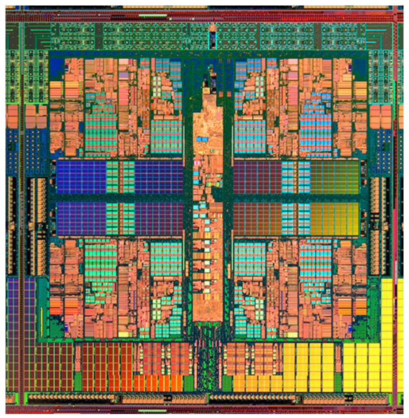
Lecture 17
Introduction to Memory Hierarchies

Suggested reading:

(HP Chapter 5.1-5.2)

Processor components

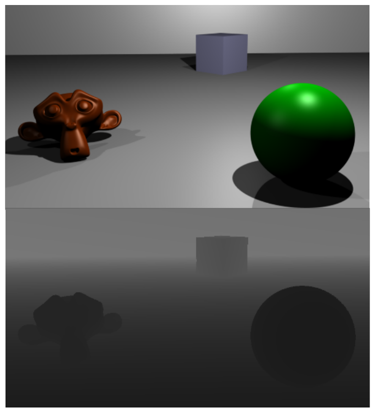
Multicore processors and programming



Processor comparison

AMD Athlon 64 vs. intel Pentium Dual-Core inside

CSE 30321



Writing more efficient code

The right HW for the right application

```
for i=0; i<5; i++ {
        a = (a*b) + c;
}
                      ↓
MULT r1,r2,r3      # r1 ← r2*r3
ADD r2,r1,r4    ↓  # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

HLL code translation

Fundamental lesson(s)

- When you load data from disk or store data in memory, where does it go? How does the processor find it?

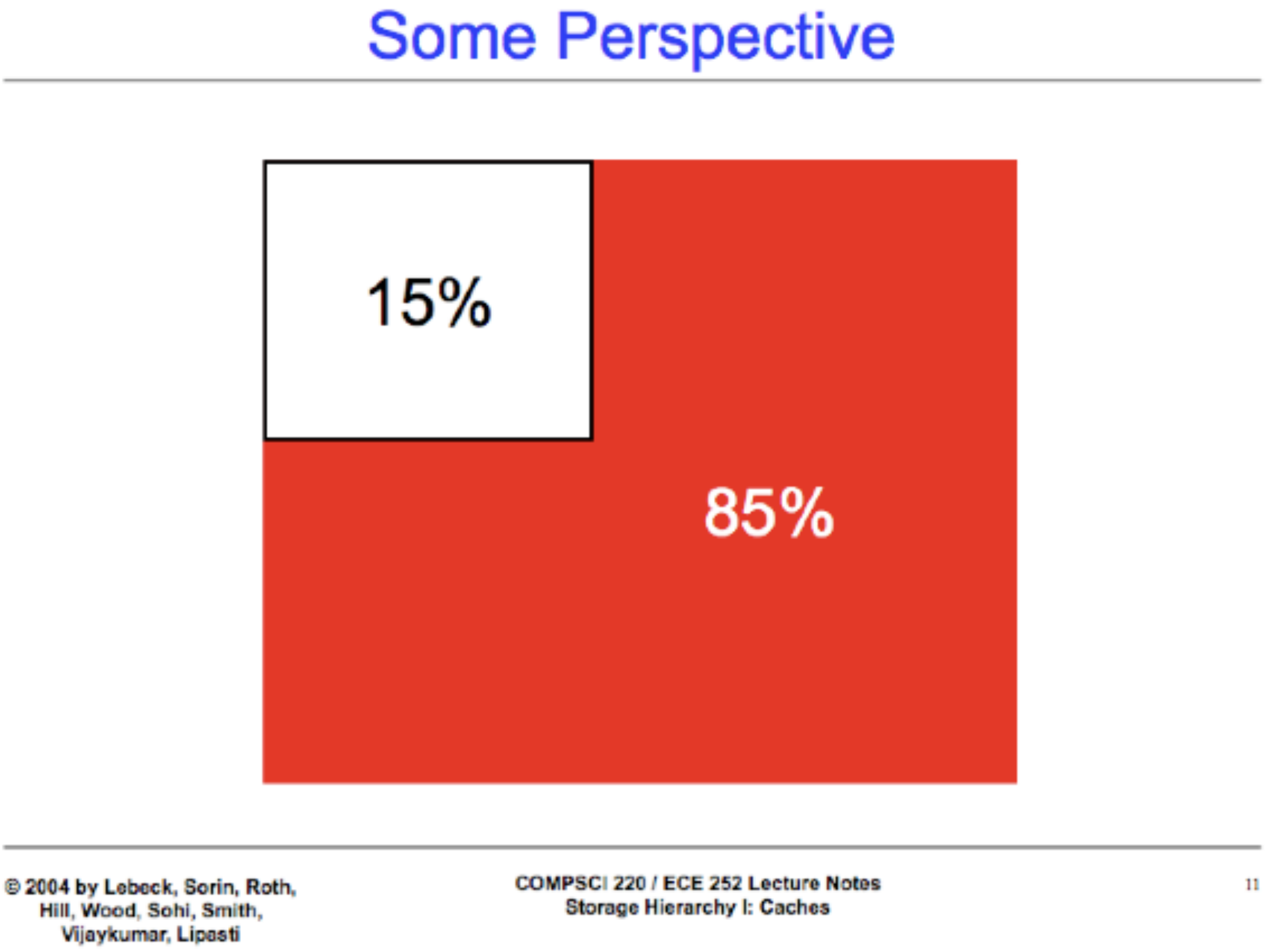
- In a pipelined datapath, we assume that a memory reference takes 1 CC. But an off-chip memory access requires 100s of CCs.
  - How can a memory reference truly be accomplished in 1 CC?

Why it’s important…

- Memory hierarchies prevent programs from having absolutely abysmal performance
  - In lab 4, you’ll see how a detailed understanding of the memory organization can help you write code that is *at least* 2X more efficient

Question?

- How much of a chip is “memory”?
  - 10%
  - 25%
  - 50%
  - 75%
  - 85%



Some Perspective

© 2004 by Lebeck, Sorin, Roth, Hill, Wood, Sohi, Smith, Vijaykumar, Lipasti

COMPSCI 220 / ECE 252 Lecture Notes Storage Hierarchy I: Caches

11

Memory and Pipelining

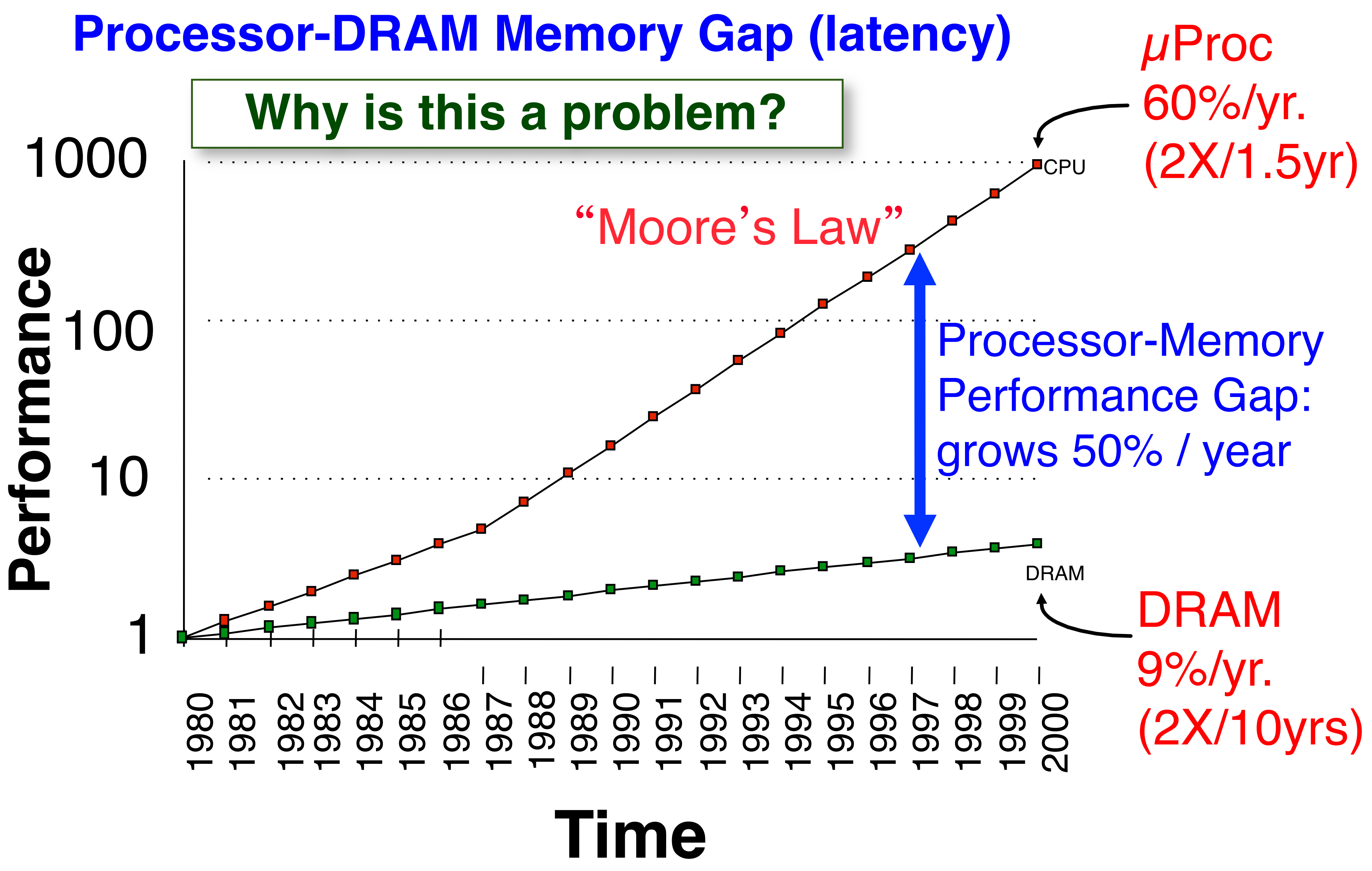
- In our 5 stage pipe, we've constantly been assuming that we can access our operand from memory in 1 clock cycle…
  – We'd like this to be the case, and its possible...but its also complicated
  – We'll discuss how this happens in the next several lectures
- We'll talk about…
  – Memory Technology
  – Memory Hierarchy
    - Caches
    - Memory
    - Virtual Memory

If I say "Memory" what do you think of?

- Memory Comes in Many Flavors
  - SRAM (Static Random Access Memory)
  - DRAM (Dynamic Random Access Memory)
  - ROM, Flash, etc.
  - Disks, Tapes, etc.
- Difference in speed, price and "size"
  - Fast is small and/or expensive
  - Large is slow and/or inexpensive
- The search is on for a "universal memory"
  - What's a "universal memory"
    - Fast and non-volatile.
  - May be MRAM, PCRAM, etc. etc.

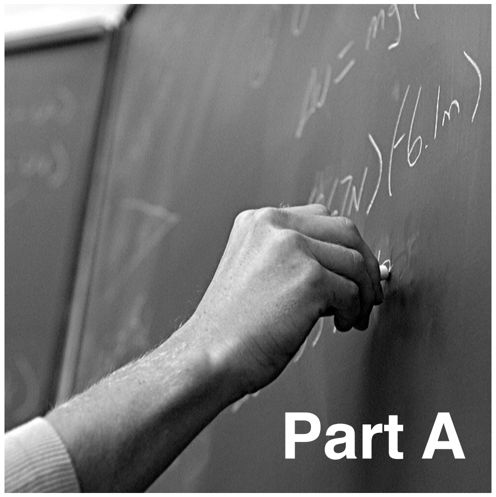
Let's start with DRAM.
Its generally the largest piece of RAM.

Is there a problem with DRAM?



More on DRAM

- DRAM access on order of 100 ns…
- What if clock rate for our 5 stage pipeline was 1 GHZ?
  - Would the memory *and* fetch stages stall for 100 cycles?
    - Actually, yes … if only DRAM
- Caches come to the rescue
  - As transistors have gotten smaller, its allowed us to put more (faster) memory closer to the processing logic
    - The idea is to “mask” the latency of having to go off to main memory



Caches and the principle of locality…

- The principle of locality…
  - …says that most programs don't access all code or data uniformly
    - i.e. in a loop, small subset of instructions might be executed over and over again…
    - …and a block of memory addresses might be accessed sequentially…

- This has led to "memory hierarchies"
- Some important things to note:
  - Fast memory is expensive in cost/size
  - Levels of memory usually smaller/faster than previous
  - Levels of memory usually "subset" one another
    - All the stuff in a higher level is in some level below it

Common memory hierarchy:

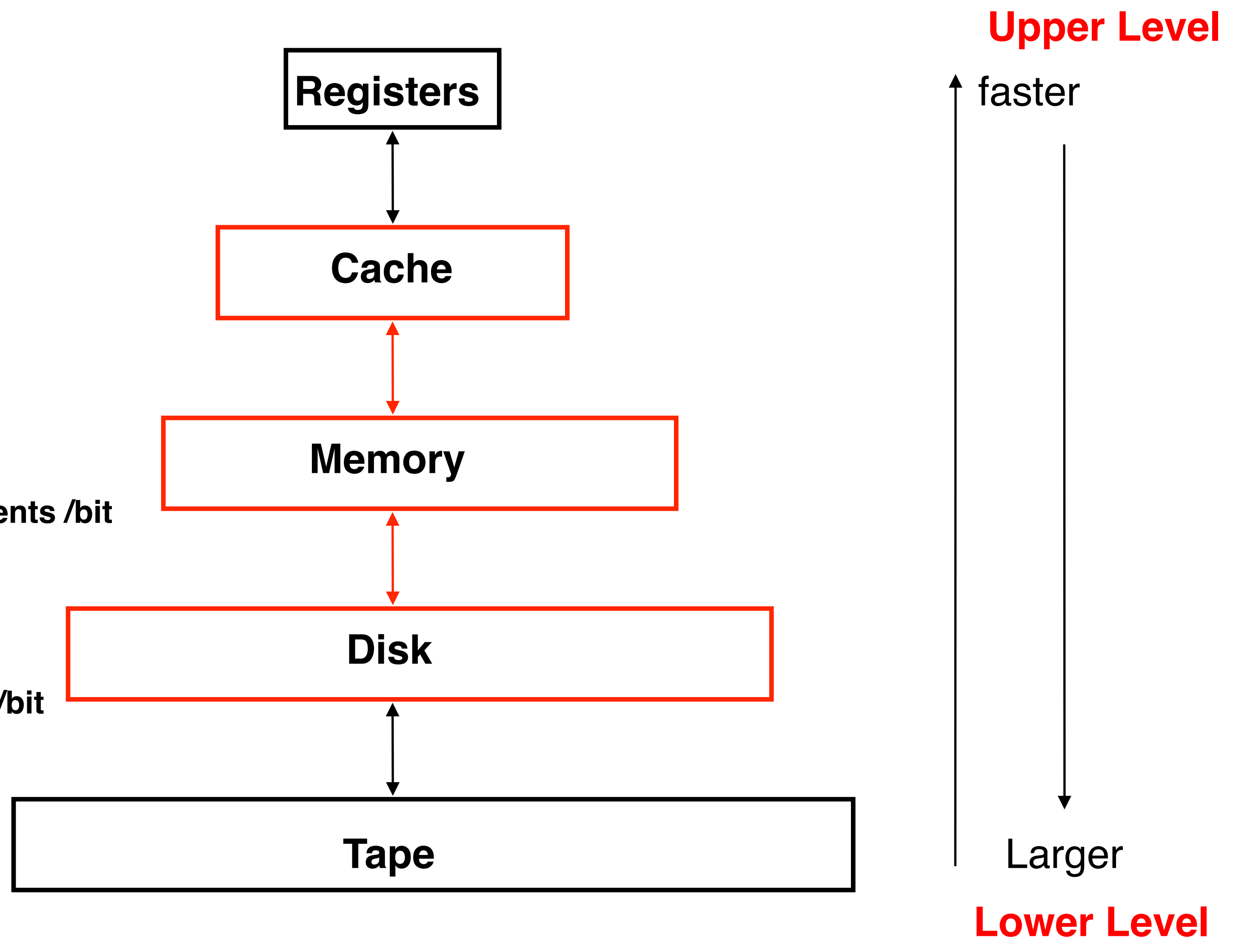
CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
1-0.1 cents/bit

Main Memory
M Bytes
200ns- 500ns
$.0001-.00001 cents /bit

Disk
G Bytes, 10 ms
(10,000,000 ns)
10^{-5} - 10^{-6} cents/bit

Tape
infinite
sec-min
10^{-8}



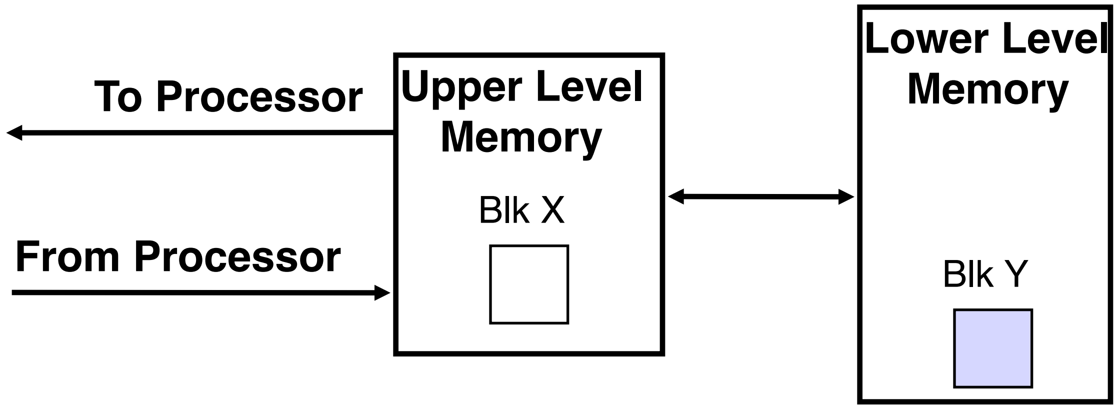
Terminology Summary

- Hit: data appears in block in upper level (i.e. block X in cache)
  - Hit Rate: fraction of memory access found in upper level
  - Hit Time: time to access upper level which consists of
    - RAM access time + Time to determine hit/miss

- Miss: data needs to be retrieved from a block in the lower level

  (i.e. block Y in memory)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Extra time to replace a block in the upper level +
    - Time to deliver the block the processor

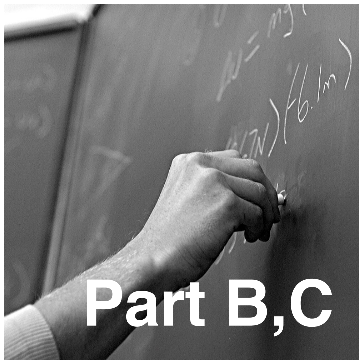
- Hit Time << Miss Penalty (500 instructions on some microprocessors)



Average Memory Access Time

AMAT = (Hit Time) + (1 - h) x (Miss Penalty)

- **Hit time:**
  - basic time of every access.
- **Hit rate (h):**
  - fraction of access that hit
- **Miss penalty:**
  - extra time to fetch a block from lower level, including time to replace in CPU



Cache Basics

- Cache entries usually referred to as "blocks"
  - Block is minimum amount of information that can be in cache
  - Line size typically power of two
  - Typically 16 to 128 bytes in size

Some initial questions to consider

- Where can a block be placed in an upper level of memory hierarchy (i.e. a cache)?

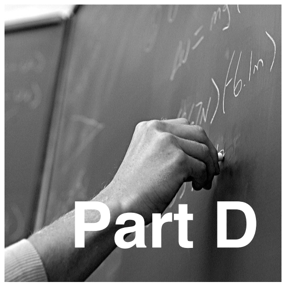
- How is a block found in an upper level of memory hierarchy?

- Which cache block should be replaced on a cache miss if entire cache is full and we want to bring in new data?

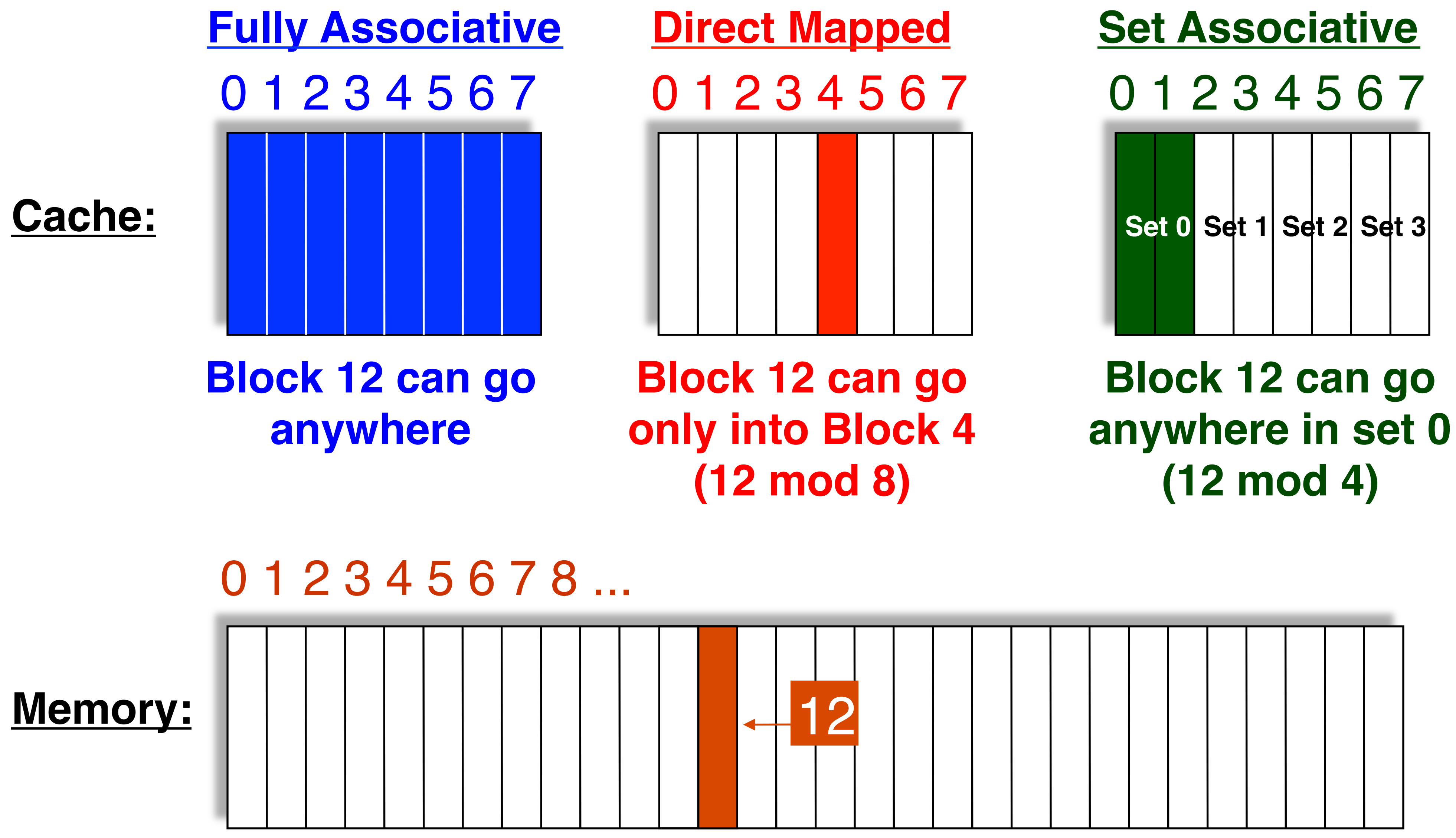
- What happens if a you want to write back to a memory location?
  - Do you just write to the cache?
  - Do you write somewhere else?

Where can a block be placed in a cache?

- 3 schemes for block placement in a cache:
  - Direct mapped cache:
    - Block (or data to be stored) can go to only 1 place in cache
    - Usually: (Block address) MOD (# of blocks in the cache)
  - Fully associative cache:
    - Block can be placed anywhere in cache
  - Set associative cache:
    - "Set" = a group of blocks in the cache
    - Block mapped onto a set & then block can be placed anywhere within that set
    - Usually: (Block address) MOD (# of sets in the cache)
    - If n blocks, we call it n-way set associative



Where can a block be placed in a cache?



Associativity

- If you have associativity > 1 you have to have a replacement policy
  - FIFO
  - LRU
  - Random

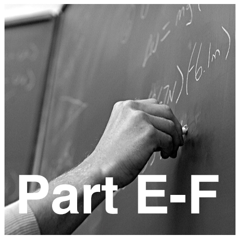
- “Full” or “Full-map” associativity means you check every tag in parallel and a memory block can go into any cache block
  - Virtual memory is effectively fully associative
  - (But don’t worry about virtual memory yet)

How is a block found in the cache?

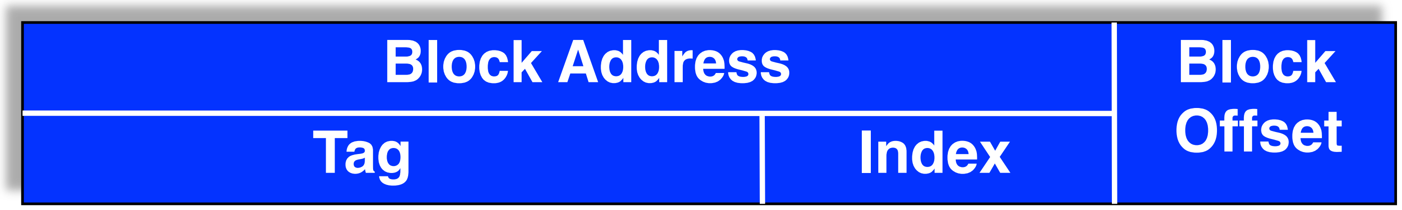
- Cache's have address tag on each block frame that provides block address
  - Tag of every cache block that might have entry is examined against CPU address (in parallel! – why?)

- Each entry usually has a valid bit
  - Tells us if cache data is useful/not garbage
  - If bit is not set, there can't be a match…

- How does address provided to CPU relate to entry in cache?
  - Entry divided between block address & block offset…
  - …and further divided between tag field & index field



How is a block found in the cache?



- Block offset field selects data from block
  - (i.e. address of desired data within block)
- Index field selects a specific set
- Tag field is compared against it for a hit

- Could we compare on more of address than the tag?
  - Not necessary; checking index is redundant
    - Used to select set to be checked
    - Ex.: Address stored in set 0 must have 0 in index field
  - Offset not necessary in comparison – entire block is present or not and all block offsets must match

Which block is replaced on a cache miss?

- If we look something up in cache and entry not there, generally want to get data from memory and put it in cache
  - B/c principle of locality says we'll probably use it again

- Direct mapped caches have 1 choice of what block to replace
- Fully associative or set associative offer more choices
- Usually 2 strategies:
  - Random – pick any possible block and replace it
  - LRU – stands for "Least Recently Used"
    - Why not throw out the block not used for the longest time
    - Usually approximated, not much better than random – i.e. 5.18% vs. 5.69% for 16KB 2-way set associative

What happens on a write?

- FYI most accesses to a cache are reads:
  - Used to fetch instructions (reads)
  - Most instructions don't write to memory
    - For MIPS only about 7% of memory traffic involve writes
    - Translates to about 25% of cache data traffic

- Make common case fast! Optimize cache for reads!
  - Actually pretty easy to do…
  - Can read block while comparing/reading tag
  - Block read begins as soon as address available
  - If a hit, address just passed right on to CPU

What happens on a write?

- Generically, there are 2 kinds of write policies:
  - Write through
    - With write through, information written to block in cache and to block in lower-level memory
  - Write back
    - With write back, information written only to cache. It will be written back to lower-level memory when cache block is replaced
- The dirty bit:
  - Each cache entry usually has a bit that specifies if a write has occurred in that block or not…
  - Helps reduce frequency of writes to lower-level memory upon block replacement

What happens on a write?

- Write back versus write through:
  - Write back advantageous because:
    - Writes occur at the speed of cache and don't incur delay of lower-level memory
    - Multiple writes to cache block result in only 1 lower-level memory access
  - Write through advantageous because:
    - Lower-levels of memory have most recent copy of data

- If CPU has to wait for a write, we have write stall
  - 1 way around this is a write buffer
  - Ideally, CPU shouldn't have to stall during a write
  - Instead, data written to buffer which sends it to lower-levels of memory hierarchy

What happens on a write?

- What if we want to write and block we want to write to isn’t in cache?

- There are 2 common policies:
  - **Write allocate:**
    - The block is loaded on a write miss
    - The idea behind this is that subsequent writes will be captured by the cache (ideal for a write back cache)
  - **No-write allocate:**
    - Block modified in lower-level and not loaded into cache
    - Usually used for write-through caches
      - (subsequent writes still have to go to memory)

Memory access equations

- Using what we defined on previous slide, we can say:
  - Memory stall clock cycles =
    - Reads x Read miss rate x Read miss penalty +
    - Writes x Write miss rate x Write miss penalty

- Often, reads and writes are combined/averaged:
  - Memory stall cycles =
    - Memory access x Miss rate x Miss penalty (approximation)

- Also possible to factor in instruction count to get a "complete" formula:

$$CPU\ time = IC \times \left(CPI_{execution} + \frac{Memory\ stall\ clock\ cycles}{Instruction} \right) \times Clock\ cycle\ time$$

Reducing cache misses

- Obviously, we want data accesses to result in cache hits, not misses –this will optimize performance

- Start by looking at ways to increase % of hits….

- …but first look at 3 kinds of misses!
  - Compulsory misses:
    - Very 1st access to cache block will not be a hit –the data's not there yet!
  - Capacity misses:
    - Cache is only so big. Won't be able to store every block accessed in a program – must swap out!
  - Conflict misses:
    - Result from set-associative or direct mapped caches
    - Blocks discarded/retrieved if too many map to a location