

# CPU Pipelining Issues

What have you been  
beating your head  
against?



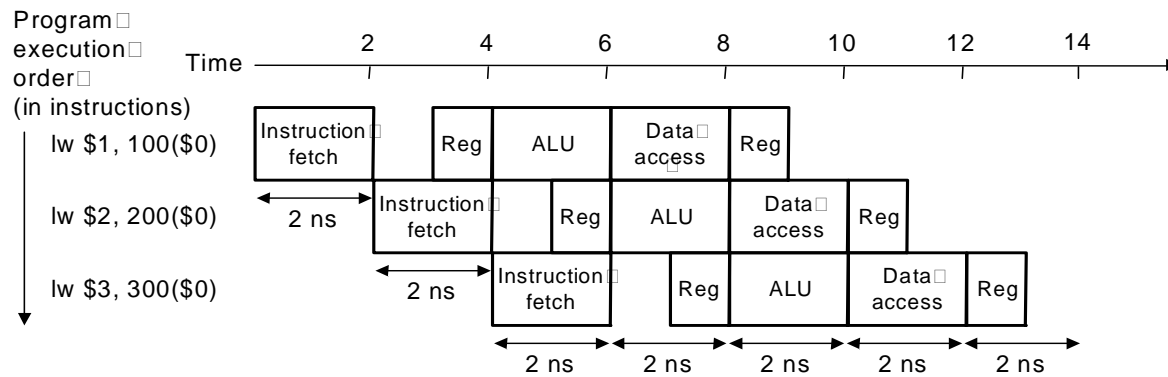
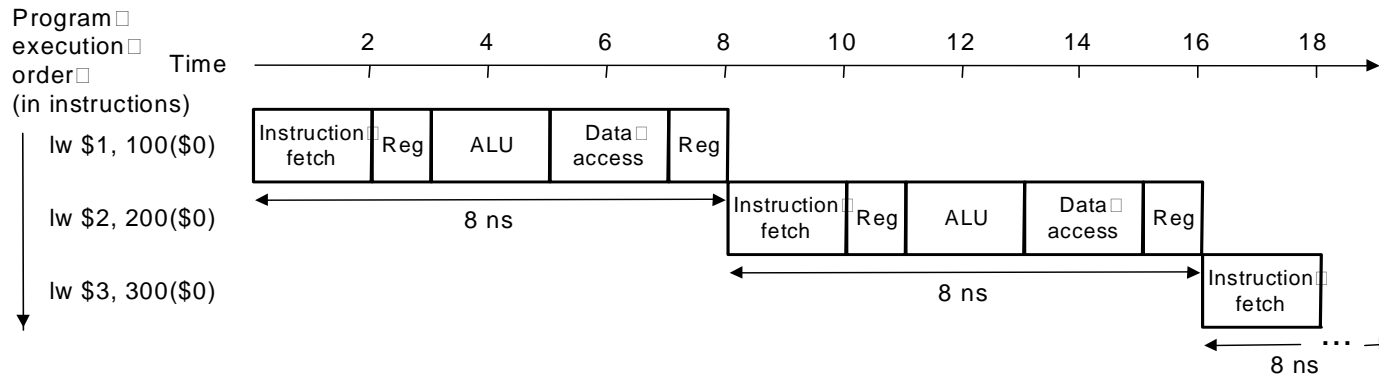
This pipe stuff makes  
my head hurt!



## Finishing up Chapter 6

# Pipelining

Improve performance by increasing instruction throughput



*Ideal speedup is number of stages in the pipeline. Do we achieve this?*

# Pipelining

## What makes it easy

all instructions are the same length

just a few instruction formats

memory operands appear only in loads and stores

## What makes it hard?

structural hazards: suppose we had only one memory

control hazards: need to worry about branch instructions

data hazards: an instruction depends on a previous instruction

Individual Instructions still take the same number of cycles

But we've improved the through-put by increasing the number of simultaneously executing instructions

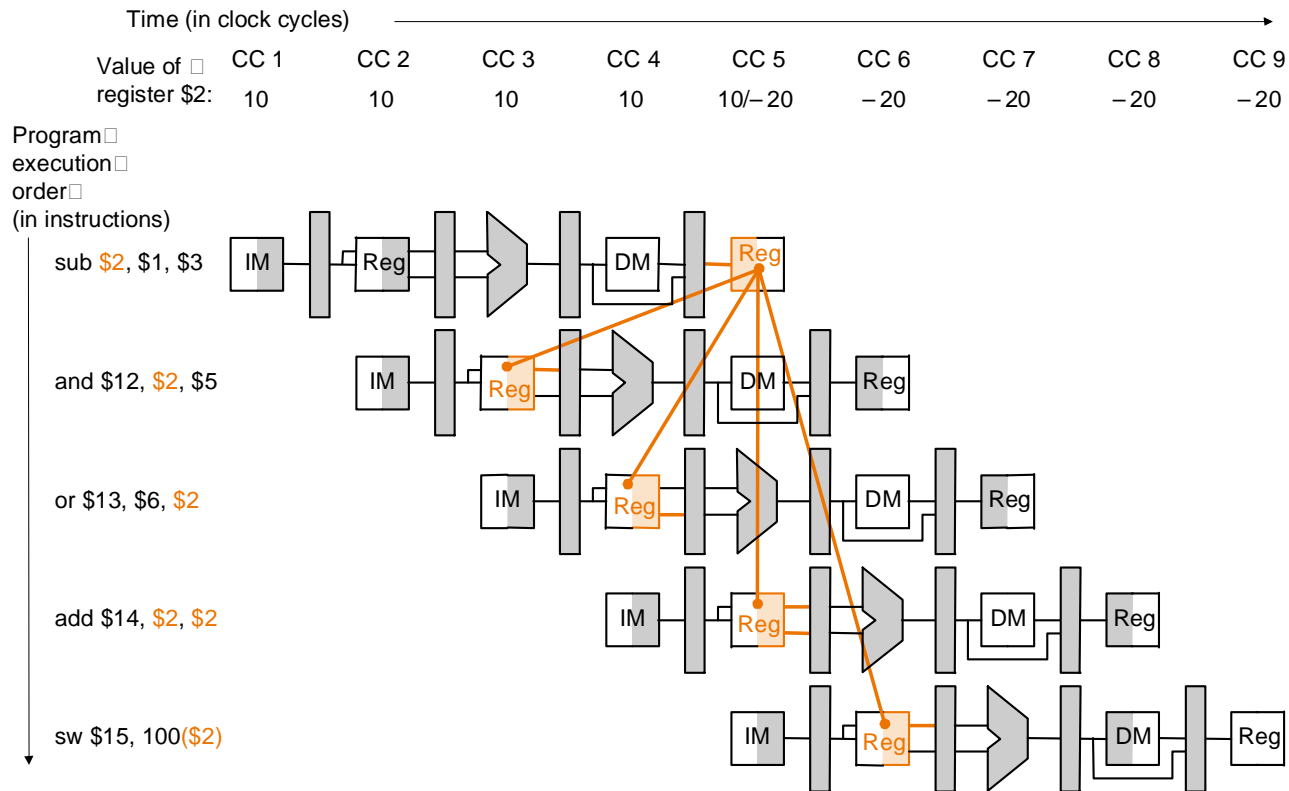
# Structural Hazards

Inst Fetch	Reg Read	ALU	Data Access	Reg Write			
	Inst Fetch	Reg Read	ALU	Data Access	Reg Write		
		Inst Fetch	Reg Read	ALU	Data Access	Reg Write	
			Inst Fetch	Reg Read	ALU	Data Access	Reg Write

# Data Hazards

Problem with *starting next instruction before first is finished*

*dependencies that “go backward in time” are data hazards*



# Software Solution

Have compiler guarantee no hazards

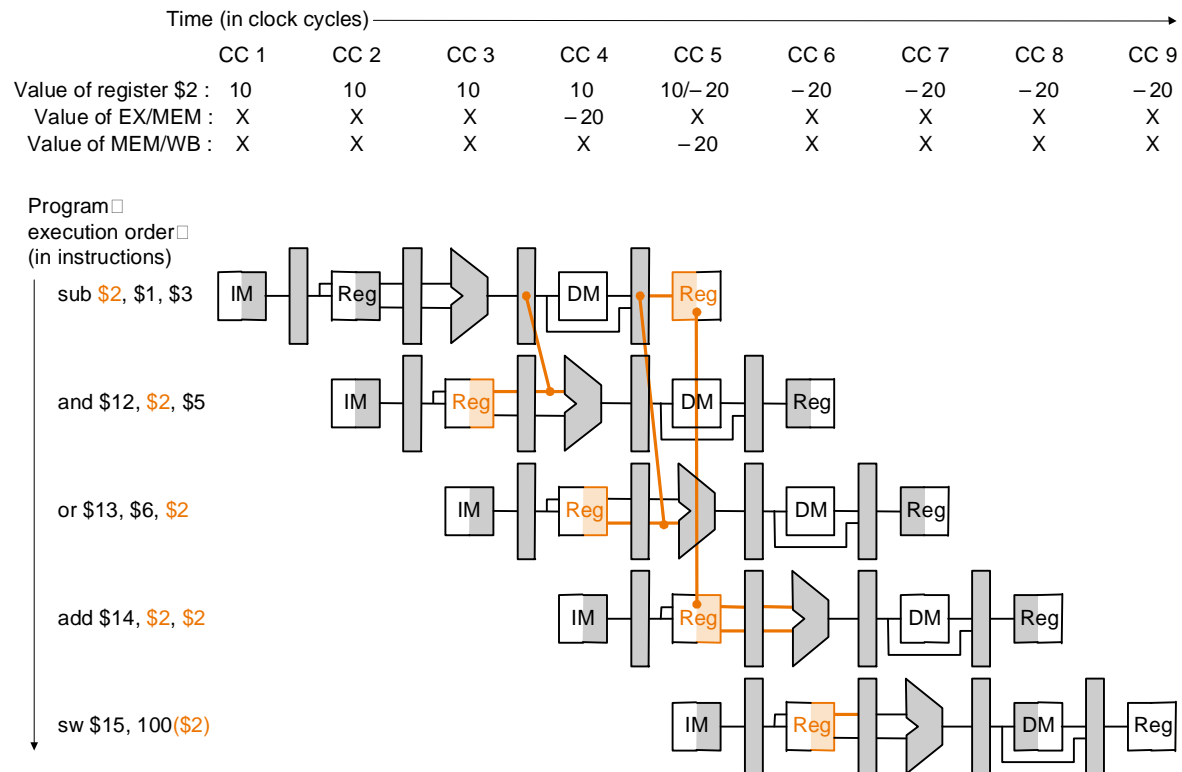
Where do we insert the “nops” ?

```
sub    $2, $1, $3
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)
```

Problem: this really slows us down!

# Forwarding

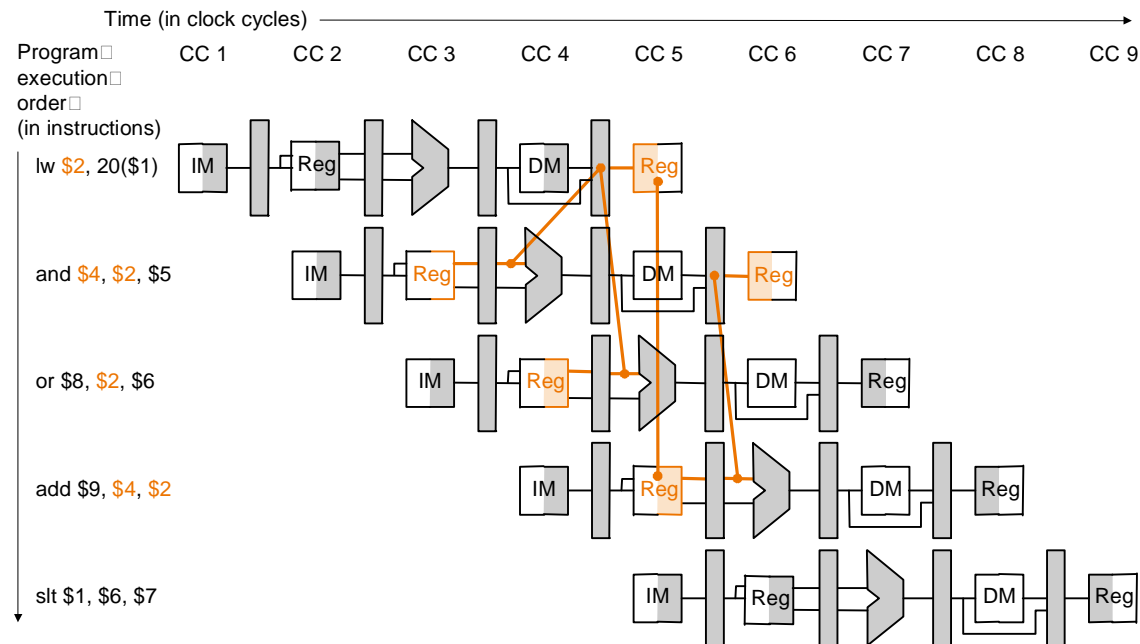
Use temporary results, don't wait for them to be written register file  
 forwarding to handle read/write to same register ALU forwarding



# Can't always forward

Load word can still cause a hazard:

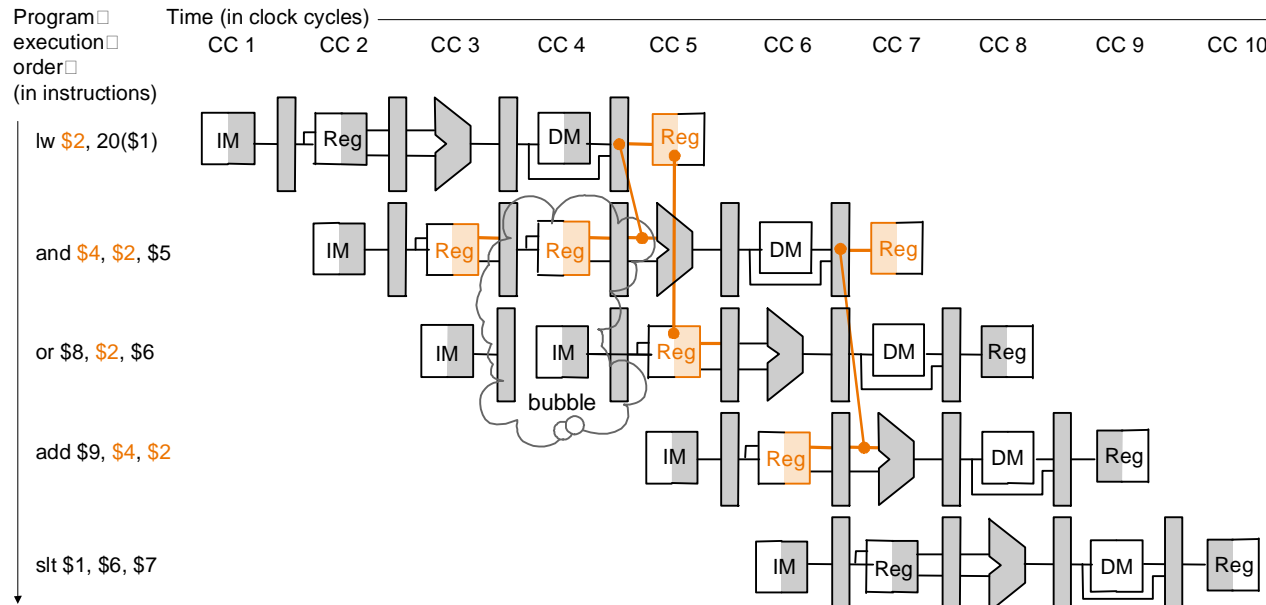
an instruction tries to read a register following a load instruction that writes to the same register.



Thus, we need a hazard detection unit to “stall” the instruction

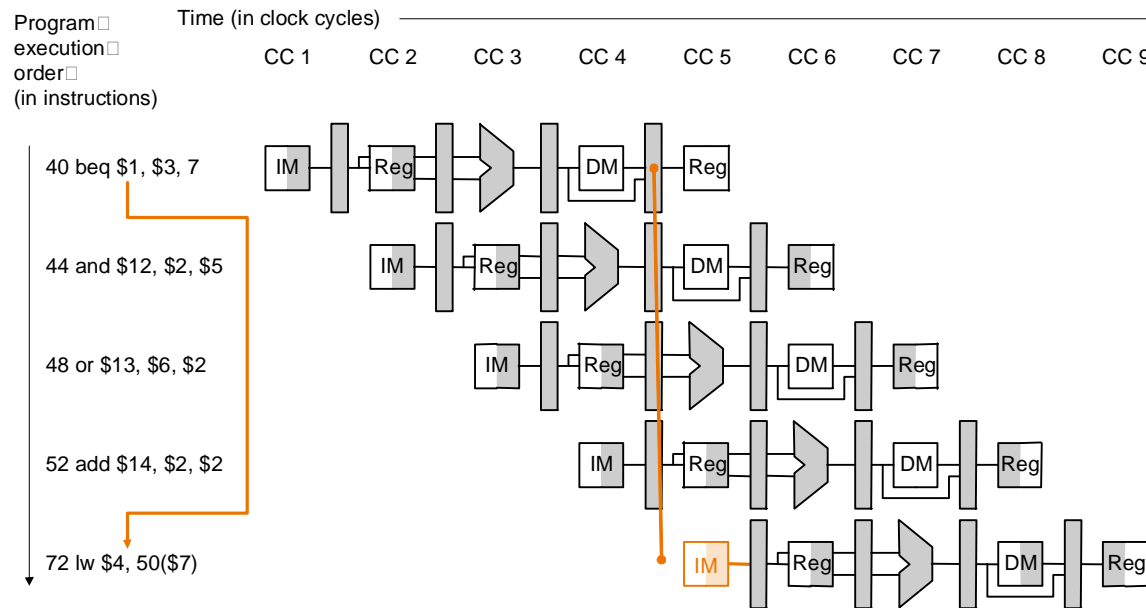
# Stalling

We can stall the pipeline by keeping an instruction in the same stage



# Branch Hazards

When we decide to branch, other instructions are in the pipeline!



We are predicting “branch not taken”

need to add hardware for flushing instructions if we are wrong

# Improving Performance

Try to avoid stalls! E.g., reorder these instructions:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Add a “branch delay slot”

the next instruction after a branch is always executed  
rely on compiler to “fill” the slot with something useful

Superscalar: start more than one instruction in the same cycle

# Dynamic Scheduling

The hardware performs the “scheduling”

hardware tries to find instructions to execute

out of order execution is possible

speculative execution and dynamic branch prediction

All modern processors are very complicated

Pentium 4: 20 stage pipeline, 6 simultaneous instructions

PowerPC and Pentium: branch history table

Compiler technology important

# Pipeline Summary (I)

- Started with unpipelined implementation
  - direct execute, 1 cycle/instruction
  - it had a long cycle time: mem + regs + alu + mem + wb
- We ended up with a 5-stage pipelined implementation
  - increase throughput (3x???)
  - delayed branch decision (1 cycle)
    - Choose to execute instruction after branch
  - delayed register writeback (3 cycles)
    - Add bypass paths ( $6 \times 2 = 12$ ) to forward correct value
  - memory data available only in WB stage
    - Introduce NOPs at IR<sup>ALU</sup>, to stall IF and RF stages until LD result was ready

# Pipeline Summary (II)

## Fallacy #1: Pipelining is easy

Smart people get it wrong all of the time!

## Fallacy #2: Pipelining is independent of ISA

Many ISA decisions impact how easy/costly it is to implement pipelining (i.e. branch semantics, addressing modes).

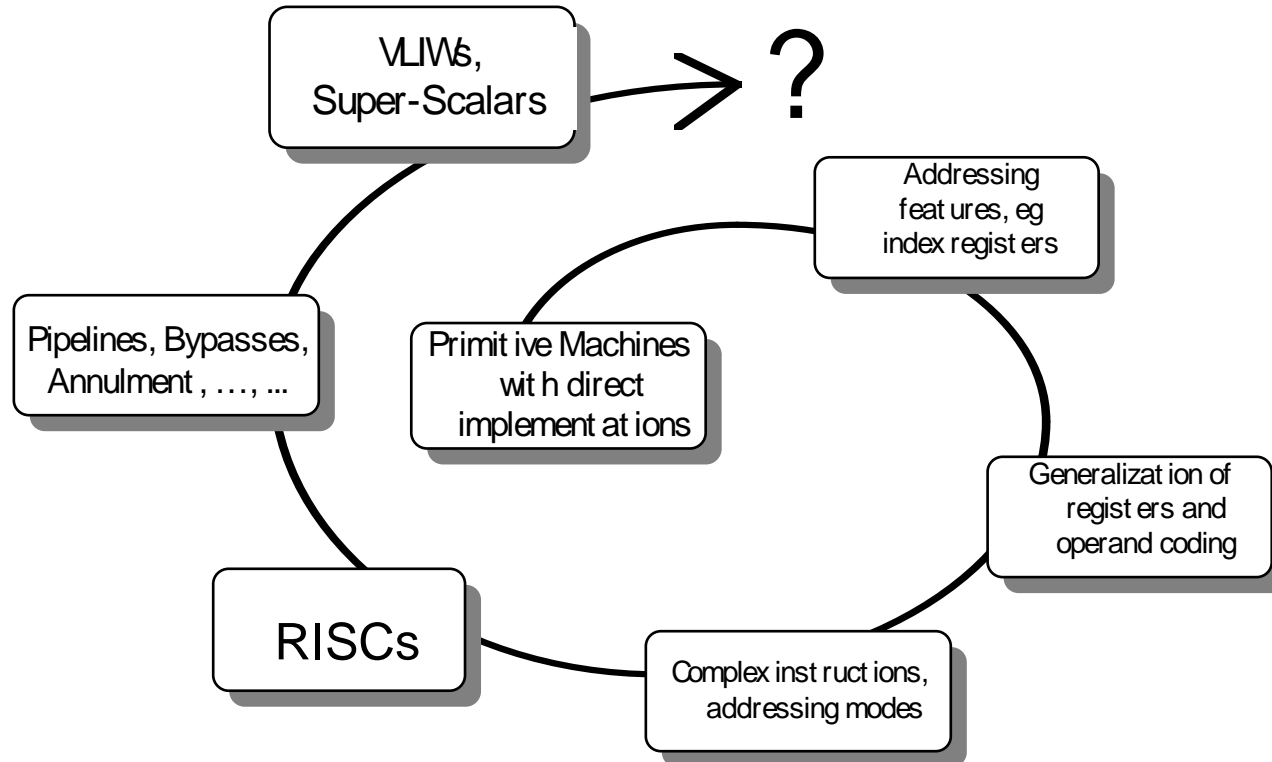
## Fallacy #3: Increasing Pipeline stages improves performance

Diminishing returns. Increasing complexity.

# RISC = Simplicity???

“The P.T. Barnum World’s Tallest Dwarf Competition”

World’s Most Complex RISC?



# Chapter 7 Preview

## Memory Hierarchy

# Memory Hierarchy

Memory devices come in several different flavors

SRAM – Static Ram

fast (1 to 10ns)

expensive (>10 times DRAM)

small capacity (< 1/4 DRAM)

DRAM – Dynamic RAM

16 times slower than SRAM (50ns – 100ns)

Access time varies with address

Affordable (\$160 / gigabyte)

1 Gig considered big

DISK

Slow! (10ms access time)

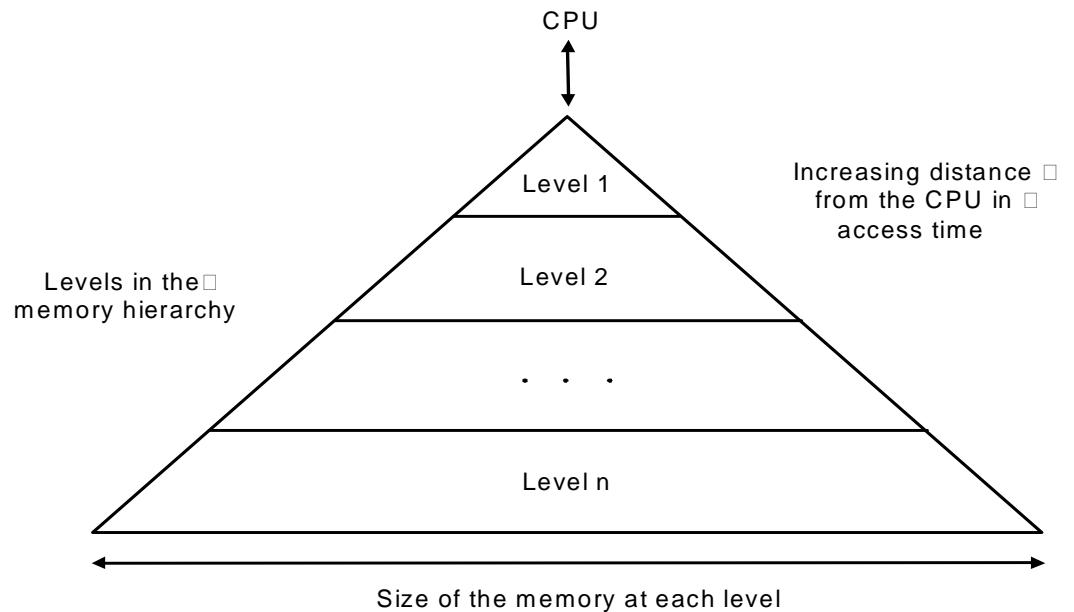
Cheap! (< \$1 / gigabyte)

Big! (1Tbyte is no problem)

# Memory Hierarchy

Users want large and fast memories!

Try to give it to them  
build a memory hierarchy



# Locality

A principle that makes having a memory hierarchy a good idea

If an item is referenced,

*temporal locality*: it will tend to be referenced again soon

*spatial locality*: nearby items will tend to be referenced soon.

*Why does code have locality?*