332
Advanced Computer Architecture
Chapter 1

Introduction and review of Pipelines, Performance, Caches, and Virtual Memory

January 2004

Paul H J Kelly

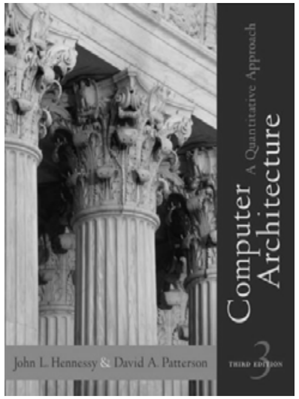
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd ed), and on the lecture slides of David Patterson's Berkeley course (CS252)*

Course materials online at
http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html
username aca, password kelly

Pre-requisites

- This a third-level computer architecture course
- The usual path would be to take this course after following a course based on a textbook like "Computer Organization and Design" (Patterson and Hennessy, Morgan Kaufmann)
- This course is based on the more advanced book by the same authors (see next slide)
- You *can* take this course provided you're prepared to catch up if necessary
  - Read chapters 1 to 8 of "Computer Organization and Design" (COD) if this material is new to you
  - If you have studied computer architecture before, make sure COD Chapters 2, 6, 7 are familiar
  - See also "Appendix A Pipelining: Basic and Intermediate Concepts" of course textbook
- FAST review today of Pipelining, Performance, Caches, and Virtual Memory

This is a textbook-based course



- Computer Architecture: A Quantitative Approach (3rd Edition)

  John L. Hennessy, David A. Patterson

  - 1128 pages. Morgan Kaufmann (29 May, 2002); ISBN: 1558607242
  - http://www.elsevier-international.com/catalogue/title.cfm?ISBN=1558607242
  - Price: £ 33.95 (Amazon.co.uk, Jan 2004)

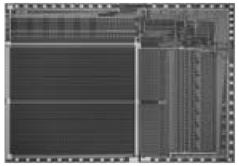
Who are these guys anyway and why should I read their book?

- John Hennessy:
  - Founder, MIPS Computer Systems
  - President, Stanford University

    (previous president: Condoleezza Rice)

RAID-I (1989) consisted of a Sun 4/280 workstation with 128 MB of DRAM, four dual-string SCSI controllers, 28 5.25-inch SCSI disks and specialized disk striping software.

- David Patterson
  - Leader, Berkeley RISC project (led to Sun's SPARC)
  - RAID (redundant arrays of inexpensive disks)
  - Professor, University of California, Berkeley



http://www.cs.berkeley.edu/~pattrsn/Arch/prototypes2.html

RISC-I (1982) Contains 44,420 transistors, fabbed in 5 micron NMOS, with a die area of 77 mm^2, ran at 1 MHz. This chip is probably the first VLSI RISC.

Administrivia

- **Course web site:**
  - **http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html**
  - **See web site for link to course news group:**
    - **news:icdoc.courses.332aca**.
  - **Course textbook: H&P 3rd ed**
    - **Read Appendix A *right away***

Course organisation

- Lecturers:
  - **Paul Kelly** – first few weeks
  - **Jeremy Bradley** – last couple of weeks
- Tutorial helper:
  - **Jeyarajan Thiyagalingam** (Jeyan)
- 3 hours per week
- Nominally two hours of lectures, one hour of classroom tutorials
- We will use the time more flexibly
- Assessment:
  - Exam
    - For CS M.Eng. Class, exam will take place on last Monday of term
    - For everyone else, exam will take place early in the summer term
    - The goal of the course is to teach you how to think about computer architecture
    - The exam usually includes some architectural ideas not presented in the lectures
  - Coursework
    - You will be assigned a substantial, laboratory-based exercise
    - You will learn about performance tuning for computationally-intensive kernels
    - You will learn about using simulators, and experimentally evaluating hypotheses to understand system performance
    - Use the machines in Studio A to get started and get help during tutorials
- Please do not use the computers for anything else during classes

A "Typical" RISC

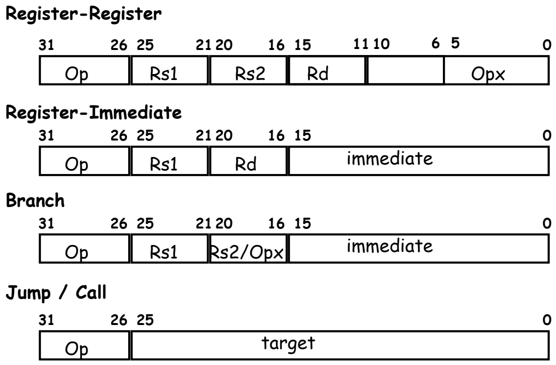
- 32-bit fixed format instruction (3 formats, see next slide)
- 32 32-bit general-purpose registers
  - (R0 contains zero, double-precision/long operands occupy a pair)
- Memory access only via load/store instructions
  - No instruction both accesses memory and does arithmetic
  - All arithmetic is done on registers
- 3-address, reg-reg arithmetic instruction
  - Subw r1,r2,r3 means r1 := r2-r3
  - registers identifiers always occupy same bits of instruction encoding
- Single addressing mode for load/store: base + displacement
  - ie register contents are added to constant from instruction word, and used as address, eg “lw R2,100(r1)” means “r2 := Mem[100+r1]”
  - no indirection
- Simple branch conditions
- Delayed branch

see: **SPARC, MIPS, ARM, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3**

Not: **Intel IA-32, IA-64 (?), Motorola 68000, DEC VAX, PDP-11, IBM 360/370**

Eg: **VAX matchc instruction!**

Example: MIPS (Note register location)


Register-Register

31 26 25 21 20 16 15 11 10 6 5 0

Op | Rs1 | Rs2 | Rd | | Opx

Register-Immediate

31 26 25 21 20 16 15 0

Op | Rs1 | Rd | immediate

Branch

31 26 25 21 20 16 15 0

Op | Rs1 | Rs2/Opx | immediate

Jump / Call

31 26 25 0

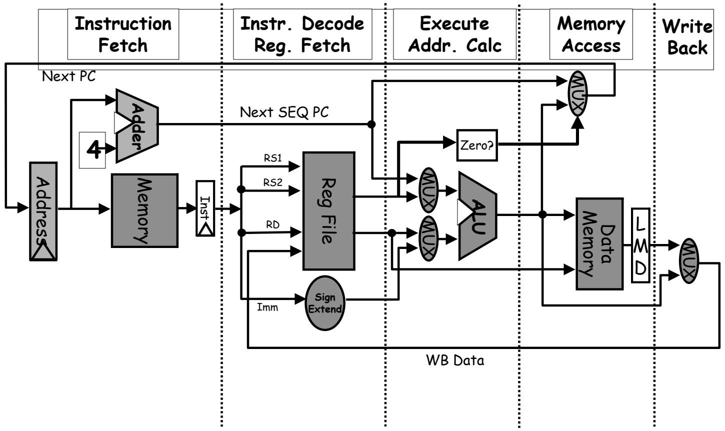
Op | target

Q: What is the largest signed immediate operand for “subw r1,r2,X”?

Q: To what range of addresses can a conditional branch jump to?

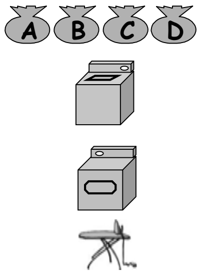
5 Steps of MIPS Datapath

Figure 3.1, Page 130, CA:AQA 2e

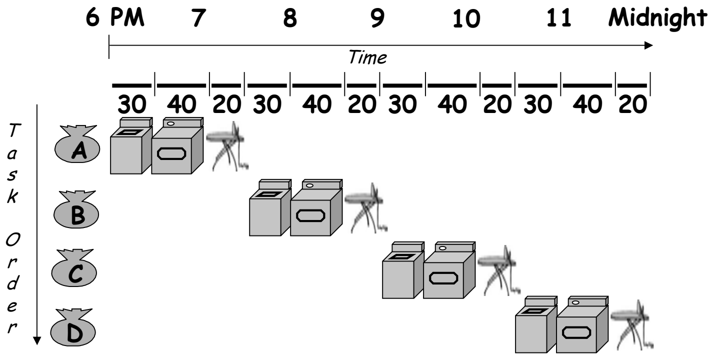


Pipelining: A very familiar idea…

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and iron
- Washer takes 30 minutes
- Dryer takes 40 minutes
- Ironing takes 20 minutes

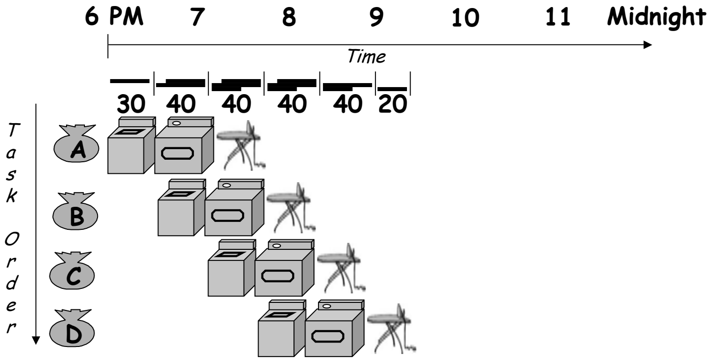


Sequential Laundry



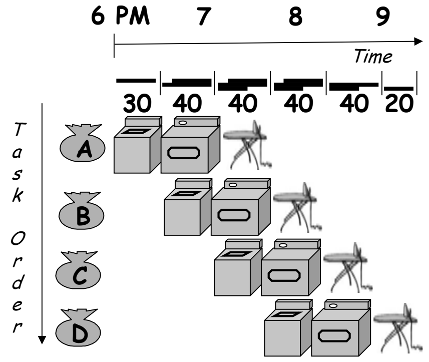
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Principle: everyone starts work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Pipelined Laundry: Lessons-

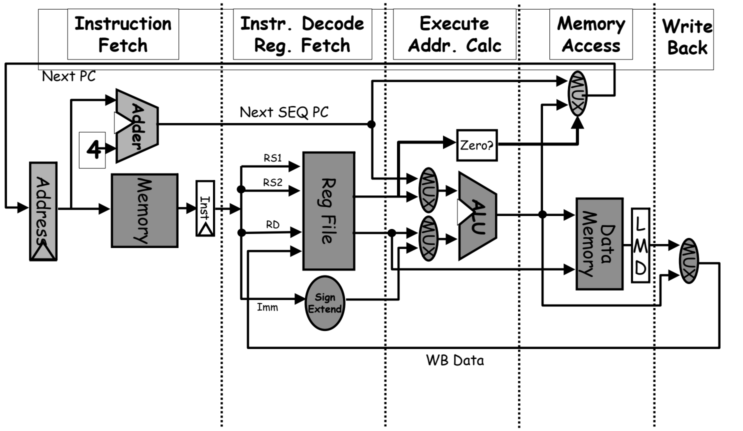


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Speedup comes from parallelism
  - For free – no new hardware

- Pipelined laundry takes 3.5 hours for 4 loads

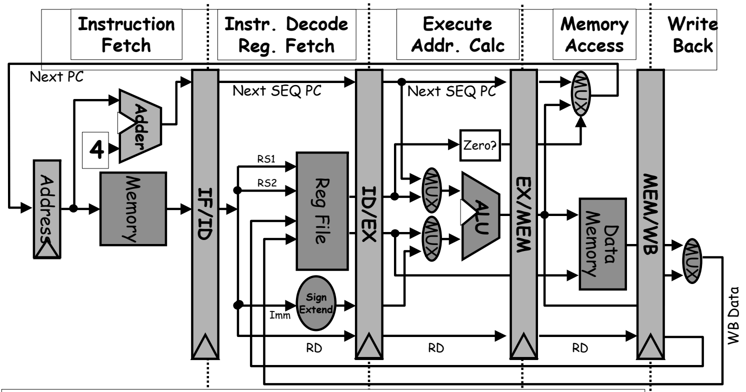
How can we apply this idea to the MIPS datapath?

Figure 3.1, Page 130, *CA:AQA 2e*



5-stage MIPS pipeline with pipeline buffers

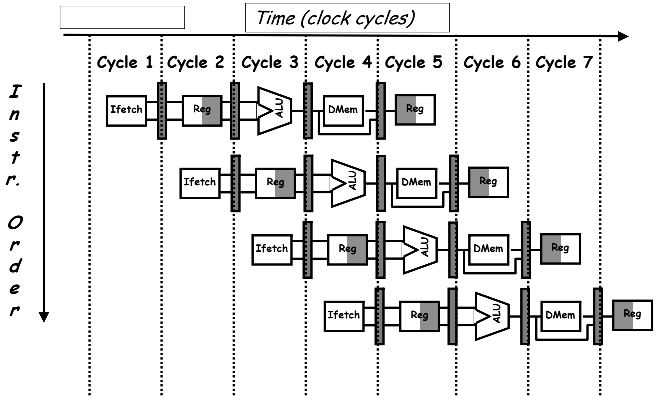
Figure 3.4, Page 134 , *CA:AQA 2e*



- Data stationary control
  - local decode for each instruction phase / pipeline stage

Visualizing Pipelining

Figure 3.3, Page 133 , *CA:AQA 2e*

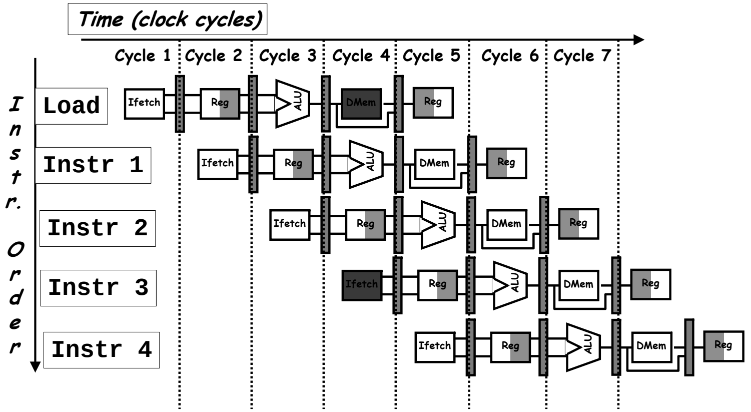


It's Not That Easy for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

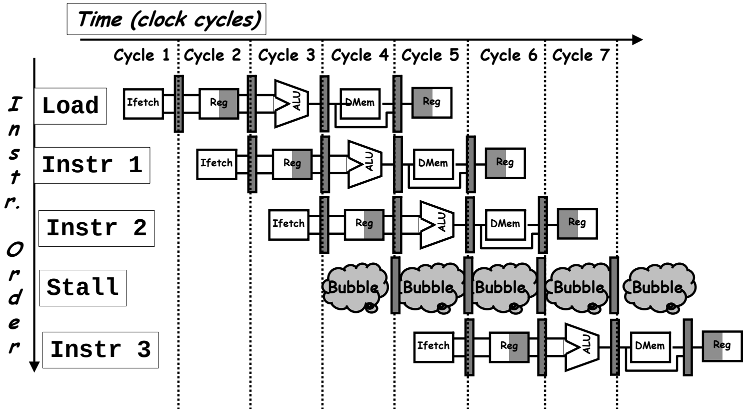
One Memory Port/Structural Hazards

Figure 3.6, Page 142 , *CA:AQA 2e*



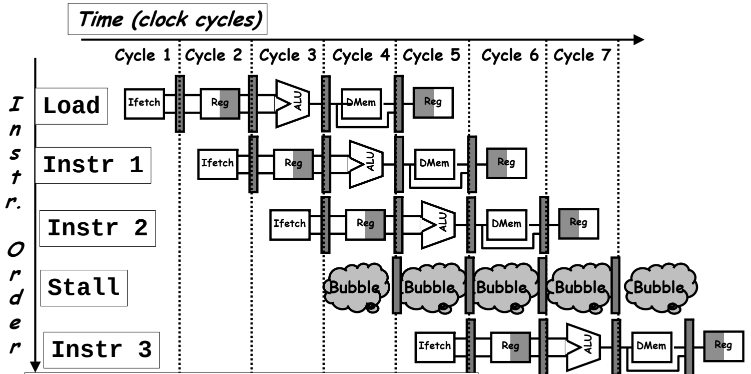
One Memory Port/Structural Hazards

Figure 3.7, Page 143 , *CA:AQA 2e*



One Memory Port/Structural Hazards

Figure 3.7, Page 143 , *CA:AQA 2e*



"bubble" propagates through pipeline – at each subsequent cycle one of the pipeline stages lies idle

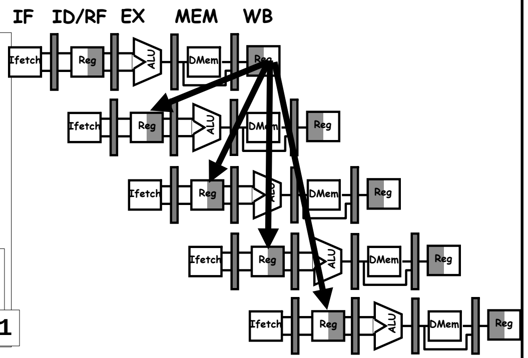
Data Hazard on R1

Figure 3.9, page 147 , CA:AQA 2e

Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```



Three Generic Data Hazards

- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- Write After Read (WAR)
  Instr$_J$ writes operand *before* Instr$_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Three Generic Data Hazards

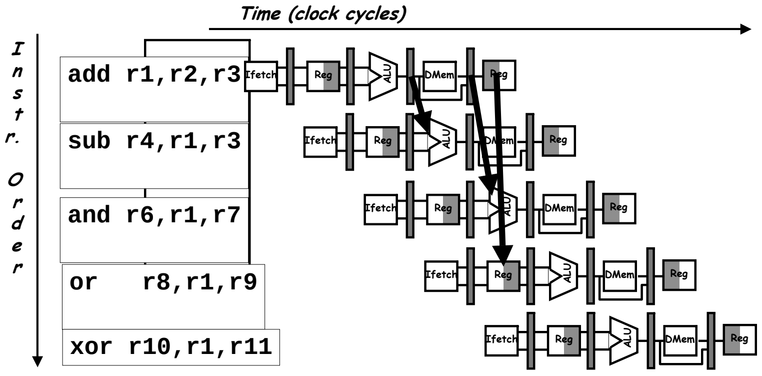
- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

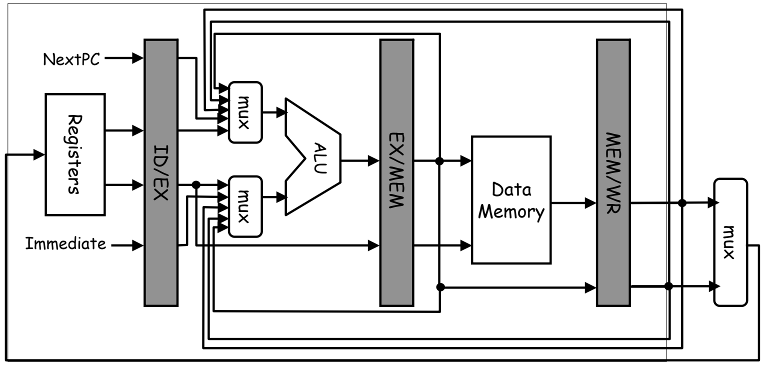
Forwarding to Avoid Data Hazard

Figure 3.10, Page 149 , CA:AQA 2e



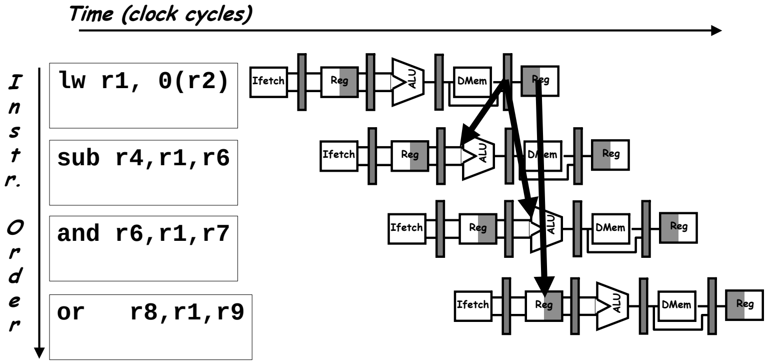
HW Change for Forwarding

Figure 3.20, Page 161, CA:AQA 2e



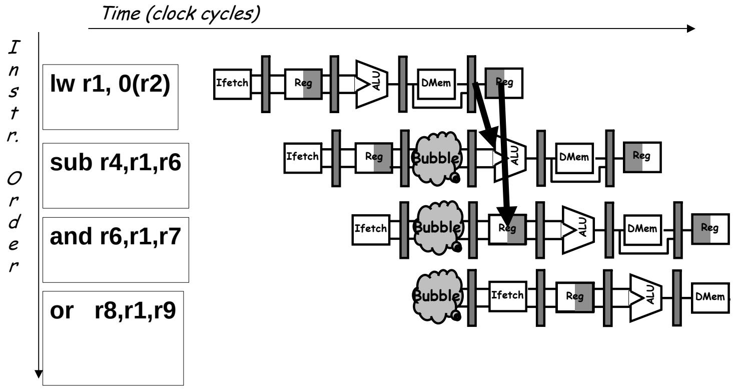
Data Hazard Even with Forwarding

Figure 3.12, Page 153 , CA:AQA 2e



Data Hazard Even with Forwarding

Figure 3.13, Page 154 , CA:AQA 2e



Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

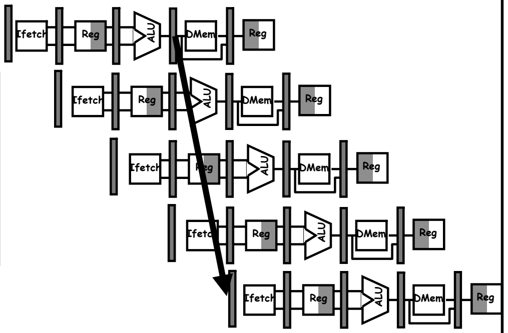
d = e – f;

assuming a, b, c, d ,e, and f in memory.

```
Slow code:                  Fast code:
     LW    Rb,b                  LW    Rb,b
     LW    Rc,c                  LW    Rc,c
     ADD   Ra,Rb,Rc              LW    Re,e
     SW    a,Ra                  ADD   Ra,Rb,Rc
     LW    Re,e                  LW    Rf,f
     LW    Rf,f                  SW    a,Ra
     SUB   Rd,Re,Rf              SUB   Rd,Re,Rf
     SW    d,Rd                  SW    d,Rd
```

Control Hazard on Branches Three Stage Stall

```
10: beq r1,r3,36
14: and r2,r3,r5
18: or  r6,r1,r7
22: add r8,r1,r9
36: xor r10,r1,r11
```

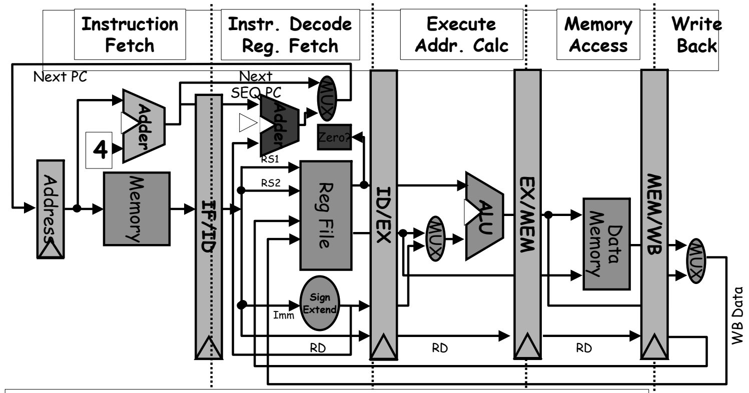


Example: Branch Stall Impact

- If 30% branch, Stall 3 cycles significant
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Pipelined MIPS Datapath with early branch determination

Figure 3.22, page 163, *CA:AQA 2/e*



- Data stationary control
  - local decode for each instruction phase / pipeline stage

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER a following instruction

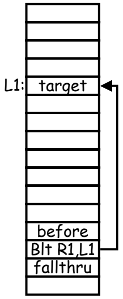
```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)



- Canceling branches
  - Branch delay slot instruction is executed but write-back is disabled if it is not supposed to be executed
  - Two variants: branch "likely taken", branch "likely not-taken"
  - allows more slots to be filled

Now, review basic performance issues in processor design

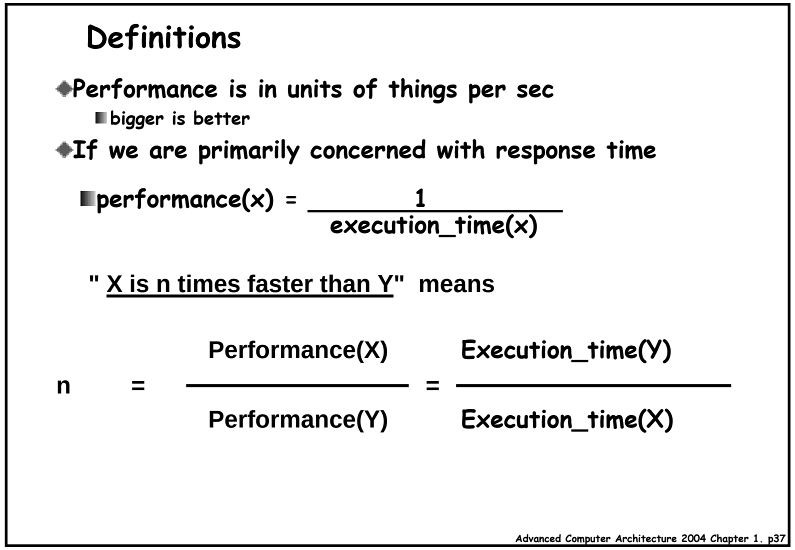
Which is faster?

Plane	Washington DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAC/Sud Concorde	3 hours	1350 mph	132	178,200

First flew in February, 1969

First flew in

- Time to run the task (ExTime)
  - Execution time, response time, latency
- Tasks per day, hour, week, sec, ns … (Performance)
  - Throughput, bandwidth

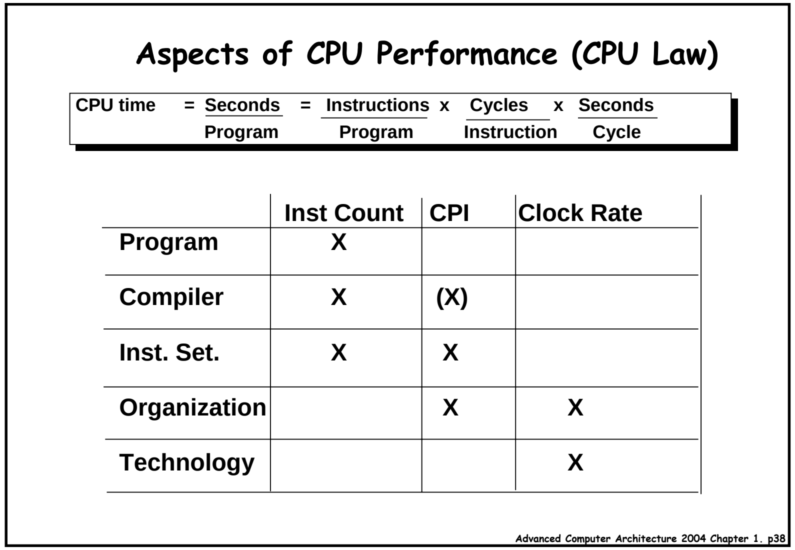


Definitions

- Performance is in units of things per sec
  - bigger is better
- If we are primarily concerned with response time
  - $\text{performance}(x) = \dfrac{1}{\text{execution\_time}(x)}$

" X is n times faster than Y" means

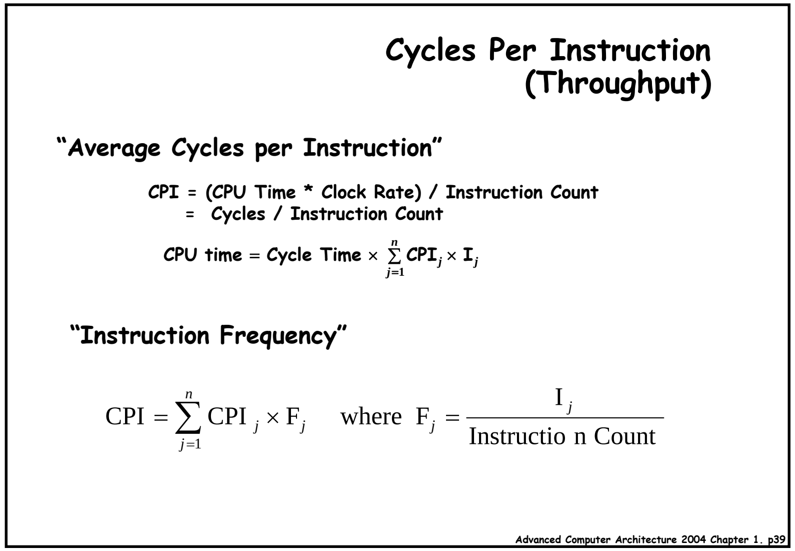
$$n = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{Execution\_time}(Y)}{\text{Execution\_time}(X)}$$



Aspects of CPU Performance (CPU Law)

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization		X	X
Technology			X



Cycles Per Instruction (Throughput)

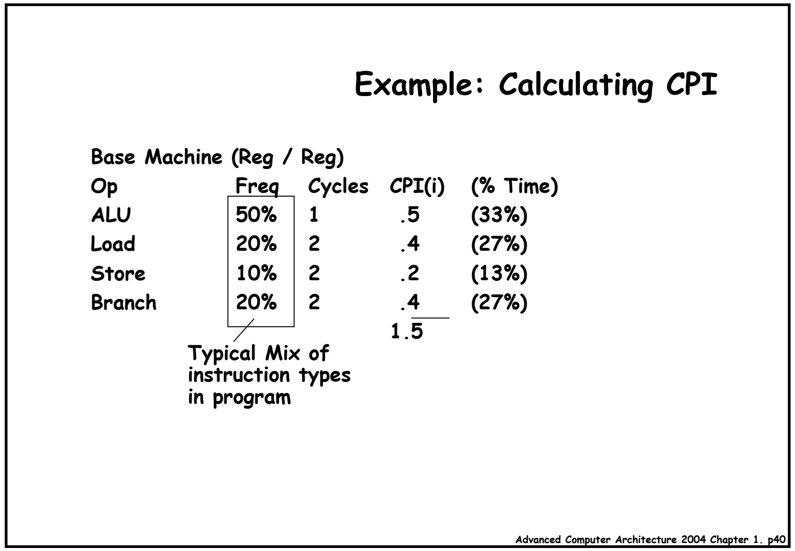
“Average Cycles per Instruction”

CPI = (CPU Time * Clock Rate) / Instruction Count
= Cycles / Instruction Count

$$\text{CPU time} = \text{Cycle Time} \times \sum_{j=1}^{n} \text{CPI}_j \times \text{I}_j$$

“Instruction Frequency”

$$\text{CPI} = \sum_{j=1}^{n} \text{CPI}_j \times \text{F}_j \quad \text{where } \text{F}_j = \frac{\text{I}_j}{\text{Instructio n Count}}$$



Example: Calculating CPI

Base Machine (Reg / Reg)

Op	Freq	Cycles	CPI(i)	(% Time)
ALU	50%	1	.5	(33%)
Load	20%	2	.4	(27%)
Store	10%	2	.2	(13%)
Branch	20%	2	.4	(27%)
			1.5	

Typical Mix of instruction types in program

Example: Branch Stall Impact

- Assume CPI = 1.0 ignoring branches
- Assume solution was stalling for 3 cycles
- If 30% branch, Stall 3 cycles

Op	Freq	Cycles	CPI(i)	(% Time)
Other	70%	1	.7	(37%)
Branch	30%	4	1.2	(63%)

- => new CPI = 1.9, or almost 2 times slower

Example 2: Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, Ideal CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

Example 3: Evaluating Branch Alternatives (for 1 program)

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. stall*
Stall pipeline	3	1.42	1.0
Predict taken	1	1.14	1.26
Predict not taken	1	1.09	1.29
Delayed branch	0.5	1.07	1.31

Assuming Conditional & Unconditional branches make up 14% of the total instruction count, and 65% of them change the PC

Example 4: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

SpeedUp_A = Pipeline Depth/(1 + 0) x ($\text{clock}_{unpipe}/\text{clock}_{pipe}$)
= Pipeline Depth

SpeedUp_B = Pipeline Depth/(1 + 0.4 x 1) x ($\text{clock}_{unpipe}/(\text{clock}_{unpipe}$ / 1.05)
= (Pipeline Depth/1.4) x 1.05
= 0.75 x Pipeline Depth

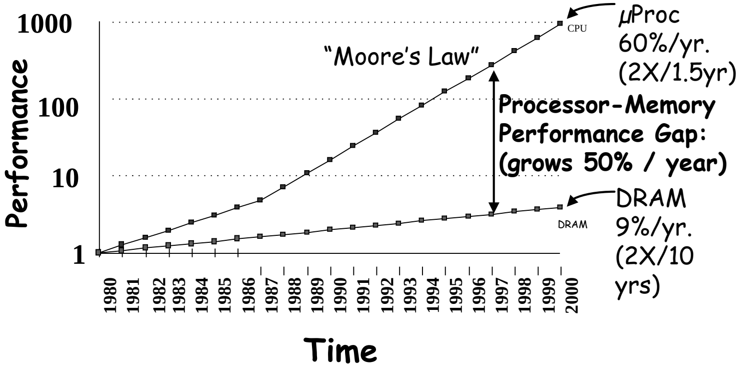
SpeedUp_A / SpeedUp_B = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

- Machine A is 1.33 times faster

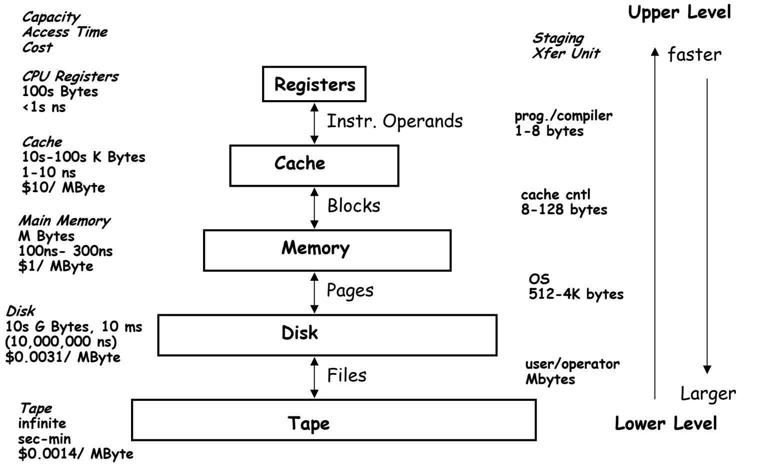
Now, Review of Memory Hierarchy

Recap: Who Cares About the Memory Hierarchy?

Processor-DRAM Memory Gap (latency)



Levels of the Memory Hierarchy



Capacity
Access Time
Cost

CPU Registers
100s Bytes
<1s ns

Cache
10s-100s K Bytes
1-10 ns
$10/ MByte

Main Memory
M Bytes
100ns- 300ns
$1/ MByte

Disk
10s G Bytes, 10 ms
(10,000,000 ns)
$0.0031/ MByte

Tape
infinite
sec-min
$0.0014/ MByte

Registers

Instr. Operands

Cache

Blocks

Memory

Pages

Disk

Files

Tape

Staging
Xfer Unit

prog./compiler
1-8 bytes

cache cntl
8-128 bytes

OS
512-4K bytes

user/operator
Mbytes

Upper Level

faster

Larger

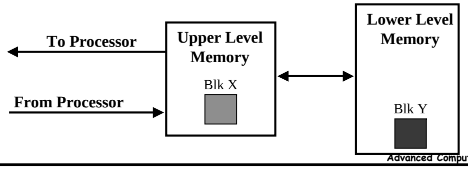
Lower Level

The Principle of Locality

- The Principle of Locality:
  - Programs access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- In recent years, architectures have become increasingly reliant (totally reliant?) on locality for speed
  - (interesting exception: Cray/Tera MTA, www.cray.com/products/systems/mta/)

Memory Hierarchy: Terminology

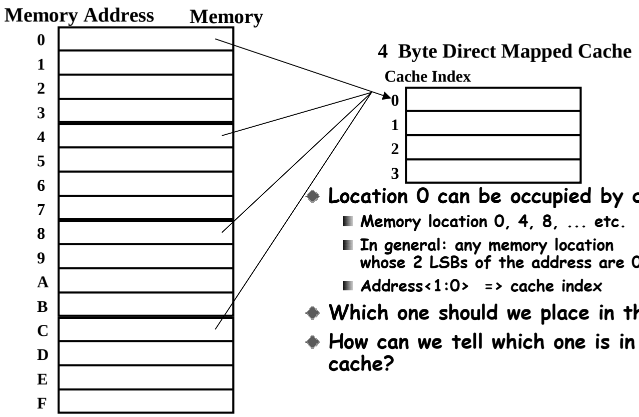
- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264!)



Cache Measures

- *Hit rate*: fraction found in that level
  - So high that usually talk about *Miss rate*
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory
- Average memory-access time
  = Hit time + Miss rate x Miss penalty
  (ns or clocks)
- *Miss penalty*: time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to lower level
    = f(latency to lower level)
  - *transfer time*: time to transfer block
    =f(BW between upper & lower levels)

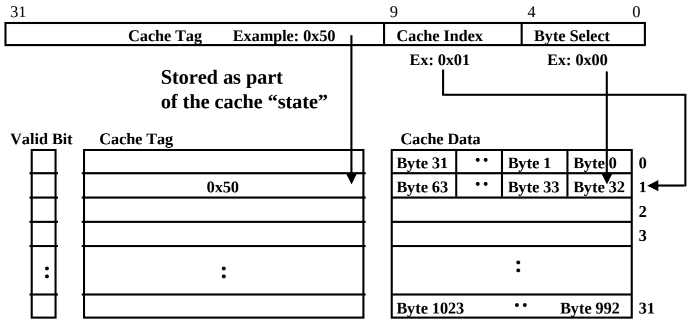
Simplest Cache: Direct Mapped



- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index
- Which one should we place in the cache?
- How can we tell which one is in the cache?

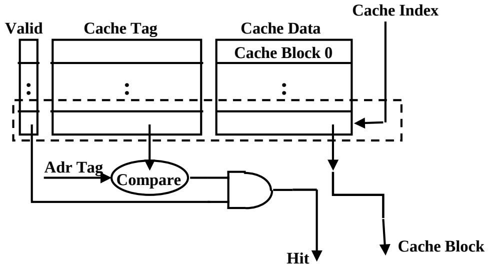
1 KB Direct Mapped Cache, 32B blocks

- For a 2 ** N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



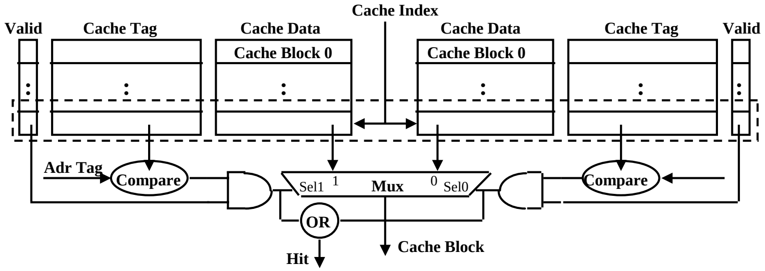
Direct-mapped Cache - structure

- Capacity: C bytes (eg 1KB)
- Blocksize: B bytes (eg 32)
- Byte select bits: 0..log(B)-1 (eg 0..4)
- Number of blocks: C/B (eg 32)
- Address size: A (eg 32 bits)
- Cache index size: I=log(C/B) (eg log(32)=5)
- Tag size: A-I-log(B) (eg 32-5-5=22)



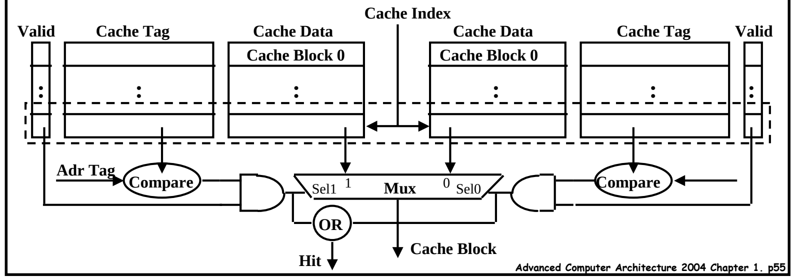
Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operated in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



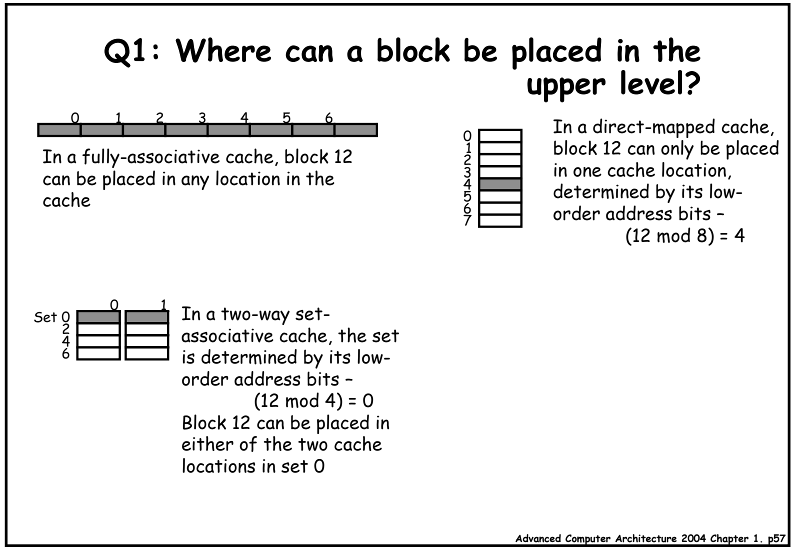
Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
  (Block placement)
- Q2: How is a block found if it is in the upper level?
  (Block identification)
- Q3: Which block should be replaced on a miss?
  (Block replacement)
- Q4: What happens on a write?
  (Write strategy)



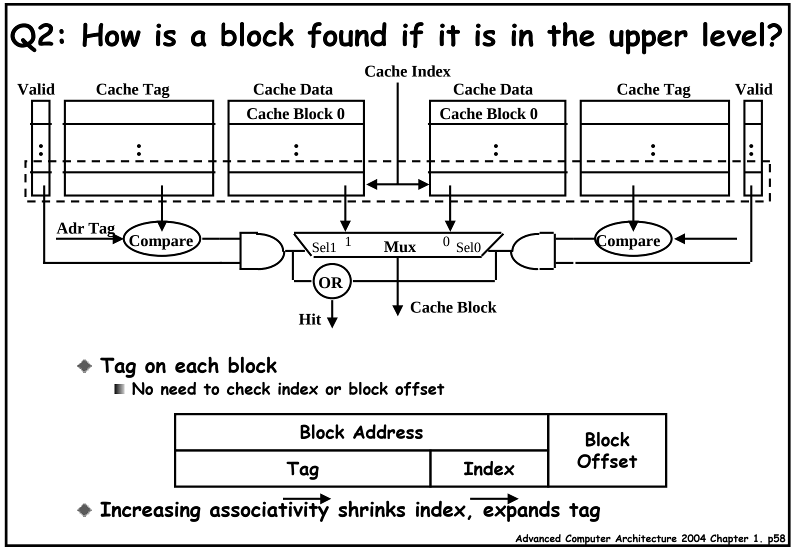
Q1: Where can a block be placed in the upper level?

In a fully-associative cache, block 12 can be placed in any location in the cache

In a direct-mapped cache, block 12 can only be placed in one cache location, determined by its low-order address bits –
(12 mod 8) = 4

In a two-way set-associative cache, the set is determined by its low-order address bits –
(12 mod 4) = 0
Block 12 can be placed in either of the two cache locations in set 0

Advanced Computer Architecture 2004 Chapter 1. p57



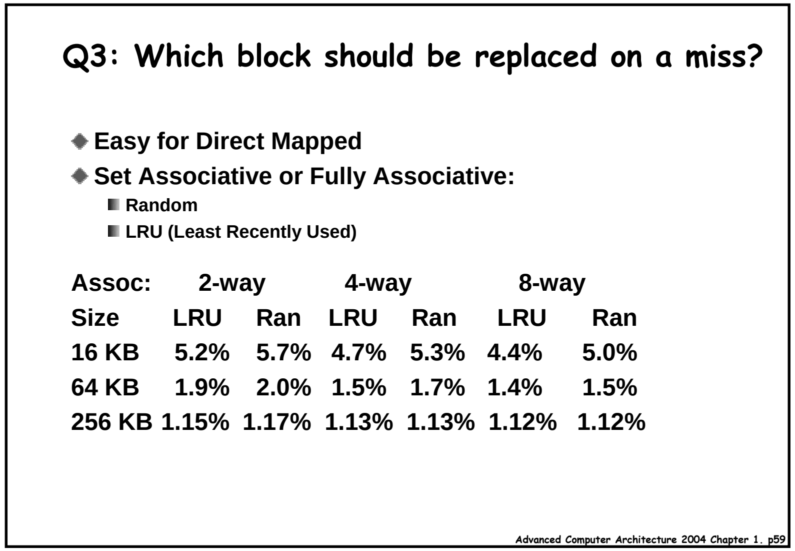
Q2: How is a block found if it is in the upper level?

- Tag on each block
  - No need to check index or block offset

Block Address		Block Offset
Tag	Index	

- Increasing associativity shrinks index, expands tag

Advanced Computer Architecture 2004 Chapter 1. p58

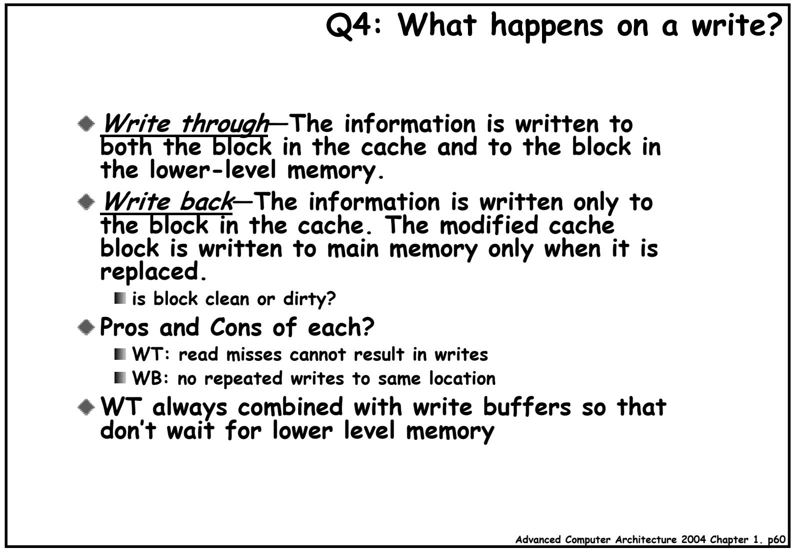


Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Advanced Computer Architecture 2004 Chapter 1. p59

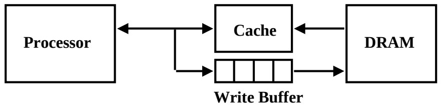


Q4: What happens on a write?

- *Write through*—The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location
- WT always combined with write buffers so that don't wait for lower level memory

Advanced Computer Architecture 2004 Chapter 1. p60

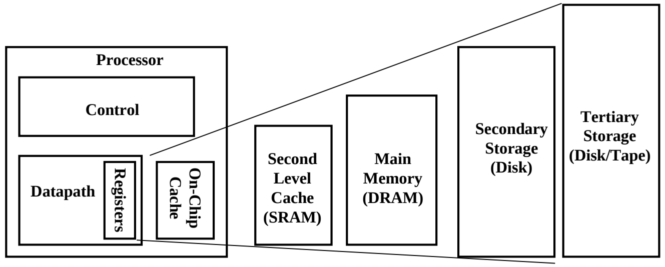
Write Buffer for Write Through



- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

A Modern Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



Speed (ns):	1s	10s	100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks	Ms	Gs	Ts

Summary #1/4: Pipelining & Performance

- Just overlap tasks; easy if tasks are independent
- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- Time is measure of performance: latency or throughput
- CPI Law:

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \text{ x } \frac{\text{Cycles}}{\text{Instruction}} \text{ x } \frac{\text{Seconds}}{\text{Cycle}}$$

Summary #2/4: Caches

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity.
- Write Policy:
  - Write Through: needs a write buffer.
  - Write Back: control can be complex
- Today CPU time is often dominated by memory access time, not just computational work. What does this mean to Compilers, Data structures, Algorithms?