CS 61C: Great Ideas in Computer Architecture

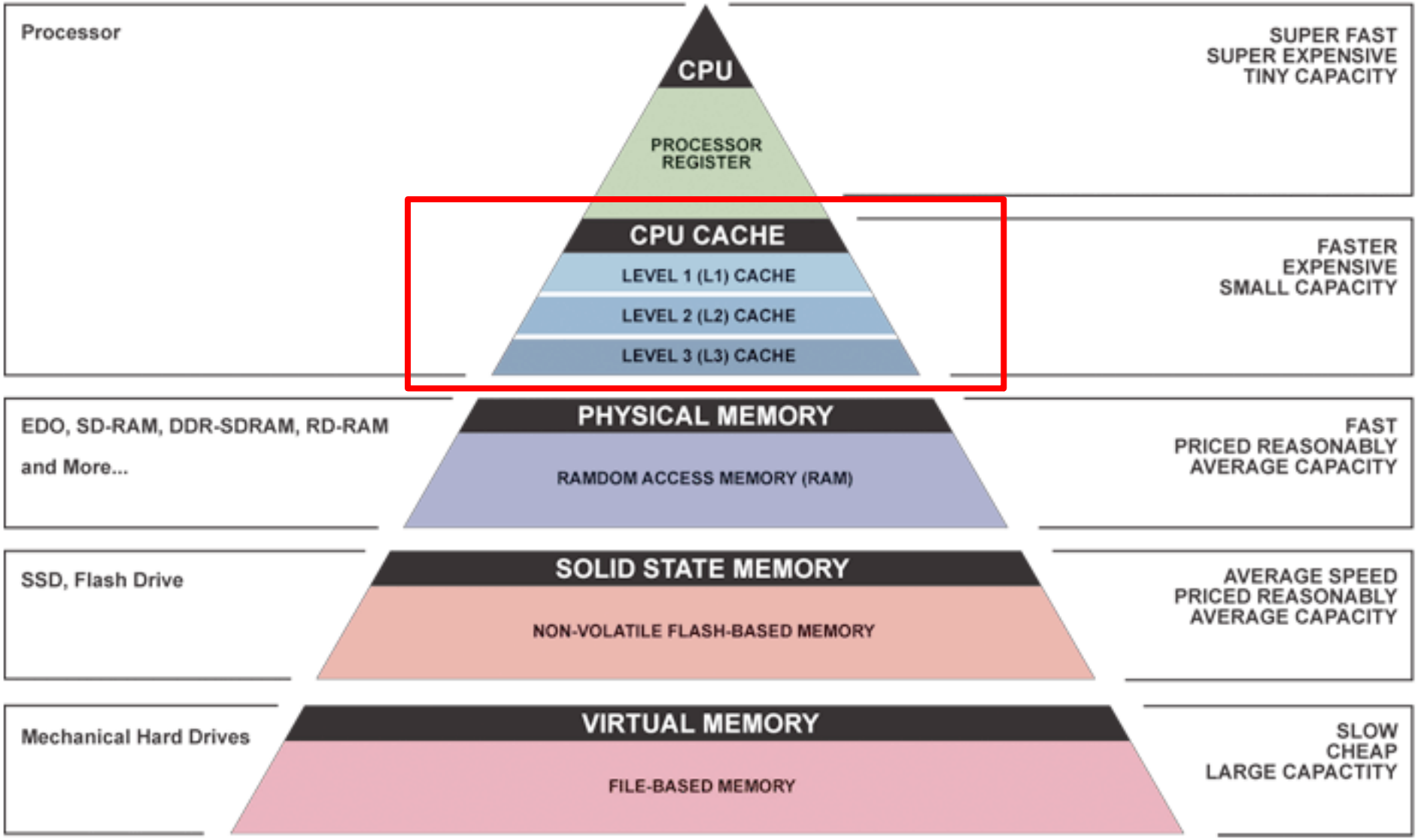
The Memory Hierarchy, Fully Associative Caches

Instructor: Alan Christopher

Review of Last Lecture

- Floating point (single and double precision) approximates real numbers
  - Exponent field uses *biased notation*
  - Special cases: 0, ±∞, NaN, denorm
  - High precision for small numbers
  - Low precision for large numbers
- Performance measured in *latency* or *bandwidth*
- Latency measurement:
  - $\text{CPU Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Cycle Time}$
  - Affected by different components of the computer

Great Idea #3: Principle of Locality/ Memory Hierarchy



Agenda

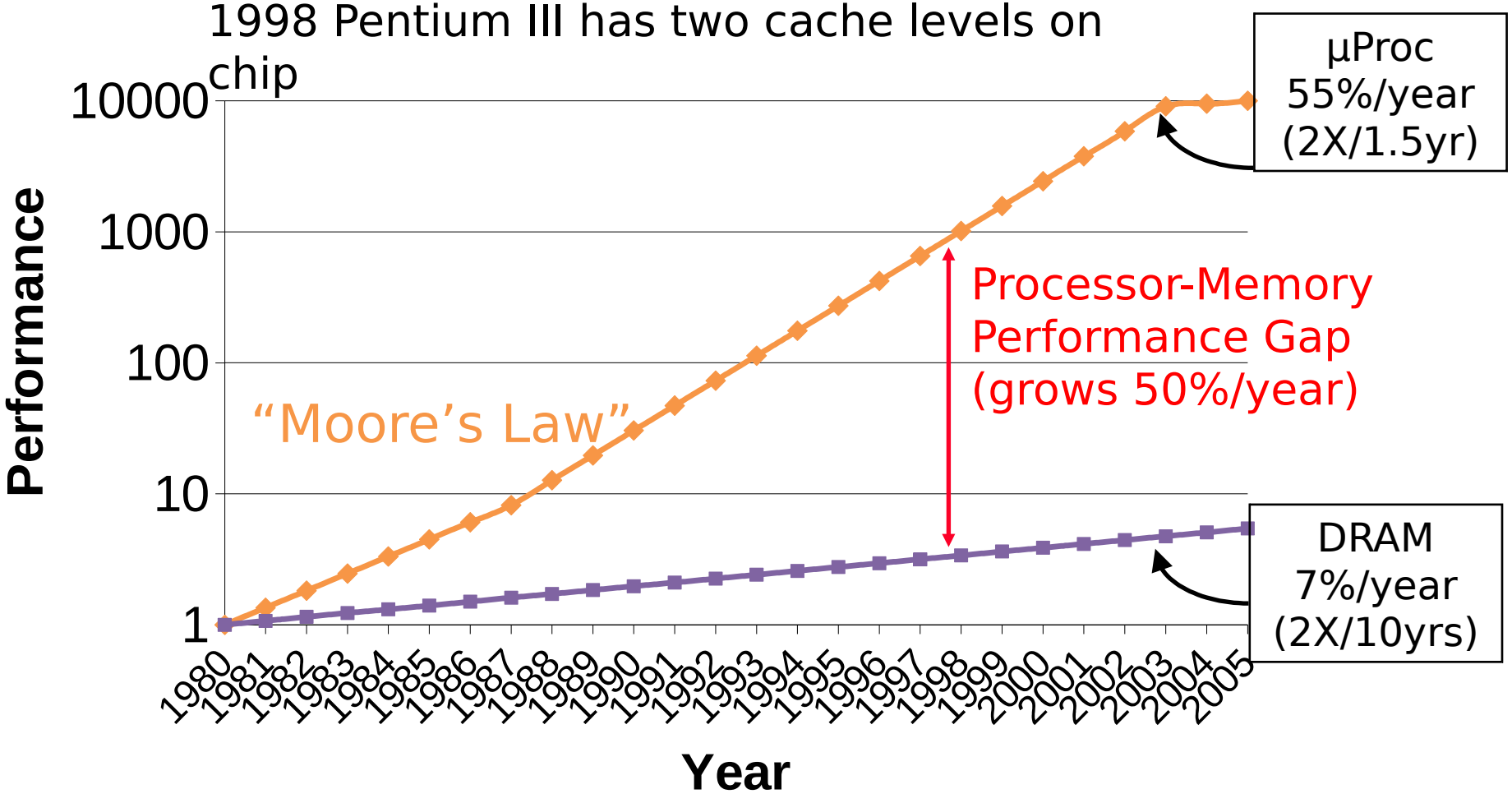
- Memory Hierarchy Overview
- Administrivia
- Fully Associative Caches
- Cache Reads and Writes

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 keep them on desk
- If need more, go get them and bring back to desk
  - Don’t return earlier books since might still need them
  - Limited space on desk; which books do we keep?
- You hope these ~10 books on desk enough to write report
  - Only 0.00001% of the books in UC Berkeley 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

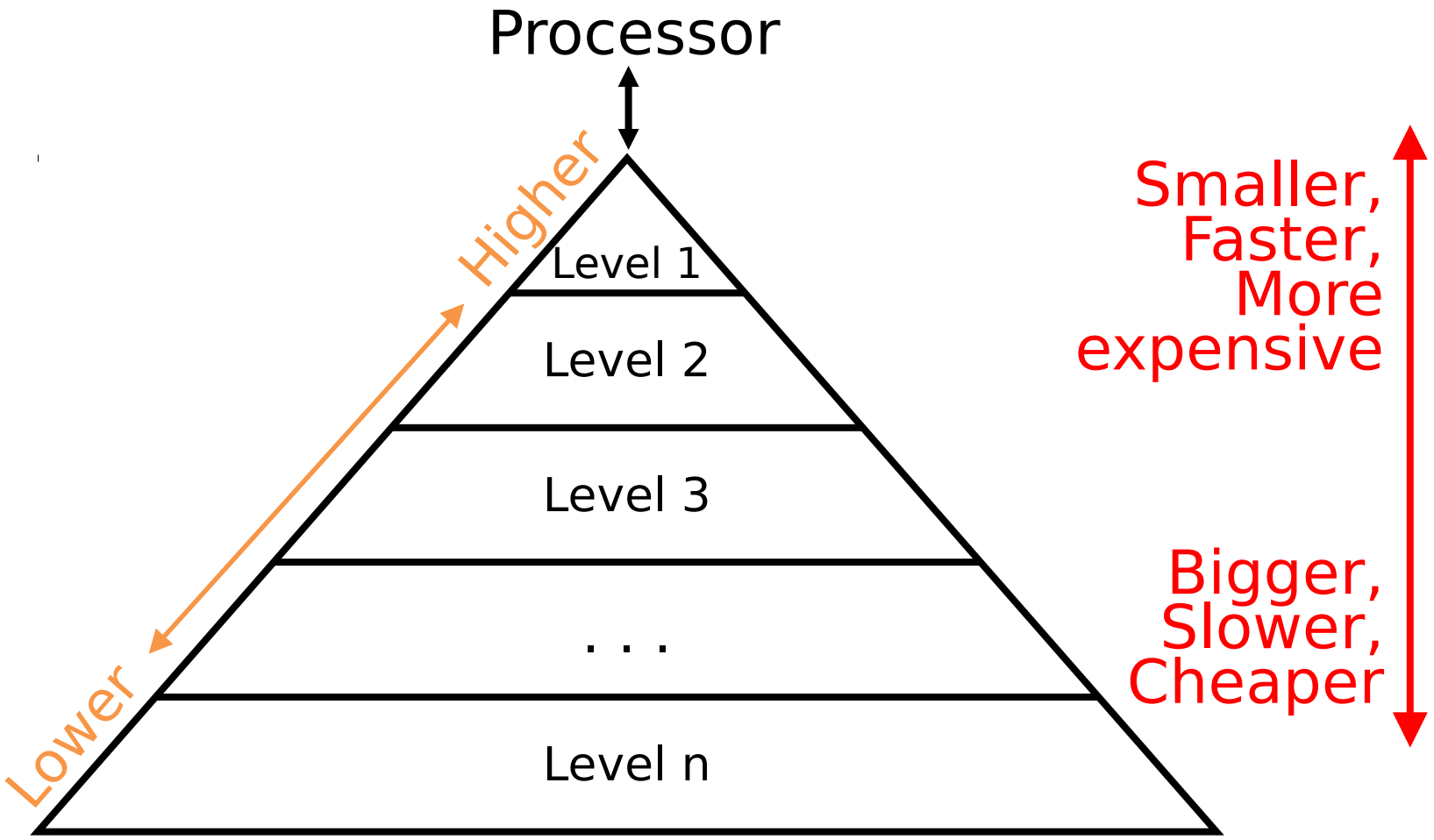
Principle of Locality (2/3)

- *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 as large as biggest memory of the hierarchy

Memory Hierarchy Schematic



Cache Concept

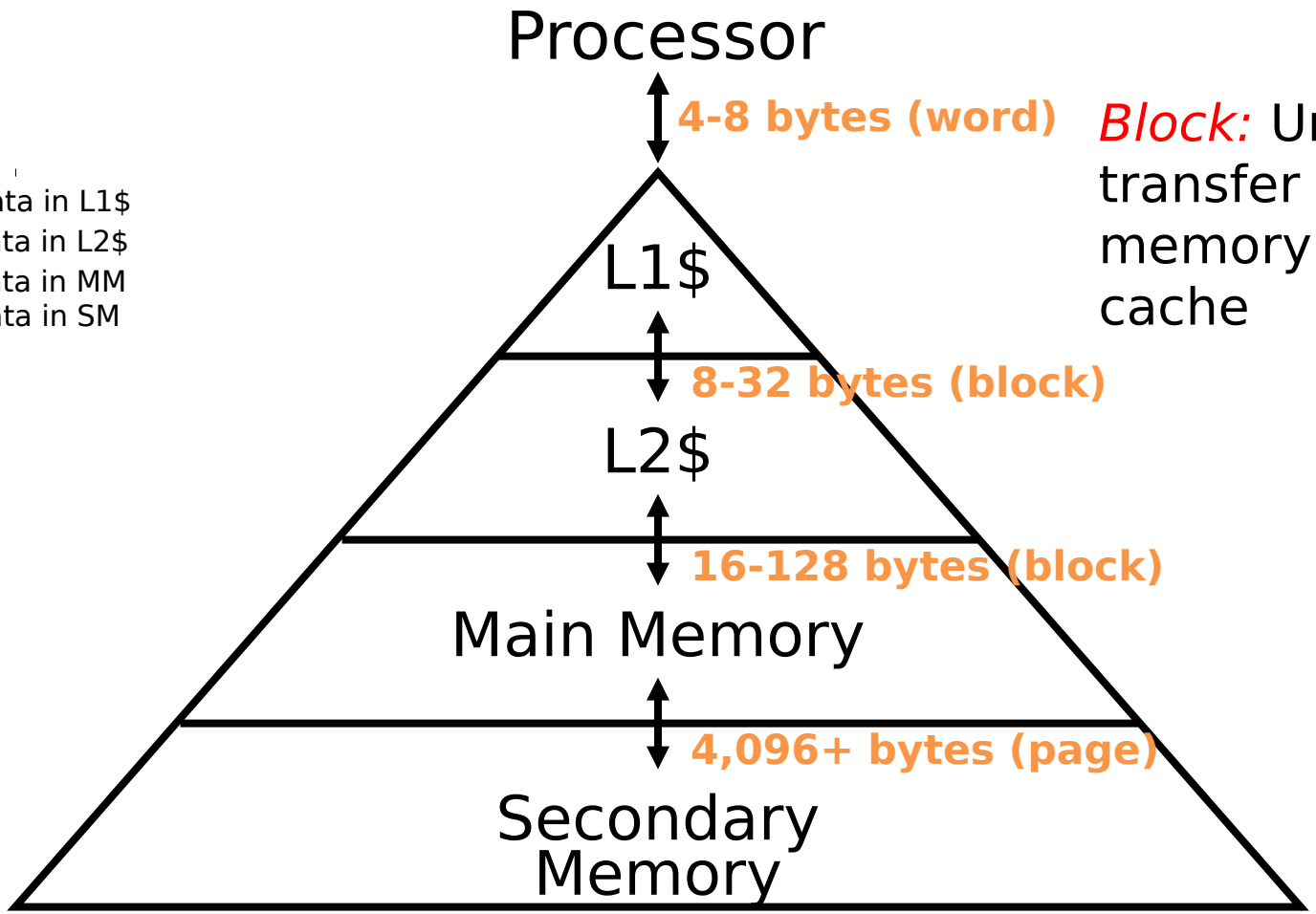
- Introduce intermediate hierarchy level: memory *cache*, which holds a copy of a subset of main memory
  - As a pun, often use $ (“cash”) to abbreviate cache (e.g. D$ = Data Cache, L1$ = Level 1 Cache)
- Modern processors have separate caches for instructions and data, as well as several levels of caches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

Memory Hierarchy Technologies

- Caches use static RAM (SRAM)
  - Fast (typical access times of 0.5 to 2.5 ns)
  - Low density (6 transistor cells), 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)
  - High density (1 transistor cells), 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)

Memory Transfer in the Hierarchy

Inclusive: data in L1$
⊂ data in L2$
⊂ data in MM
⊂ data in SM



Block: Unit of transfer between memory and cache

The Cache Block

- The *Cache Block* is the fundamental unit of memory that caches work with
  - Always retrieve one cache block when grabbing values from lower levels
  - Store data together in blocks
  - Evict blocks at a time (when necessary)
- Only access data at finer granularity when serving the level above

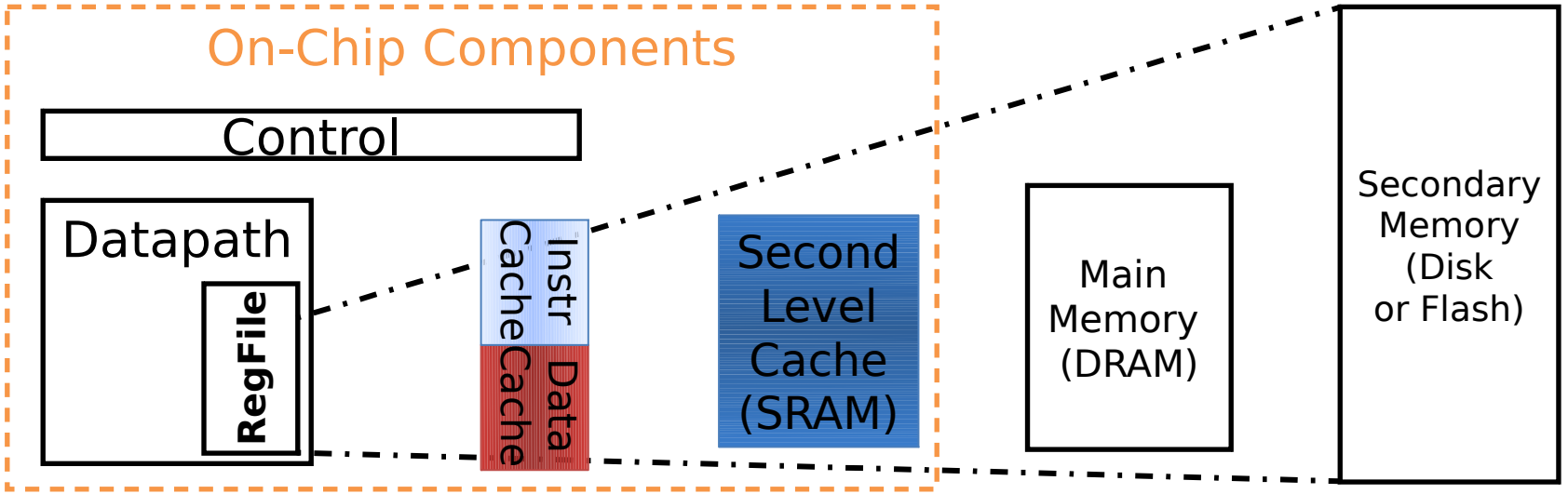
Managing the Hierarchy

- registers ↔ memory
  - By compiler (or assembly level programmer)
- cache ↔ main memory
  - By the cache controller hardware

We are here

- main memory ↔ disks (secondary storage)
  - By the OS (virtual memory, which is a later topic)
  - Virtual to physical address mapping assisted by the hardware (TLB)
  - By the programmer (files)

Typical Memory Hierarchy



Speed: (cycles)	½'s	1's	10's	100's	1,000,000's
Size: (bytes)	100's	10K's	M's	G's	T's
Cost/bit:	highest	←		→	lowest

Review So Far

- **Goal:** present the programmer with as much memory as the *largest* memory at ≈ the speed of the *fastest* memory
- **Approach:** Memory Hierarchy
  - Successively higher levels contain "most used" data from lower levels
  - Exploits *temporal and spatial locality*
  - Caches fill in the space between CPU and DRAM

Agenda

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

Administrivia

- Project 1 check-in (T hours so far)
  - (blue) 0 <= T < 6
  - (green) 6 <= T < 12
  - (purple) 12 <= T < 18
  - (yellow) 18 <= T

Administrivia

- Project 1 check-in – which parts have you finished?
  - Test suite?
  - Lexer?
  - Parser?
  - Code Generator?
  - Debugging?

Agenda

- Memory Hierarchy Overview
- Administrivia
- 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 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

INDEX

  - Need some way of identifying which memory block is currently in each cache slot

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)
    - (address) modulo (# of bytes in a block)

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)

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:
  - Look at ALL cache slots in parallel
  - If Valid bit is 0, then ignore
  - If Valid bit is 1 and Tag matches, then return that data

Block Replacement Policies

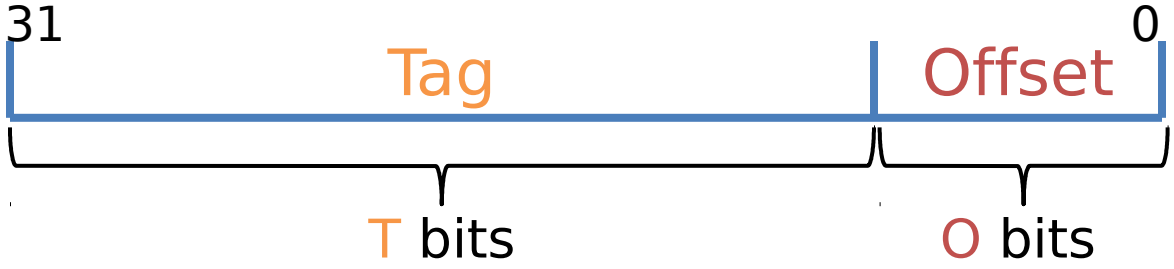
- Which block do you replace?
  - Use a *cache block replacement policy*
  - There are many (most are intuitively named), but we will just cover a few in this class

http://en.wikipedia.org/wiki/Cache_algorithms#Examples

- Of note:
  - Random Replacement
  - Least Recently Used (LRU): requires some "management bits"

Fully Associative Cache Address Breakdown

- Memory address fields:



- Meaning of the field sizes:
  - O bits ↔ 2^O bytes/block = 2^{O-2} words/block
  - T bits = A – O, where A = # of address bits (A = 32 here)

Caching Terminology (1/2)

- When reading memory, 3 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
  - Cache miss with block replacement:
    Cache does not have desired block and is full, so discard one and replace it with desired data

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 ($8 \times K = 8 \times 2^o$ bits)
  - Tag field of address as identifier (T bits)
  - Valid bit (1 bit)
  - Any necessary replacement management bits ("LRU bits" – variable # of bits)
- Total bits in cache

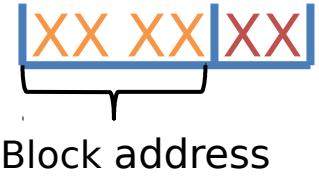
  $= \#\text{ slots} \times (8 \times K + T + 1) + ?$

  $= (C/K) \times (8 \times 2^o + T + 1) + ?$ 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 (5 bits)
- Address Breakdown:
  - 1 word = 4 bytes, so O = log2(4) = 2
  - A = log2(64) = 6 bits, so T = 6 – 2 = 4

Memory Addresses:

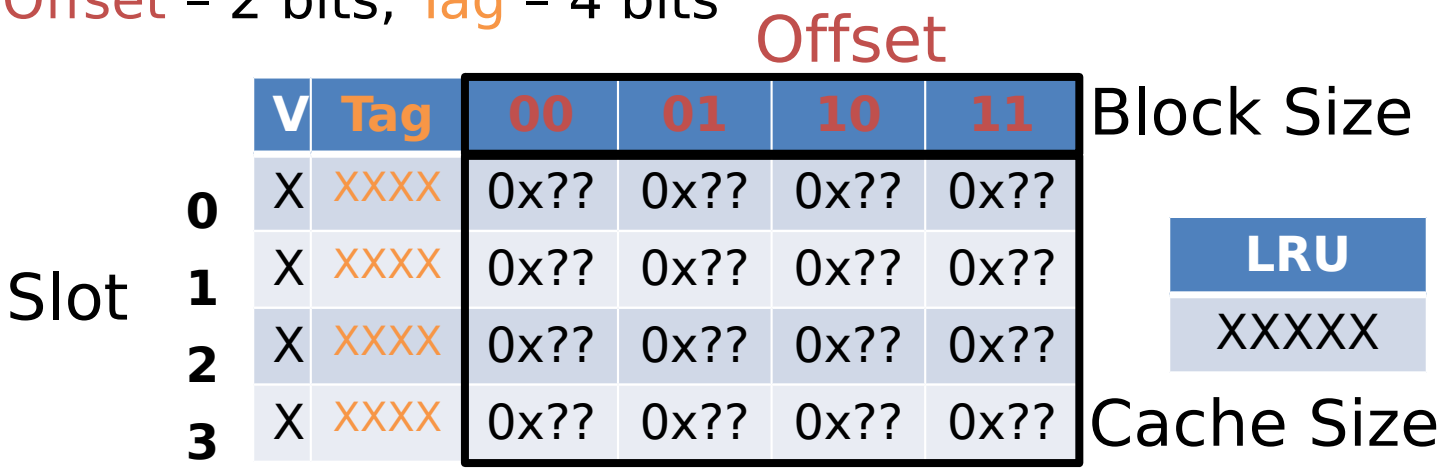


- Bits in cache

$$= (4/1) \times (8 \times 2^2 + 4 + 1) + 5 = 153 \text{ bits}$$

FA Cache Examples (1/4)

- Cache parameters:
  - Fully associative, address space of 64B, block size of 1 word, cache size of 4 words, LRU (5 bits)
  - Offset – 2 bits, Tag – 4 bits



- 37 bits per slot, 153 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 8 20 16 0 2

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??

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??

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??

8 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:

0	2	4	8	20	16	0	2
M	H	M	M				

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]

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]

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]

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 8 4

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??

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??

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??

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	8	4
M	H	H	H				

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??

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??

8 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]

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 8 20 16 0 2

0 miss

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

2 hit

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

4 hit

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

8 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
M	H	H	M				

20 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]

16 hit

1	010	M[16]	M[17]	M[18]	M[19]	M[20]	M[21]	M[22]	M[23]
1	001	M[8]	M[9]	M[10]	M[11]	M[12]	M[13]	M[14]	M[15]

0 miss

1	010	M[16]	M[17]	M[18]	M[19]	M[20]	M[21]	M[22]	M[23]
1	001	M[8]	M[9]	M[10]	M[11]	M[12]	M[13]	M[14]	M[15]

2 hit

1	010	M[16]	M[17]	M[18]	M[19]	M[20]	M[21]	M[22]	M[23]
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:

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

LRU
10

(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

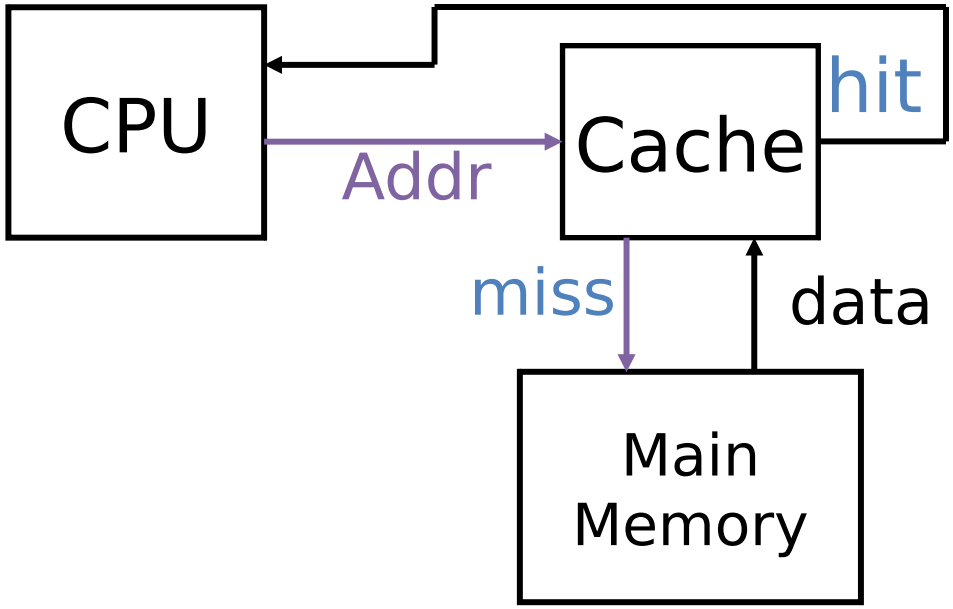
Technology Break

Agenda

- Memory Hierarchy Overview
- Administrivia
- 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)
- 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$)
  - Fastest possible scenario, so want more of these
- Write hits (D$)
  - 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

Handling Cache Hits

- Read hits (I$ and D$)
  - Fastest possible scenario, so want more of these
- Write hits (D$)
  - 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

- Read misses (I$ and D$)
  - Stall execution, fetch block from memory, put in cache, send requested data to processor, resume
- Write misses (D$)
  - Write allocate: Fetch block from memory, put in cache, execute a write hit
    - Works with either write-through or write-back
    - Ensures cache is up-to-date after write miss

Handling Cache Misses

- Read misses (I$ and D$)
  - Stall execution, fetch block from memory, put in cache, send requested data to processor, resume
- Write misses (D$)
  - No-write allocate: Skip cache altogether and write directly to memory
    - Cache is never up-to-date after write miss
    - Ensures memory is always up-to-date

Updated Cache Picture

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

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

LRU
XXXXX

- Write allocate/no-write allocate
  - Affects behavior, not design

Summary (1/2)

- 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)

Summary (2/2)

- Cache read and write policies:
  - *Write-back* and *write-through* for hits
  - *Write allocate* and *no-write allocate* for misses
  - Cache needs dirty bit if write-back