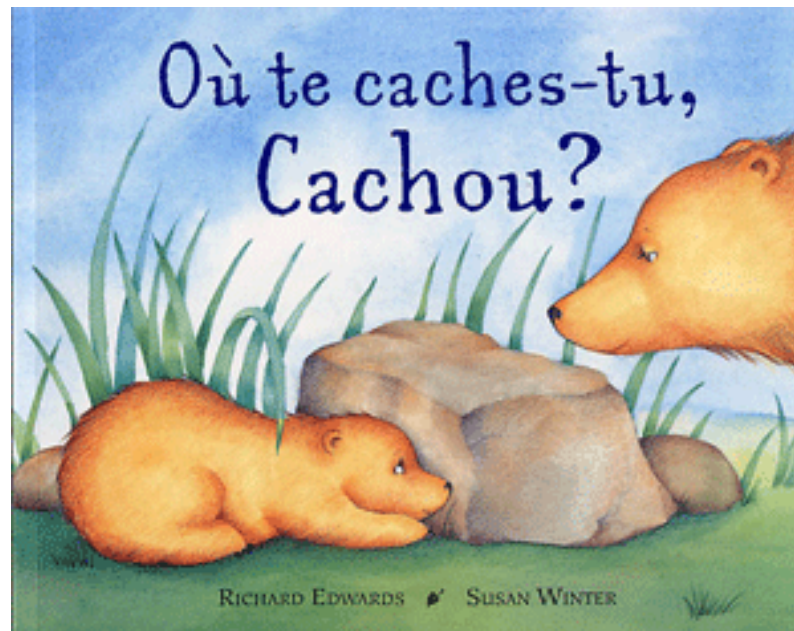


inst.eecs.berkeley.edu/~cs61c/su06  
**CS61C : Machine Structures**

## Lecture #20: Caches 1



**2006-08-02**

**Andy Carle**



CS61C L20 Caches I (1)

A Carle, Summer 2006 © UCB

# Review : Pipelining

---

- **Pipeline challenge is hazards**
  - Forwarding helps w/many data hazards
  - Delayed branch helps with control hazard in our 5 stage pipeline
  - Data hazards w/Loads  $\Rightarrow$  Load Delay Slot
    - Interlock  $\Rightarrow$  “smart” CPU has HW to detect if conflict with inst following load, if so it stalls
- **More aggressive performance:**
  - Superscalar (parallelism)
  - Out-of-order execution



# Big Ideas so far

---

- **8 weeks to learn big ideas in CS&E**
  - Principle of abstraction, used to build systems as layers
  - Pliable Data: a program determines what it is
  - Stored program concept: instructions just data
  - Compilation v. interpretation to move down layers of system
  - Greater performance by exploiting parallelism (pipeline)
  - Principle of Locality, exploited via a memory hierarchy (cache)
  - Principles/Pitfalls of Performance Measurement



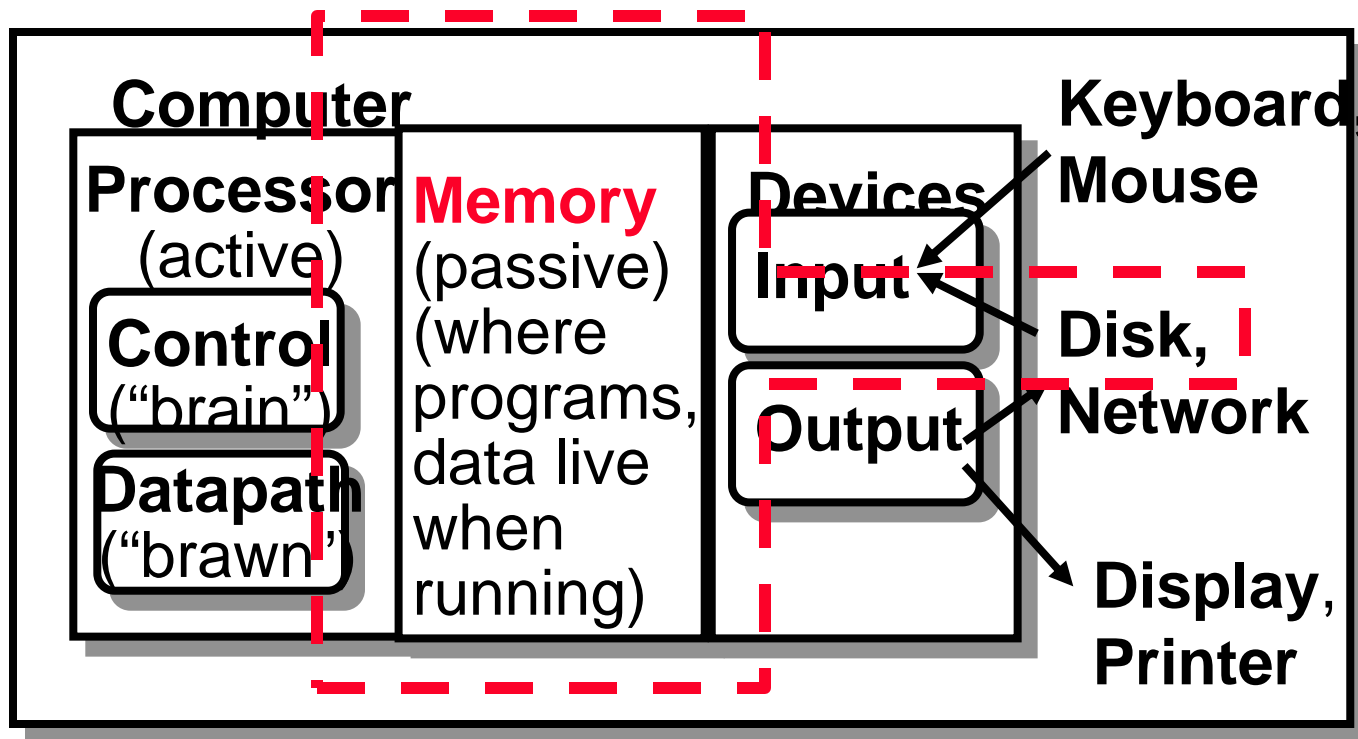
# Where are we now in 61C?

---

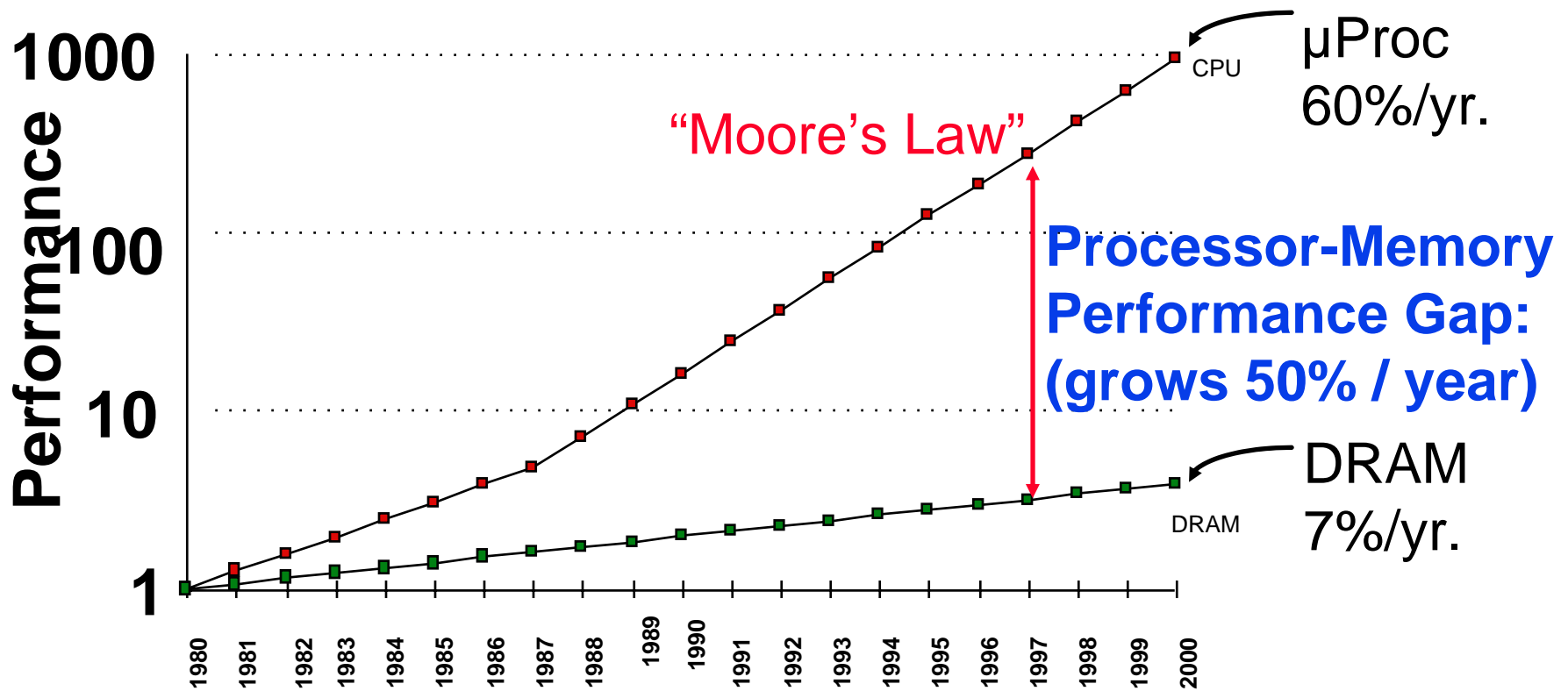
- **Architecture!** (aka “Systems”)
  - CPU Organization
  - Pipelining
  - **Caches**
  - **Virtual Memory**
  - **I / O**
  - **Networks**
  - **Performance**



# The Big Picture



# What's the Problem?



- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two levels of cache on chip



# Memory Hierarchy (1/3)

---

- **Processor**

- executes instructions on order of nanoseconds to picoseconds
- holds a small amount of code and data in registers

- **Memory**

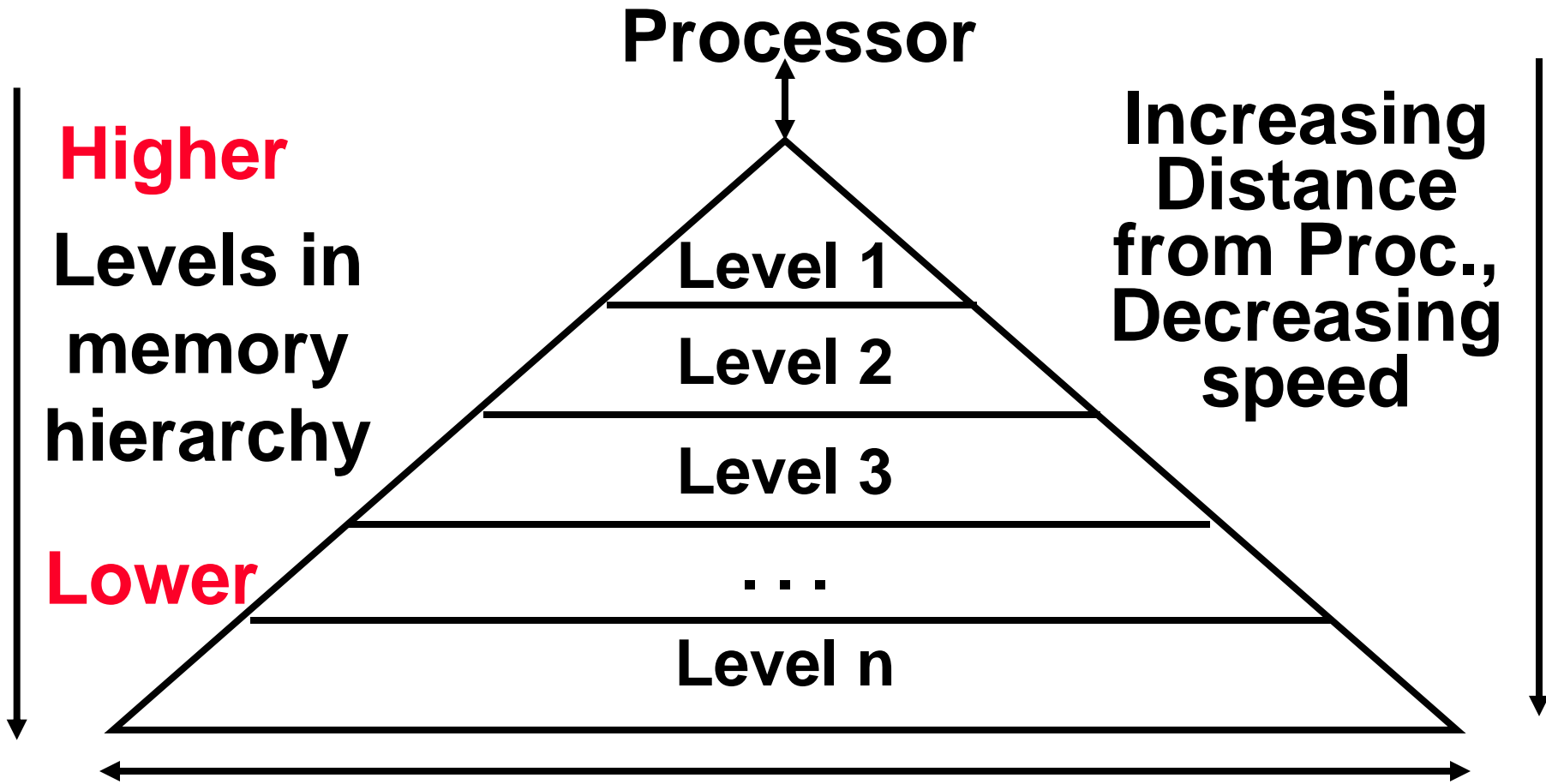
- More capacity than registers, still limited
- Access time ~50-100 ns

- **Disk**

- HUGE capacity (virtually limitless)
- VERY slow: runs ~milliseconds



# Memory Hierarchy (2/3)



**Size of memory at each level**

*As we move to deeper levels the latency goes up and price per bit goes down.*

**Q: Can \$/bit go up as move deeper?**





# Memory Hierarchy (3/3)

---

- **If level closer to Processor, it must be:**
  - smaller
  - faster
  - subset of lower levels (contains most recently used data)
- **Lowest Level (usually disk) contains all available data**
- **Other levels?**



# Memory Caching

---

- We've discussed three levels in the hierarchy: processor, memory, disk
- Mismatch between processor and memory speeds leads us to add a new level: a memory **cache**
- Implemented with SRAM technology: faster but more expensive than DRAM memory.
  - "S" = **Static**, no need to refresh, ~10ns
  - "D" = **Dynamic**, need to refresh, ~60ns
  - [arstechnica.com/paedia/r/ram\\_guide/ram\\_guide.part1-1.html](http://arstechnica.com/paedia/r/ram_guide/ram_guide.part1-1.html)



# Memory Hierarchy Analogy: Library (1/2)

- You're writing a term paper (Processor) at a **table** in **Doe**
- **Doe** Library is equivalent to disk
  - essentially limitless capacity
  - very slow to retrieve a book
- **Table** is memory
  - smaller capacity: means you must return book when table fills up
  - easier and faster to find a book there once you've already retrieved it



## Memory Hierarchy Analogy: Library (2/2)

- Open books on table are **cache**
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster to retrieve data
- Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library



# Memory Hierarchy Basis

---

- Disk contains everything.
- When Processor needs something, bring it into to all higher levels of memory.
- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Entire idea is based on Temporal Locality: if we use it now, we'll want to use it again soon (a Big Idea)



# Cache Design

---

- **How do we organize cache?**
- **Where does each memory address map to?**  
(Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
- **How do we know which elements are in cache?**
- **How do we quickly locate them?**



# Pre-Exam Exercise #1

---

**We are now going to stop for ~5 minutes. During this time, your goal is to (by yourself) come up with a potential exam exercise covering the topic of Floating Point or CALL. Make it as much like a real exam question as possible.**

**After this five minutes, you will explain your question to a small group and work through how you would go about solving it. I'll call on some random samples for the full class.**



# Administrivia

---

- **HW5 Due Now**
- **HW6 Due Saturday**
- **Project 3 Due 8/8**
  
- **Midterm 2:**
  - **Friday, 11:00am – 2:00pm**
  - **390 HMMB**
  - **Conflicts, DSP, &&|| terrified about the drop deadline: Contact Andy ASAP**





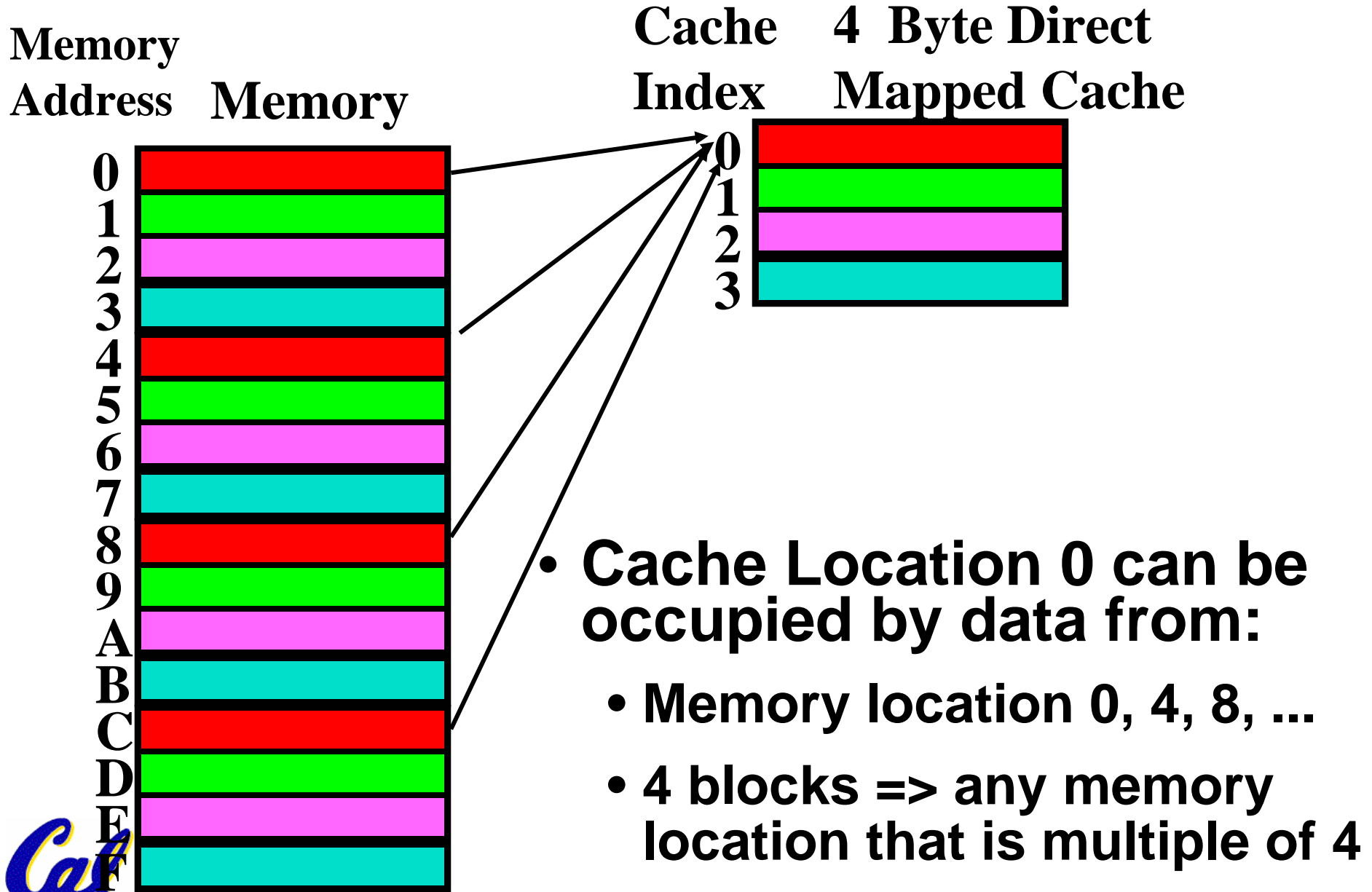
# Direct-Mapped Cache (1/2)

---

- In a **direct-mapped cache**, each memory address is associated with one possible **block** within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory



# Direct-Mapped Cache (2/2)



# Issues with Direct-Mapped

---

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size  $> 1$  byte?
- Answer: divide memory address into three fields



tag  
to check  
if have  
correct block

index  
to  
select  
block

byte  
offset  
within  
block



# Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- **Index**: specifies the cache index (which “row” of the cache we should look in)
- **Offset**: once we’ve found correct block, specifies which byte within the block we want
- **Tag**: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location



# Caching Terminology

---

- When we try to read memory, 3 things can happen:
  1. **cache hit**:  
cache block is valid and contains proper address, so read desired word
  2. **cache miss**:  
nothing in cache in appropriate block, so fetch from memory
  3. **cache miss, block replacement**:  
wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)



## Direct-Mapped Cache Example (1/3)

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- Offset
  - need to specify correct byte within a block
  - block contains 4 words
    - = 16 bytes
    - =  $2^4$  bytes
  - need 4 bits to specify correct byte



## Direct-Mapped Cache Example (2/3)

- Index: (~index into an “array of blocks”)
  - need to specify correct row in cache
  - cache contains 16 KB =  $2^{14}$  bytes
  - block contains  $2^4$  bytes (4 words)
  - # blocks/cache
    - =  $\frac{\text{bytes/cache}}{\text{bytes/block}}$
    - =  $\frac{2^{14} \text{ bytes/cache}}{2^4 \text{ bytes/block}}$
    - =  $2^{10}$  blocks/cache
  - need 10 bits to specify this many rows



## Direct-Mapped Cache Example (3/3)

- **Tag: use remaining bits as tag**
  - tag length = addr length – offset - index  
= 32 - 4 - 10 bits  
= 18 bits
  - so tag is leftmost **18 bits** of memory address
- **Why not full 32 bit address as tag?**
  - All bytes within block need same address (4b)
  - Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory (here 10 bits)





# Peer Instruction

---

- A. Mem hierarchies **were invented before 1950.** (UNIVAC I wasn't delivered 'til 1951)
- B. If you know your computer's cache size, you can often **make your code run faster.**
- C. Memory hierarchies take advantage of **spatial locality** by keeping the most recent data items **closer** to the processor.



# Peer Instruction Answer

---

- A. “We are...forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less accessible.” – von Neumann, 1946
  - B. Certainly! That’s call “tuning”
  - C. “Most Recent” items  $\Rightarrow$  Temporal locality
- 
- A. Mem hierarchies **were invented before 1950.** (UNIVAC I wasn’t delivered ‘til 1951)
  - B. If you know your computer’s cache size, you can often **make your code run faster.**
  - C. Memory hierarchies take advantage of **spatial locality** by keeping the most recent data items **closer** to the processor.



## And in conclusion...

---

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.
- So we create a memory hierarchy:
  - each successively lower level contains “most used” data from next higher level
  - exploits temporal locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- **Locality of reference is a Big Idea**

