
    - Forces cache and memory to always be consistent
    - Slow! (every memory access is long)
    - Include a *Write Buffer* that updates memory in parallel with processor

Assume present in all schemes  
when writing to memory

# Handling Cache Hits

- Read hits (I\$ and D\$)
  - Fastest possible scenario, so want more of these
- Write hits (D\$)
  - 2) **Write-Back Policy:** Write data only to cache, then update memory when block is removed
    - Allows cache and memory to be inconsistent
    - Multiple writes collected in cache; single write to memory per block
    - **Dirty bit:** Extra bit per cache row that is set if block was written to (is “dirty”) and needs to be written back

# Handling Cache Misses

- Miss penalty grows as block size does
- Read misses (I\$ and D\$)
  - Stall execution, fetch block from memory, put in cache, send requested data to processor, resume
- Write misses (D\$)
  - Always have to update block from memory
  - We have to make a choice:
    - Carry the updated block into cache or not?

# Write Allocate

- *Write Allocate* policy: when we bring the block into the cache after a write miss
- *No Write Allocate* policy: only change main memory after a write miss
  - Write allocate almost always paired with write-back
    - Eg: Accessing same address many times -> cache it
  - No write allocate typically paired with write-through
    - Eg: Infrequent/random writes -> don't bother caching it

# Updated Cache Picture

- Fully associative, write through
  - Same as previously shown
- Fully associative, write back

		V	D	Tag	00	01	10	11	LRU
Slot	0	X	X	XXXX	0x??	0x??	0x??	0x??	XX
	1	X	X	XXXX	0x??	0x??	0x??	0x??	XX
	2	X	X	XXXX	0x??	0x??	0x??	0x??	XX
	3	X	X	XXXX	0x??	0x??	0x??	0x??	XX

- Write miss procedure (write allocate or not)  
only affects **behavior**, not design

# Summary

- Memory hierarchy exploits principle of locality to deliver lots of memory at fast speeds
- Fully Associative Cache: Every block in memory maps to any cache slot
  - **Offset** to determine which byte within block
  - **Tag** to identify if it's the block you want
- Replacement policies: random and LRU
- Cache params: block size (K), cache size (C)
- Cache write policies:
  - Write-back (need dirty bit) and write-through