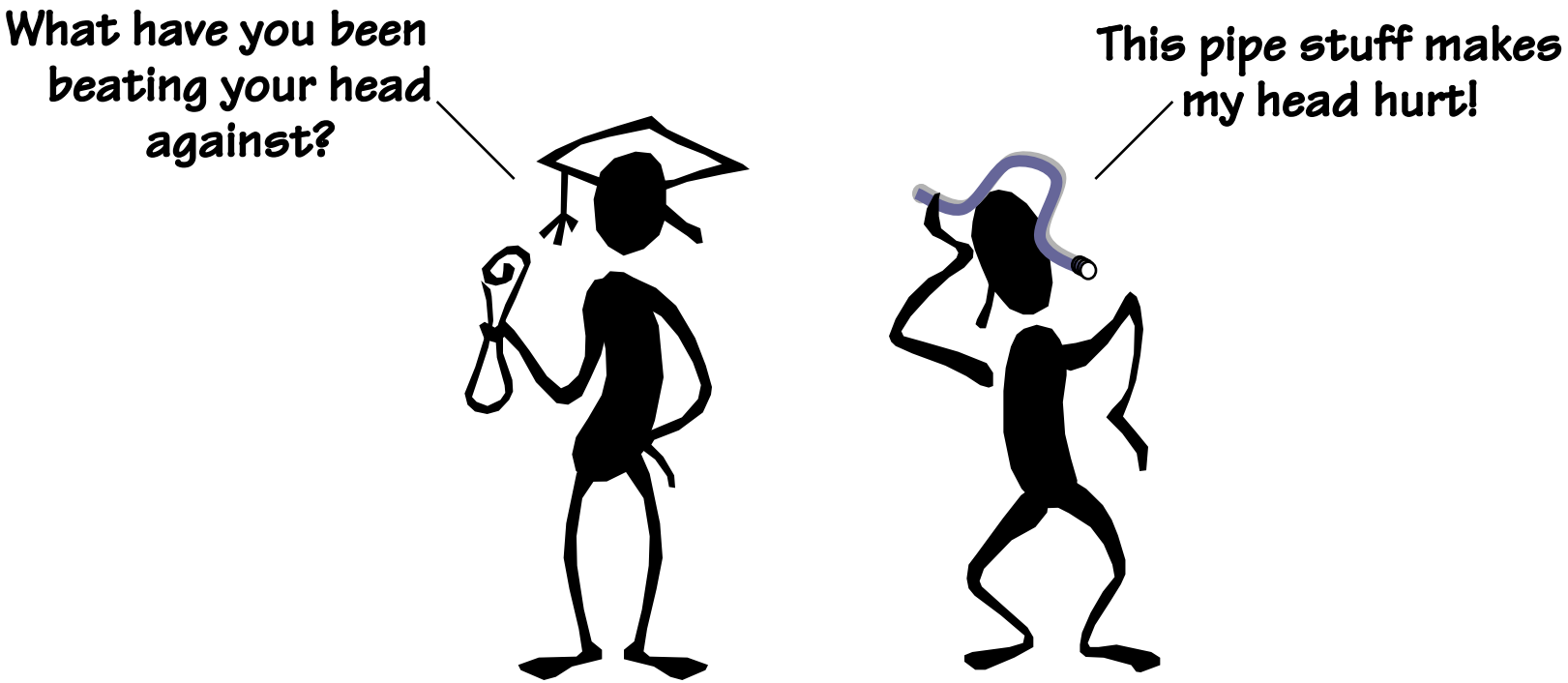
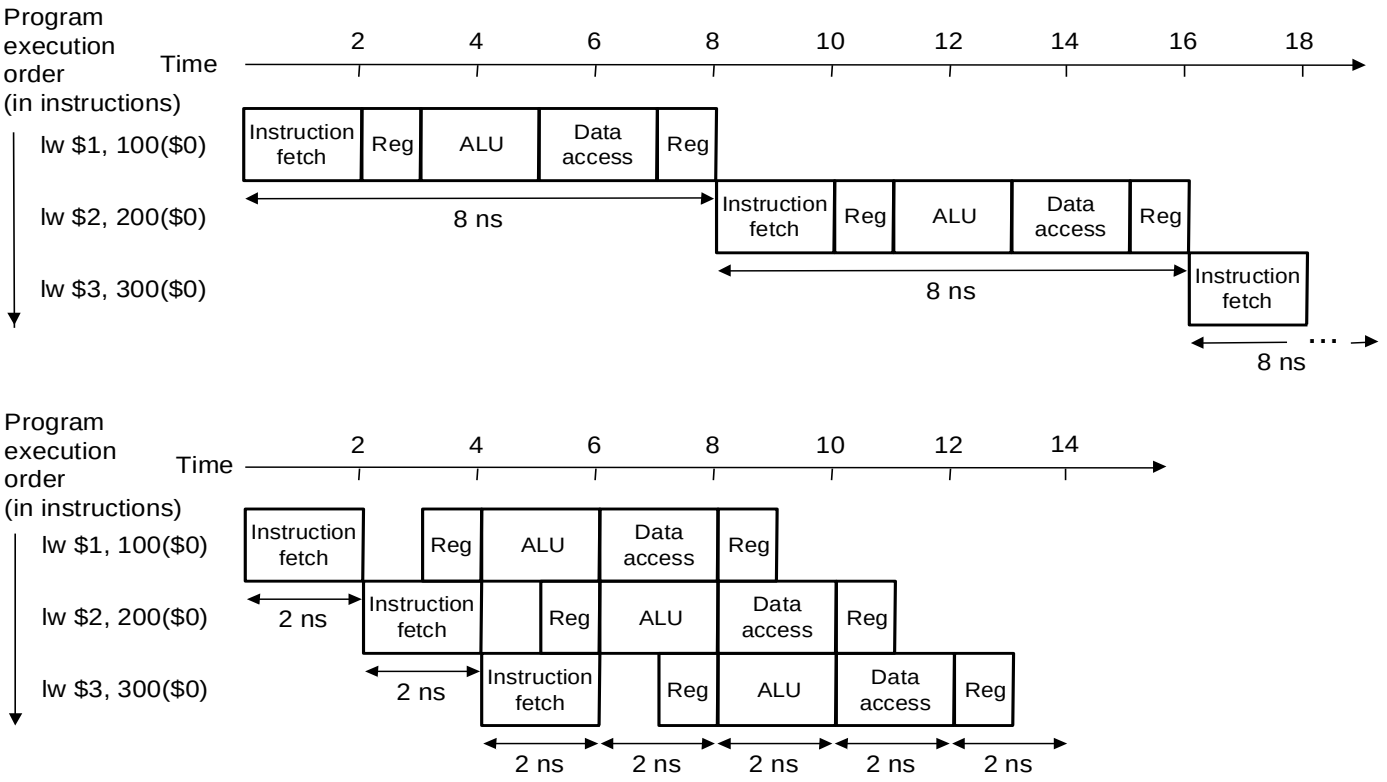
CPU Pipelining Issues



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

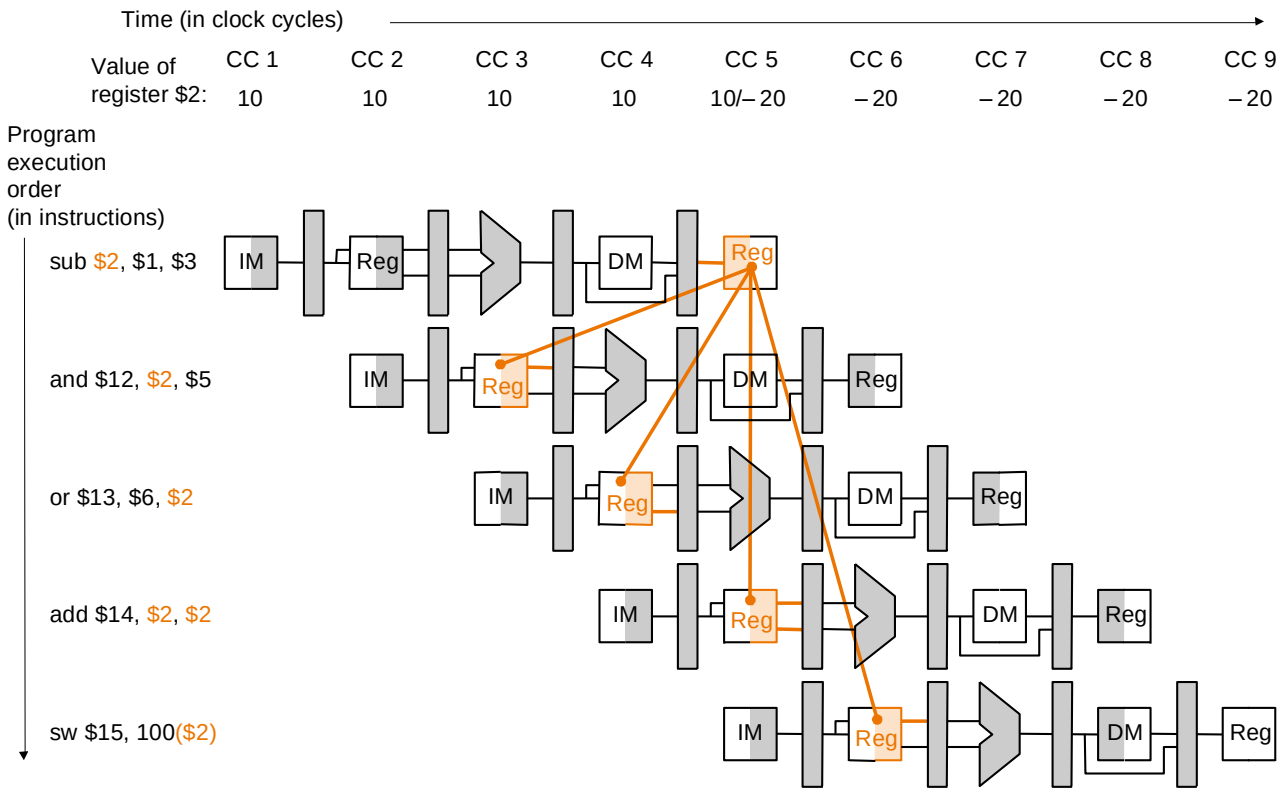
Structural Hazards

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

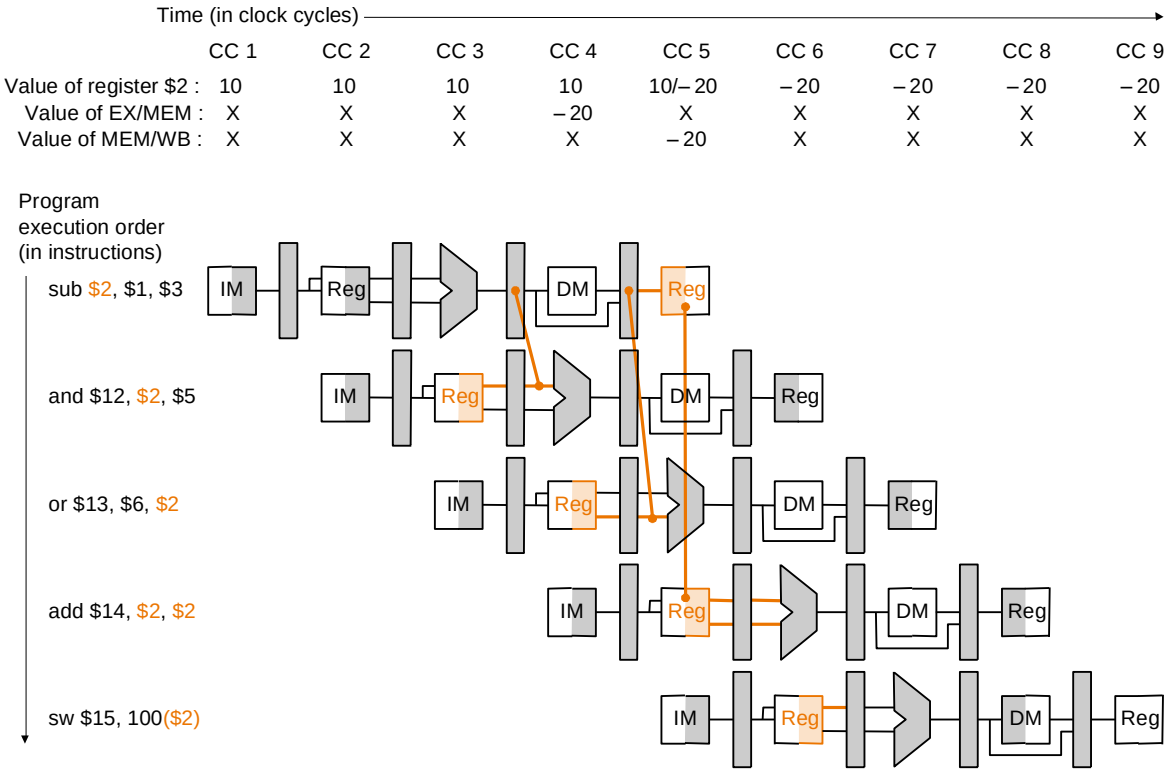
Where do we insert the "nops" ?

```
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

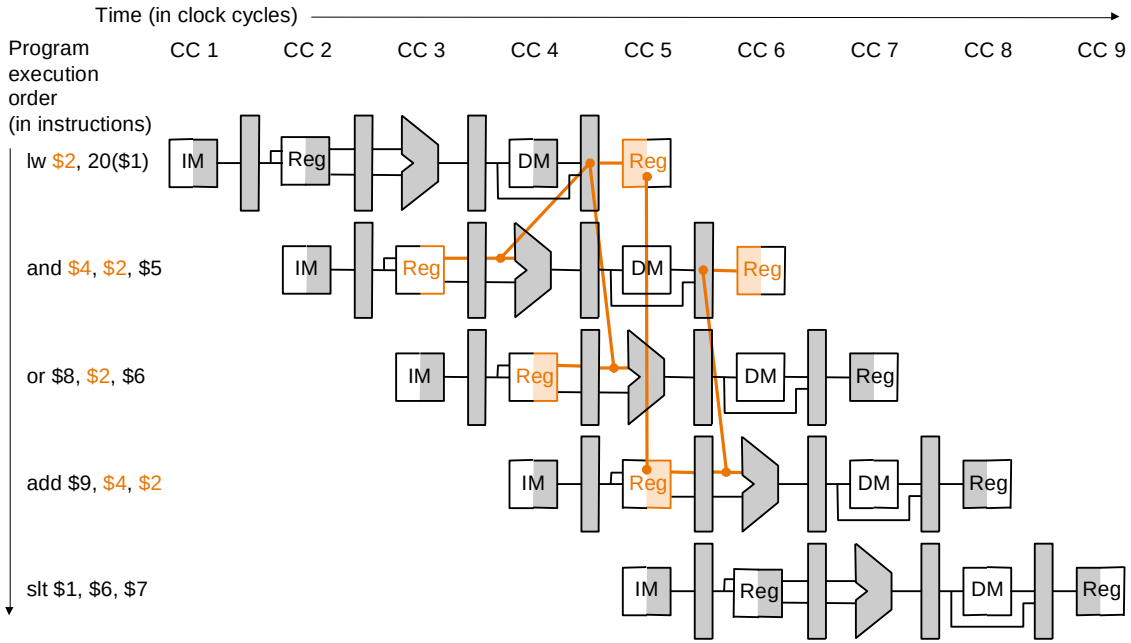
Use temporary results, don't wait for them to be written register file forwarding to handle read/write to same register ALU forwarding



Can't always forward

Load word can still cause a hazard:

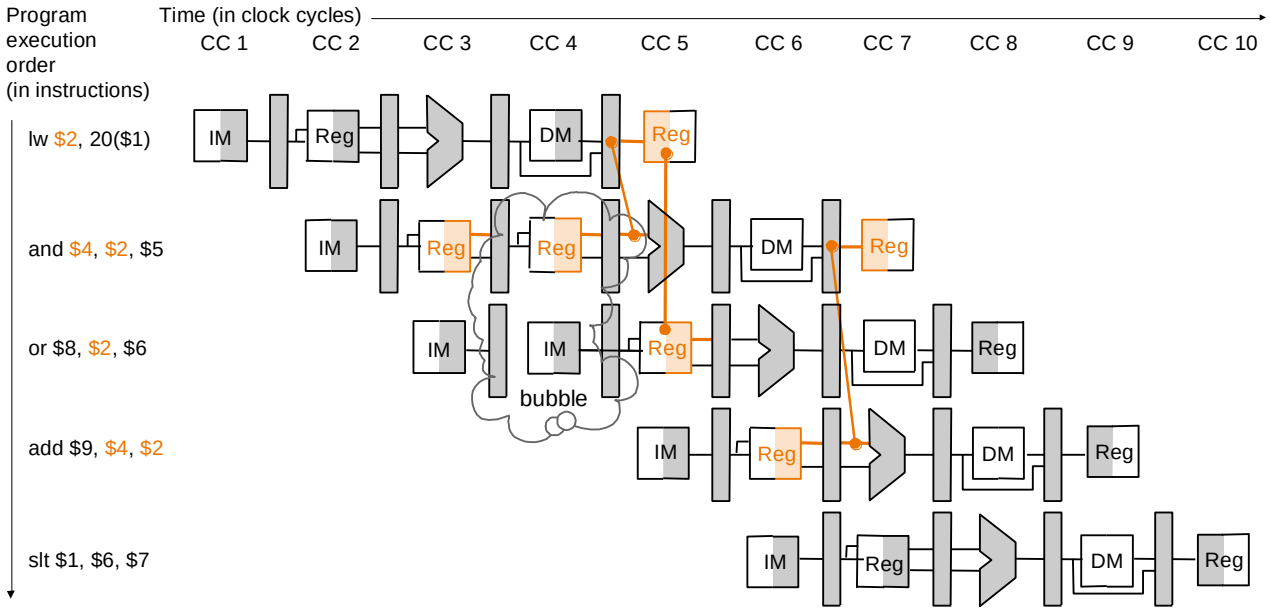
- **an instruction tries to read a register following a load instruction that writes to the same register.**



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

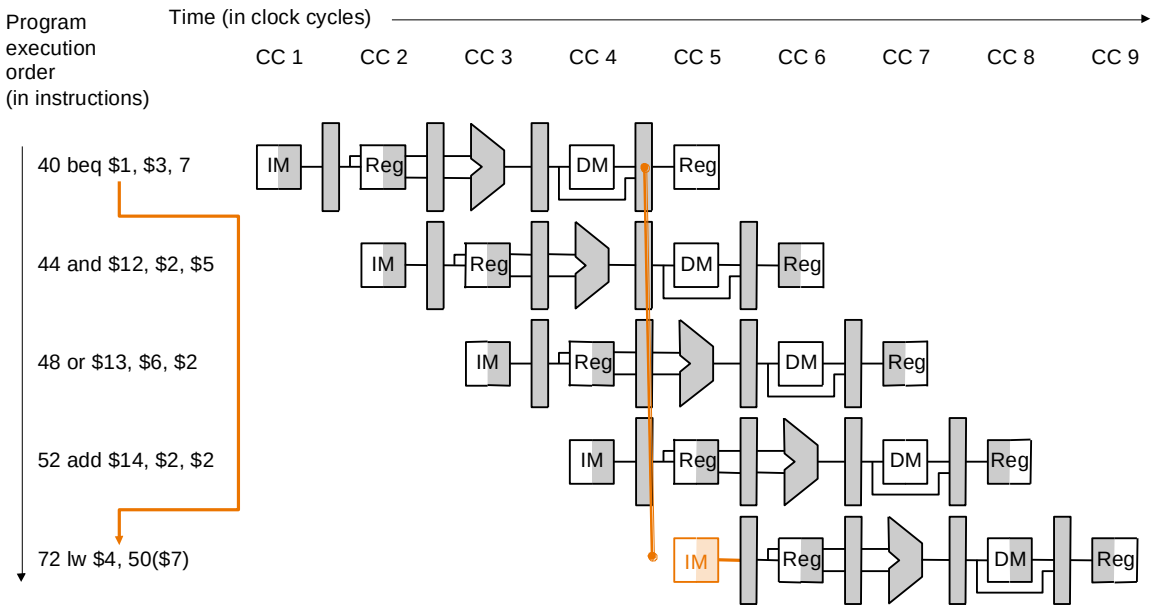
Stalling

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 x 2 = 12) to forward correct value
  - memory data available only in WB stage

    Introduce NOPs at IR^{ALU}, 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).

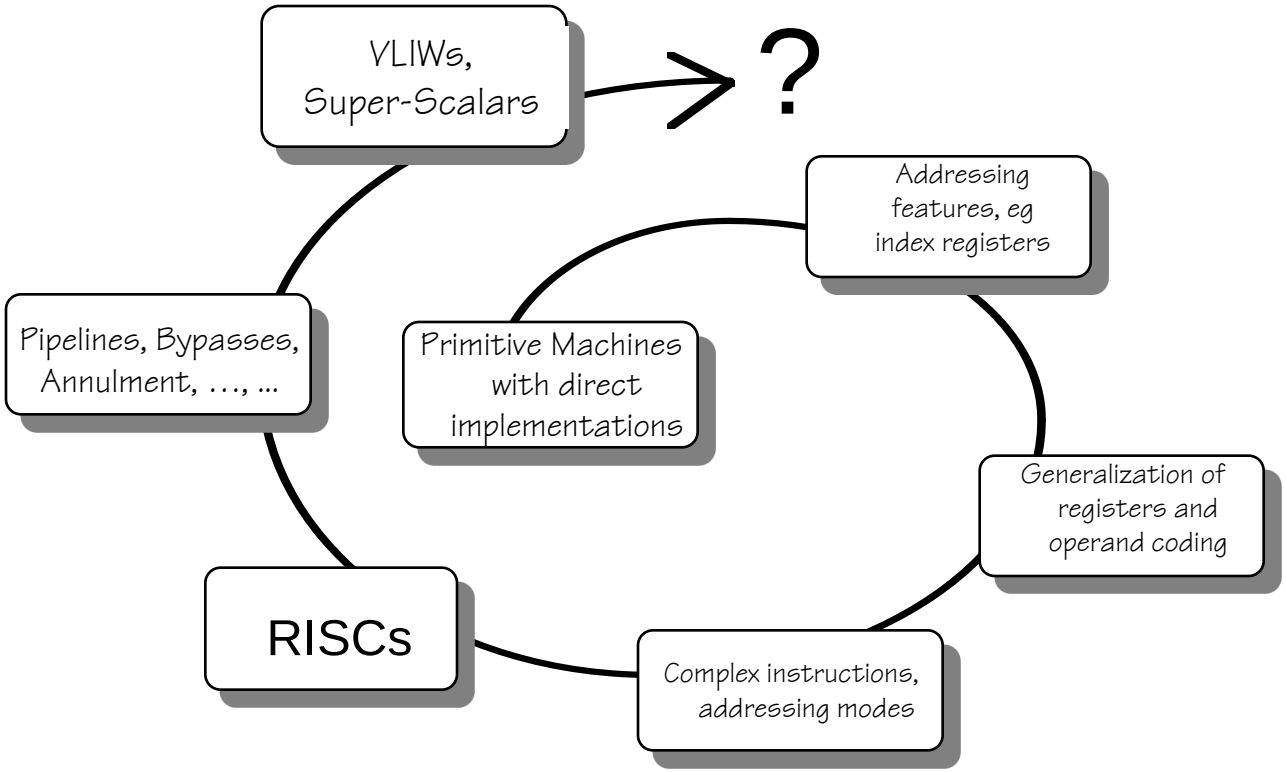
Fallacy #3: Increasing Pipeline stages improves performance

Diminishing returns. Increasing complexity.

RISC = Simplicity???

"The P.T. Barnum World's Tallest Dwarf Competition"

World's Most Complex RISC?

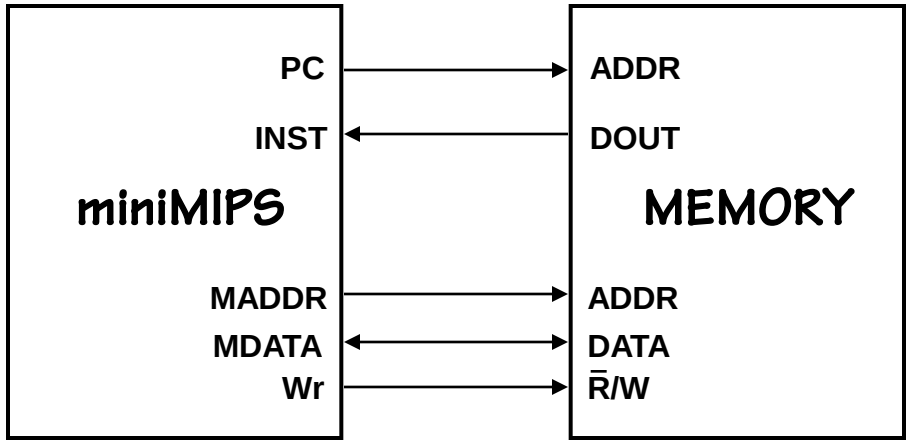


Memory Hierarchy

- Memory Flavors
- Principle of Locality
- Program Traces
- Memory Hierarchies
- Associativity

(Study Chapter 5)

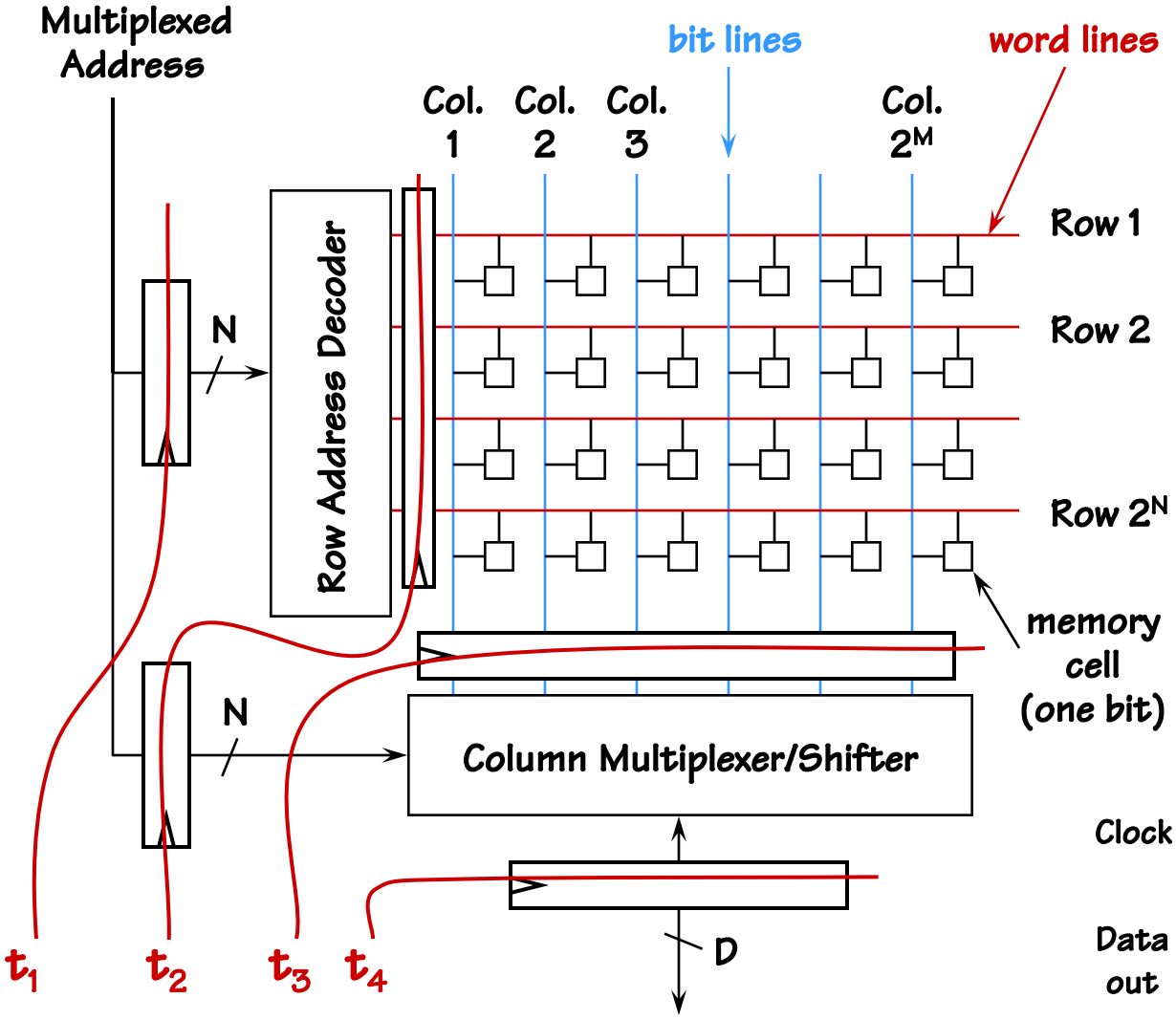
What Do We Want in a Memory?



	Capacity	Latency	Cost
Register	1000's of bits	10 ps	$$$$
SRAM	1's Mbytes	0.2 ns	$$$
DRAM	10's Gbytes	10 ns	$
Hard disk*	10's Tbytes	10 ms	¢
Want?	100 Gbytes	0.2 ns	cheap

* non-volatile

Tricks for Increasing Throughput

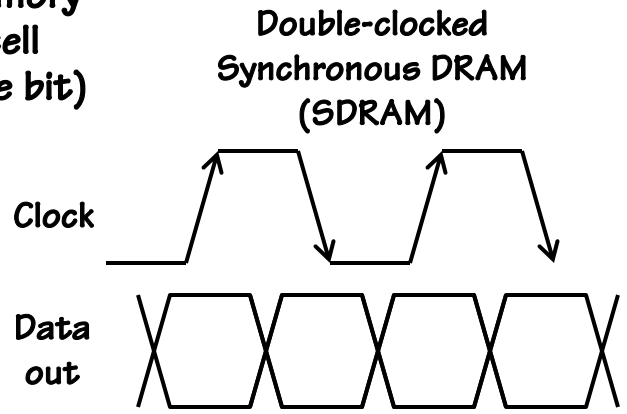


The first thing that should pop into your mind when asked to speed up a digital design…

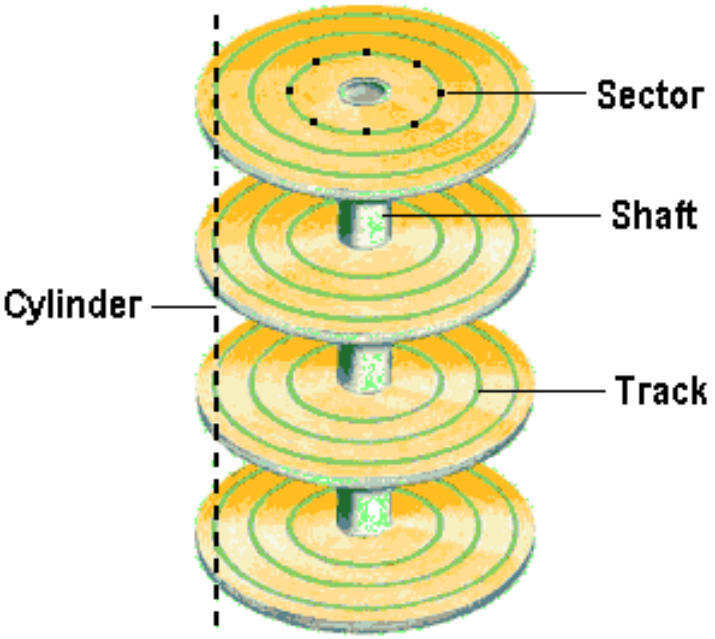
PIPELINING

Synchronous DRAM (SDRAM)

($25 per Gbyte)

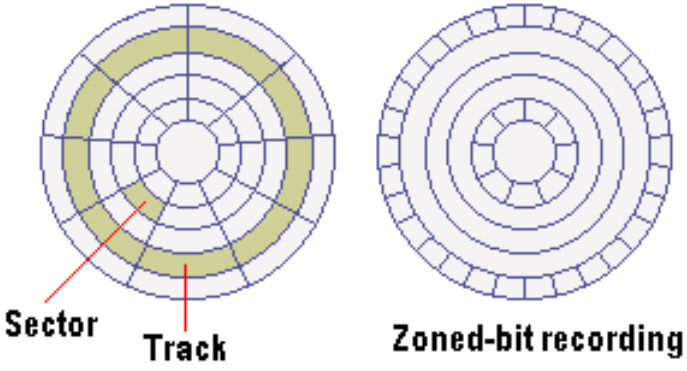


Hard Disk Drives



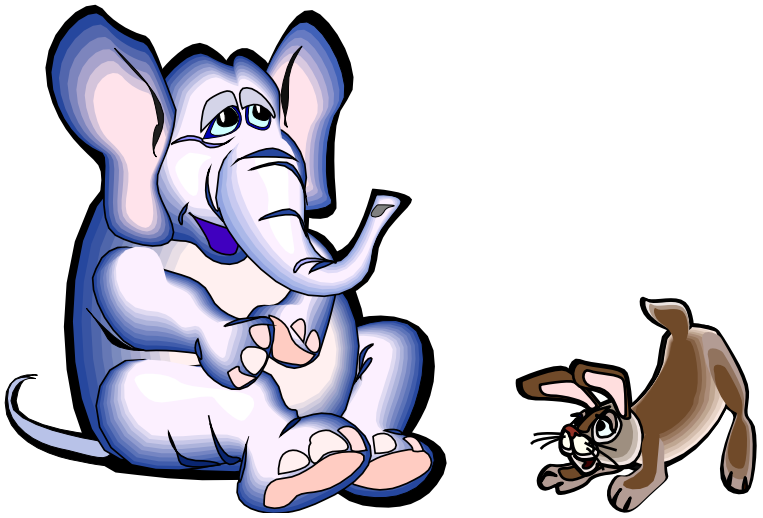
Typical drive:

- Average latency = 4 ms (7200 rpm)
- Average seek time = 8.5 ms
- Transfer rate = 140 Mbytes/s (SATA)
- Capacity = 1.5 T byte
- Cost = $149 (10¢ G byte)



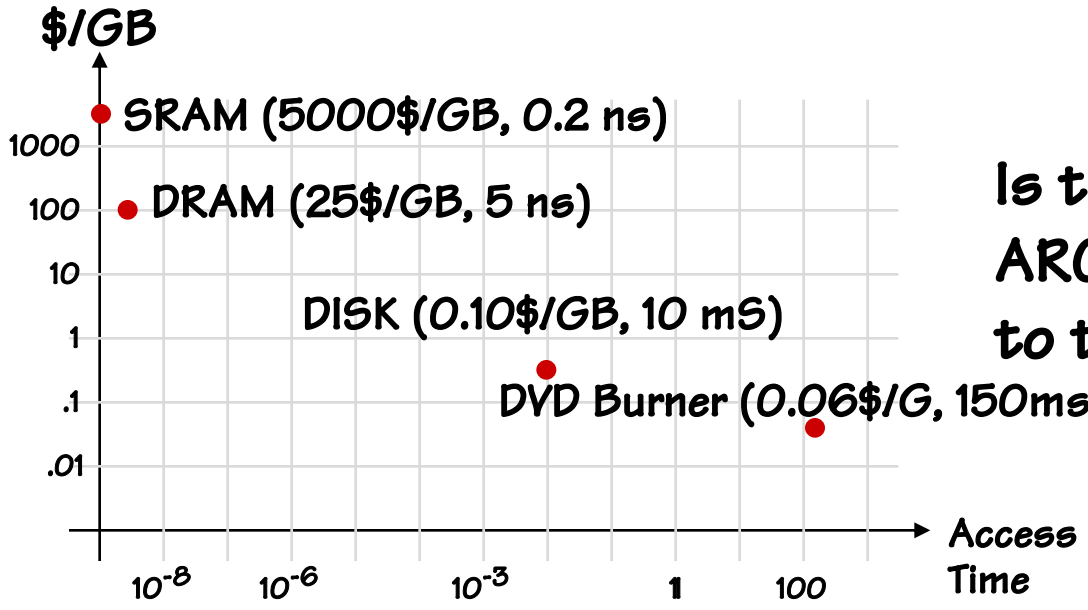
figures from www.pctechguide.com

Quantity vs Quality…



Your memory system can be
- BIG and SLOW… or
- SMALL and FAST.

We've explored a range of device-design trade-offs.



Is there an ARCHITECTURAL solution to this DELIMA?

Managing Memory via Programming

- In reality, systems are built with a mixture of all these various memory types

- How do we make the most effective use of each memory?
- We could push all of these issues off to programmers
  - Keep most frequently used variables and stack in SRAM
  - Keep large data structures (arrays, lists, etc) in DRAM
  - Keep bigger data structures on disk (databases) on DISK
- It is harder than you think… data usage evolves over a program's execution

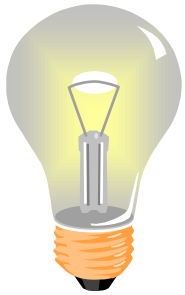
Best of Both Worlds

What we REALLY want: A BIG, FAST memory!
(Keep everything within instant access)

We'd like to have a memory system that
- PERFORMS like 10 GBytes of SRAM; but
- COSTS like 1-4 Gbytes of slow memory.

SURPRISE: We can (nearly) get our wish!

KEY: Use a hierarchy of memory technologies:



Key IDEA

- Keep the most often-used data in a small, fast SRAM (often local to CPU chip)
- Refer to Main Memory only rarely, for remaining data.

The reason this strategy works: LOCALITY

Locality of Reference:

Reference to location X at time t implies that reference to location $X+\Delta X$ at time $t+\Delta t$ becomes more probable as ΔX and Δt approach zero.

Cache

cache **(kash)**

n.

A hiding place used especially for storing provisions.

A place for concealment and safekeeping, as of valuables.

The store of goods or valuables concealed in a hiding place.

Computer Science. **A fast storage buffer in the central processing unit of a computer. In this sense, also called** *cache memory.*

v. tr. *cached, cach·ing, cach·es.*

To hide or store in a cache.

Cache Analogy

You are writing a term paper at a table in the library

As you work you realize you need a book

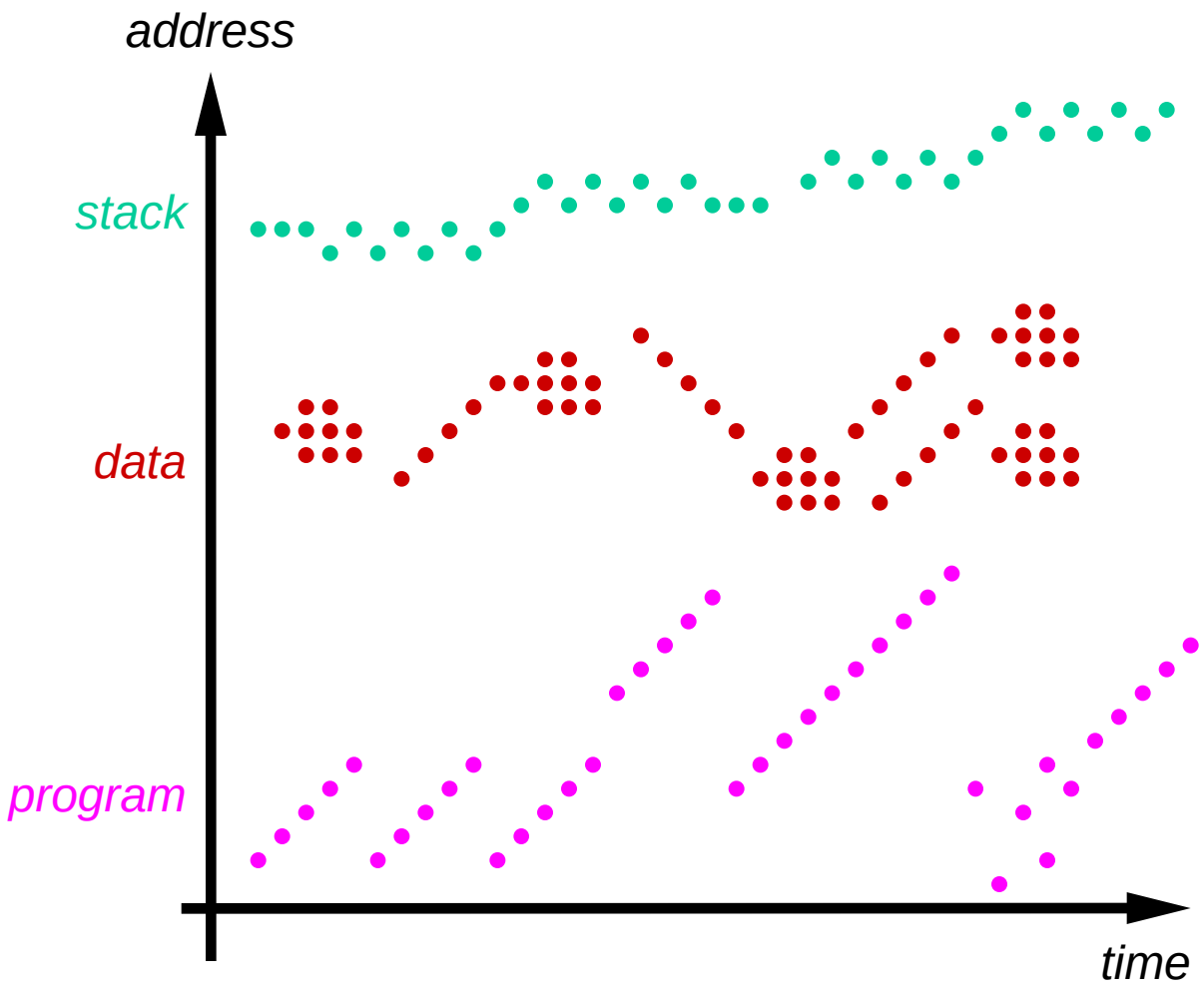
You stop writing, fetch the reference, continue writing

You don't immediately return the book, maybe you'll need it again

Soon you have a few books at your table and no longer have to fetch more books

The table is a CACHE for the rest of the library

Typical Memory Reference Patterns

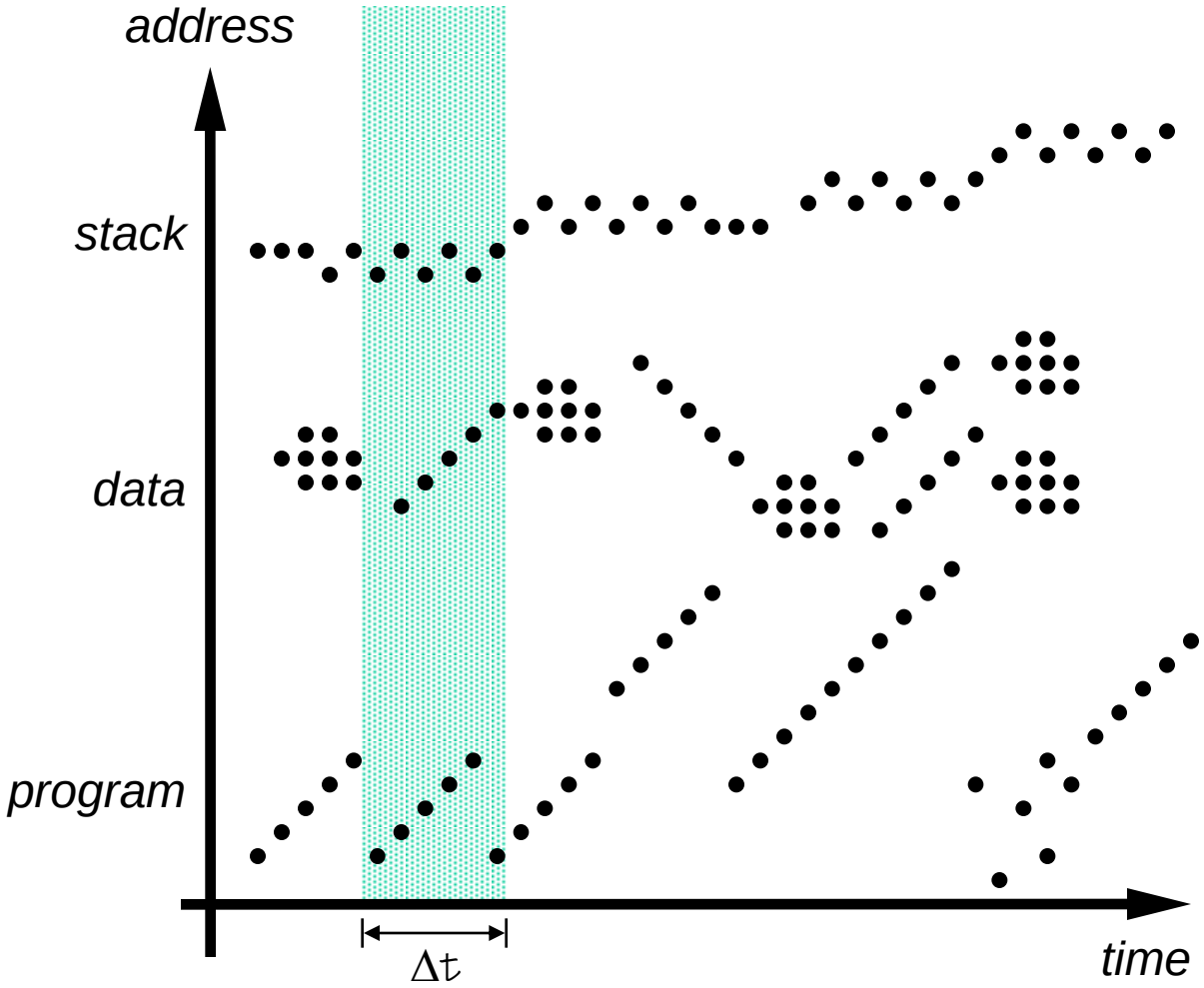


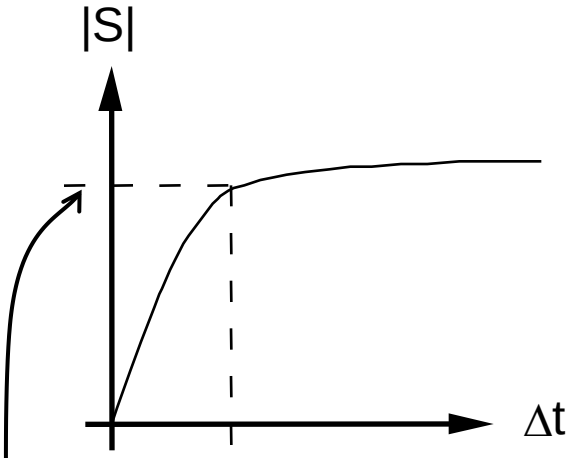
MEMORY TRACE –
A temporal sequence of memory references (addresses) from a real program.

TEMPORAL LOCALITY –
If an item is referenced, it will tend to be referenced again soon

SPATIAL LOCALITY –
If an item is referenced, nearby items will tend to be referenced soon.

Working Set





S is the set of locations accessed during Δt.

Working set: a set S which changes slowly w.r.t. access time.

Working set size, |S|

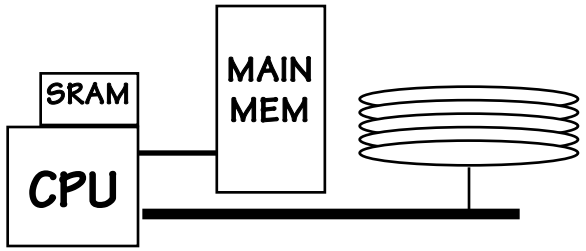
Exploiting the Memory Hierarchy

Approach 1 (Cray, others): Expose Hierarchy

- Registers, Main Memory,

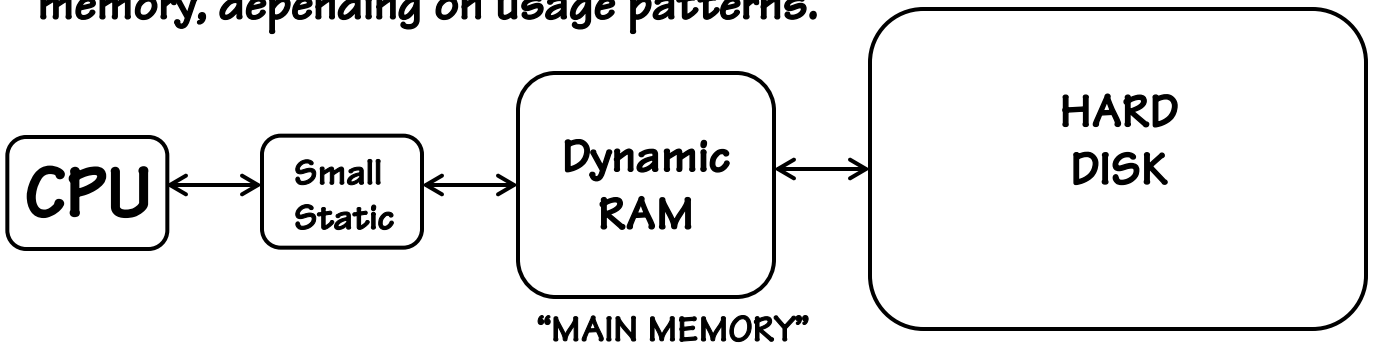
  Disk each available as
  storage alternatives;

- Tell programmers: "Use them cleverly"



Approach 2: Hide Hierarchy

- Programming model: SINGLE kind of memory, single address space.
- Machine AUTOMATICALLY assigns locations to fast or slow memory, depending on usage patterns.

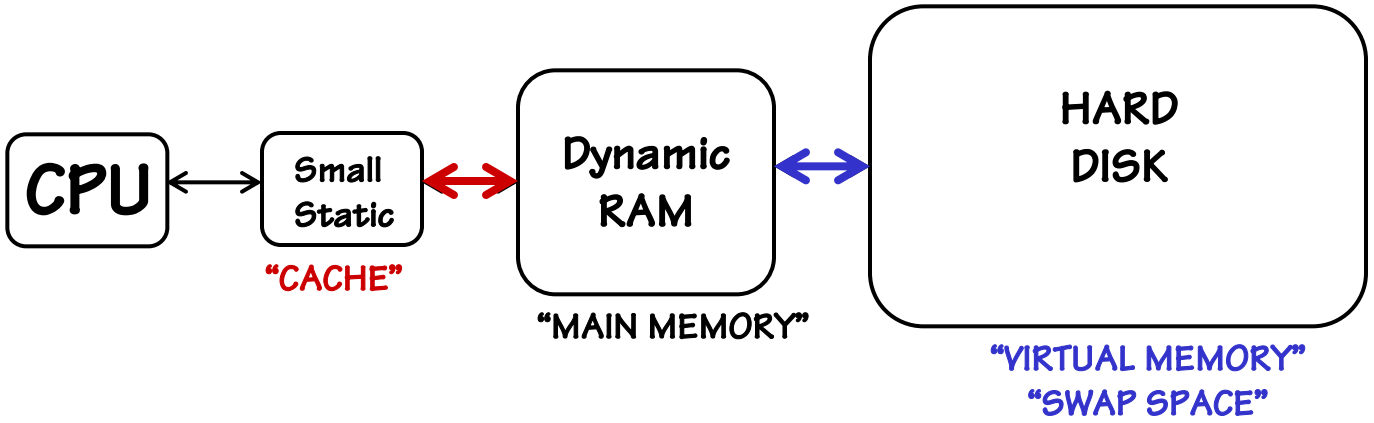


Why We Care

CPU performance is dominated by memory performance.

More significant than:

ISA, circuit optimization, pipelining, super-scalar, etc



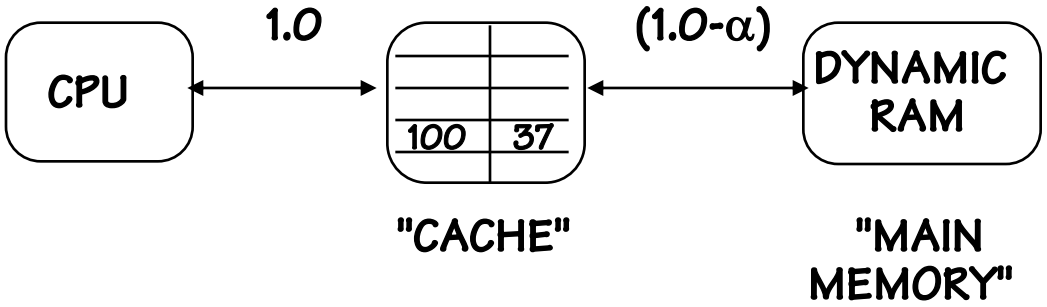
TRICK #1: How to make slow MAIN MEMORY appear faster than it is.

Technique: CACHEING

TRICK #2: How to make a small MAIN MEMORY appear bigger than it is.

Technique: VIRTUAL MEMORY

The Cache Idea:
Program-Transparent Memory Hierarchy



Cache contains TEMPORARY COPIES of selected main memory locations... eg. Mem[100] = 37

GOALS:

1) Improve the average access time

α	HIT RATIO: Fraction of refs found in CACHE.
$(1-\alpha)$	MISS RATIO: Remaining references.

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

Challenge: To make the hit ratio as high as possible.

2) Transparency (compatibility, programming ease)

How High of a Hit Ratio?

Suppose we can easily build an on-chip static memory with a 0.8 nS access time, but the fastest dynamic memories that we can buy for main memory have an average access time of 10 nS. How high of a hit rate do we need to sustain an average access time of 1 nS?

$$\alpha = 1 - \frac{t_{ave} - t_c}{t_m} = 1 - \frac{1 - 0.8}{10} = 98\%$$

WOW, a cache really needs to be good?

Cache

Sits between CPU and main memory

Very fast table that stores a *TAG* and *DATA*

TAG is the memory address

DATA is a copy of memory at the address given by TAG

Tag	Data
1000	17
1040	1
1032	97
1008	11

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Cache Access

On load we look in the TAG entries for the address we're loading

Found → a *HIT*, return the DATA

Not Found → a *MISS*, go to memory for the data and put it and the address (TAG) in the cache

Tag	Data
1000	17
1040	1
1032	97
1008	11

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Cache Lines

Usually get more data than requested (Why?)

a *LINE* is the unit of memory stored in the cache

usually much bigger than 1 word, 32 bytes per line is common

bigger LINE means fewer misses because of spatial locality

but bigger LINE means longer time on miss

Tag	Data	
1000	17	23
1040	1	4
1032	97	25
1008	11	5

Memory

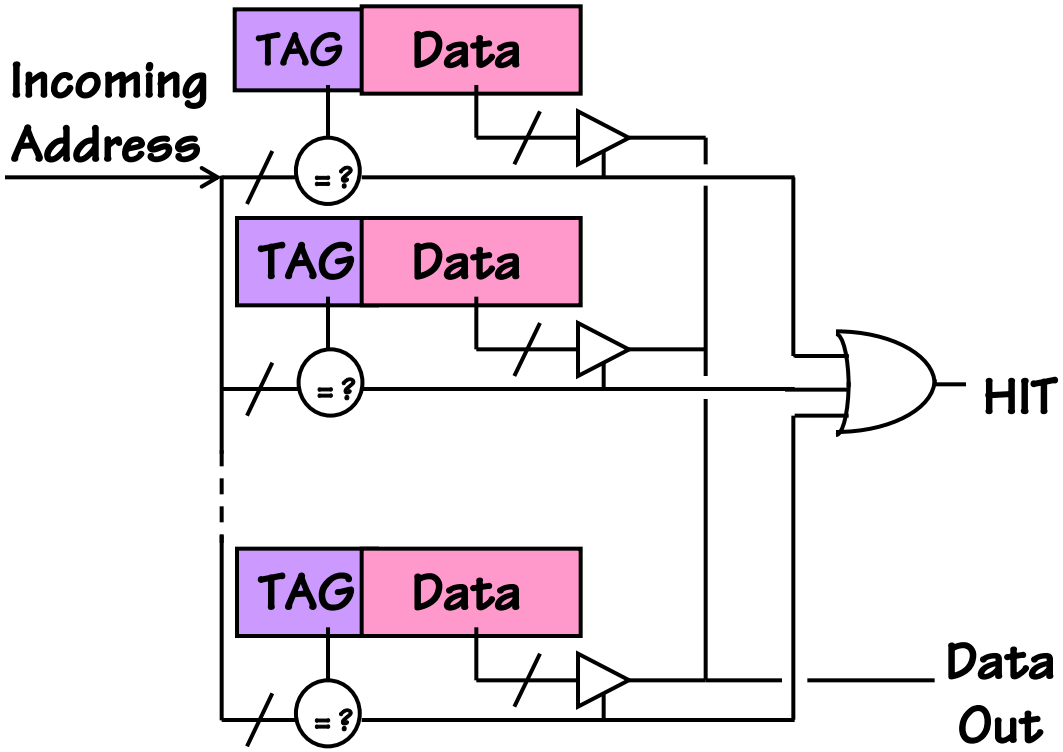
1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	174

Finding the TAG in the Cache

A 1MByte cache may have 32k different lines each of 32 bytes

We can't afford to sequentially search the 32k different tags

ASSOCIATIVE memory uses hardware to compare the address to the tags in parallel but it is expensive and 1MByte is thus unlikely



Finding the TAG in the Cache

A 1MByte cache may have 32k different lines each of 32 bytes

We can't afford to sequentially search the 32k different tags

ASSOCIATIVE memory uses hardware to compare the address to the tags in parallel but it is expensive and 1MByte is thus unlikely

DIRECT MAPPED CACHE computes the cache entry from the address

- multiple addresses map to the same cache line
- use TAG to determine if right

Choose some bits from the address to determine the Cache line

- low 5 bits determine which byte within the line
- we need 15 bits to determine which of the 32k different lines has the data
- which of the 32 – 5 = 27 remaining bits should we use?

Direct-Mapping Example

- With 8 byte lines, the bottom 3 bits determine the byte within the line
- With 4 cache lines, the next 2 bits determine which line to use

`1024d = 10000000000b → line = 00b = 0d`

`1000d = 01111101000b → line = 01b = 1d`

`1040d = 10000010000b → line = 10b = 2d`

Tag	Data	
1024	44	99
1000	17	23
1040	1	4
1016	29	38

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	

Direct Mapping Miss

•What happens when we now ask for address 1008?

`1008d = 01111110000b → line = 10b = 2d`

but earlier we put 1040d there...

`1040d = 10000010000b → line = 10b = 2d`

Tag	Data	
1024	44	99
1000	17	23
1008	11	5
1016	29	38

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	74

Miss Penalty and Rate

The *MISS PENALTY* is the time it takes to read the memory if it isn't in the cache

50 to 100 cycles is common.

The MISS RATE is the fraction of accesses which MISS

The HIT RATE is the fraction of accesses which HIT

MISS RATE + HIT RATE = 1

Suppose a particular cache has a MISS PENALTY of 100 cycles and a HIT RATE of 95%. The CPI for load on HIT is 5 but on a MISS it is 105. What is the average CPI for load?

Average CPI = 10

5 * 0.95 + 105 * 0.05 = 10

Suppose MISS PENALTY = 120 cycles?

then CPI = 11 (slower memory doesn't hurt much)

Some Associativity can help

Direct-Mapped caches are very common but can cause problems...

SET ASSOCIATIVITY can help.

Multiple Direct-mapped caches, then compare multiple TAGS

- 2-way set associative = 2 direct mapped + 2 TAG comparisons
- 4-way set associative = 4 direct mapped + 4 TAG comparisons

Now array size == power of 2 doesn't get us in trouble

But

- slower
- less memory in same area
- maybe direct mapped wins...

What about store?

What happens in the cache on a store?

- WRITE BACK CACHE → put it in the cache, write on replacement
- WRITE THROUGH CACHE → put in cache and in memory

What happens on store and a MISS?

- WRITE BACK will fetch the line into cache
- WRITE THROUGH might just put it in memory