Lecture 2 – Performance & Pipelining

Slides were used during lectures by
David Patterson, Berkeley, spring 2006

Review from last lecture

- Tracking and extrapolating technology part of architect's responsibility
- Expect Bandwidth in disks, DRAM, network, and processors to improve by at least as much as the square of the improvement in Latency
- Quantify Cost (vs. Price)
  - IC ≈ f(Area^2) + Learning curve, volume, commodity, margins
- Quantify dynamic and static power
  - Capacitance x Voltage^2 x frequency, Energy vs. power
- Quantify dependability
  - Reliability (MTTF vs. FIT), Availability (MTTF/(MTTF+MTTR)

Outline

- Quantify and summarize performance
  - Ratios, Geometric Mean, Multiplicative Standard Deviation
  - Fallacies & Pitfalls: Benchmarks age, disks fail, 1 point fail danger
- Pipelining
  - MIPS: an ISA for Pipelining
  - 5 stage pipelining
  - Structural and Data Hazards
  - Forwarding
  - Branch Schemes
  - Exceptions and Interrupts
- Conclusion

Definition: Performance

- Performance is in units of things per sec
  - bigger is better
- If we are primarily concerned with response time

$$Performance(X) = \frac{1}{ExecutionTime(X)}$$

" X is n times faster than Y" means

$$n = \frac{Performance(X)}{Performance(Y)} = \frac{ExecutionTime(Y)}{ExecutionTime(X)}$$

Performance: What to measure?

- Usually rely on benchmarks vs. real workloads
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
- SPECCPU: popular desktop benchmark suite
  - CPU only, split between integer and floating point programs
  - SPECCPU2006:
    - » *Motto: "An ounce of honest data is worth a pound of marketing hype"*
    - » 12 integer and 17 floating point programs
  - SPECSFS (NFS file server) and SPECWeb (WebServer) added as server benchmarks
- Transaction Processing Council measures server performance and cost-performance for databases
  - TPC-C Complex query for Online Transaction Processing
  - TPC-H models ad hoc decision support
  - TPC-W a transactional web benchmark
  - TPC-App application server and web services benchmark

How Summarize Suite Performance (1/5)

- Arithmetic average of execution time of all programs?
  - But they vary by 4X in speed, so some would be more important than others in arithmetic average
- Could add a weight per program, but how pick a weight?
  - Different companies want different weights for their products
- SPECRatio: Normalize execution times to reference computer, yielding a ratio proportional to

$$\text{Performance} = \frac{\text{time on reference computer}}{\text{time on computer rated}}$$

How Summarize Suite Performance (2/5)

- If program SPECRatio on Computer A is 1.25 times bigger than Computer B, then

$$1.25 = \frac{SPECRatio_A}{SPECRatio_B} = \frac{\left(\frac{ExecutionTime_{reference}}{ExecutionTime_A}\right)}{\left(\frac{ExecutionTime_{reference}}{ExecutionTime_B}\right)}$$

$$= \frac{ExecutionTime_B}{ExecutionTime_A} = \frac{Performance_A}{Performance_B}$$

  - Note that when comparing 2 computers as a ratio, execution times on the reference computer drop out, so choice of reference computer is irrelevant

How Summarize Suite Performance (3/5)

- Since ratios, proper mean is geometric mean (SPECRatio unitless, so arithmetic mean meaningless)

$$GeometricMean = \sqrt[n]{\prod_{i=1}^{n} SPECRatio_i}$$

1. Geometric mean of the ratios is the same as the ratio of the geometric means
2. Ratio of geometric means = Geometric mean of performance ratios ⇒ choice of reference computer is irrelevant!

These two points make geometric mean of ratios attractive to summarize performance

How Summarize Suite Performance (4/5)

- Does a single mean well summarize performance of programs in benchmark suite?
- Can decide if mean a good predictor by characterizing variability of distribution using standard deviation
- Like geometric mean, geometric standard deviation is multiplicative rather than arithmetic
- Can simply take the logarithm of SPECRatios, compute the standard mean and standard deviation, and then take the exponent to convert back:

$$GeometricMean = \exp\left(\frac{1}{n} \times \sum_{i=1}^{n} \ln(SPECRatio_i)\right)$$

$$GeometricStDev = \exp(StDev(\ln(SPECRatio_i)))$$

How Summarize Suite Performance (5/5)

- Standard deviation is more informative if know distribution has a standard form
  - *bell-shaped normal distribution*, whose data are symmetric around mean
  - *lognormal distribution*, where logarithms of data – not data itself – are normally distributed (symmetric) on a logarithmic scale
- For a lognormal distribution, we expect that

  68% of samples fall in range

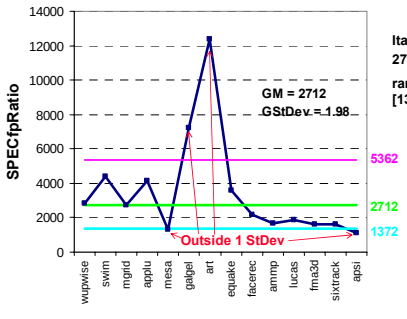
$$[mean / gstdev, mean \times gstdev]$$

  95% of samples fall in range

$$[mean / gstdev^2, mean \times gstdev^2]$$

Example Standard Deviation (1/3)

- GM and multiplicative StDev of SPECfp2000 for Itanium 2

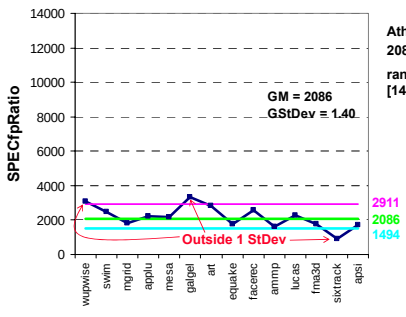


Itanium 2 = 2712/100 x Sun Ultra 5 (GM)

range within 1 StDev is [13.72, 53.62]

Example Standard Deviation (2/3)

- GM and multiplicative StDev of SPECfp2000 for AMD Athlon

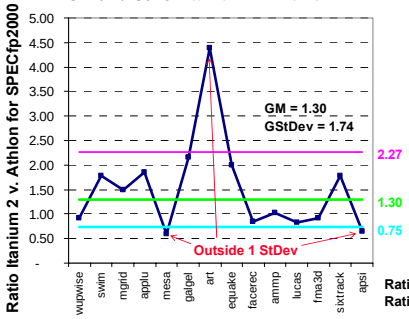


Athlon = 2086/100 x Sun Ultra 5 (GM)

range within 1 StDev is [14.94, 29.11]

Example Standard Deviation (3/3)

- GM and StDev Itanium 2 v Athlon



ExTimeratio	SPECratio
0.92	0.92
1.77	1.77
1.49	1.49
1.85	1.85
0.60	0.60
2.16	2.16
4.40	4.40
2.00	2.00
0.85	0.85
1.03	1.03
0.83	0.83
0.92	0.92
1.79	1.79
0.65	0.65

Ratio execution times (At/It) = Ratio of SPECratios (It/At)

Itanium 2 = 1.30X Athlon (GM), 1 St.Dev. range [0.75,2.27]

Comments on Itanium 2 and Athlon

- Standard deviation for SPECRatio of 1.98 for Itanium 2 is much higher –vs. 1.40– so results will differ more widely from the mean, and therefore are likely less predictable
- SPECRatios falling within one standard deviation:
  - 10 of 14 benchmarks (71%) for Itanium 2
  - 11 of 14 benchmarks (78%) for Athlon
- Thus, results are quite compatible with a lognormal distribution (expect 68% for 1 StDev)
- Itanium 2 vs. Athlon St.Dev is 1.74, which is high, so less confidence in claim that Itanium 1.30 times as fast as Athlon
  - Indeed, Athlon faster on 6 of 14 programs
- Range is [0.75,2.27] with 11/14 inside 1 StDev (78%)

Fallacies and Pitfalls (1/2)

- Fallacies - commonly held misconceptions
  - When discussing a fallacy, we try to give a counterexample.
- Pitfalls - easily made mistakes.
  - Often generalizations of principles true in limited context

Show Fallacies and Pitfalls to help you avoid these errors

- Fallacy: Benchmarks remain valid indefinitely
  - Once a benchmark becomes popular, tremendous pressure to improve performance by targeted optimizations or by aggressive interpretation of the rules for running the benchmark: "benchmarksmanship."
  - 70 benchmarks from the 5 SPEC releases. 70% were dropped from the next release since no longer useful
- Pitfall: A single point of failure
  - Rule of thumb for fault tolerant systems: make sure that every component was redundant so that no single component failure could bring down the whole system (e.g, power supply)

Fallacies and Pitfalls (2/2)

- Fallacy - Rated MTTF of disks is 1,200,000 hours or ≈ 140 years, so disks practically never fail
- But disk lifetime is 5 years ⇒ replace a disk every 5 years; on average, 28 replacements wouldn't fail
- A better unit: % that fail (1.2M MTTF = 833 FIT)
- Fail over lifetime: if had 1000 disks for 5 years
  = 1000*(5*365*24)*833 /10^9 = 36,485,000 / 10^6 = 37
  = 3.7% (37/1000) fail over 5 yr lifetime (1.2M hr MTTF)
- But this is under pristine conditions
  - little vibration, narrow temperature range ⇒ no power failures
- Real world:
  - 3% to 6% of SCSI drives fail per year
    - » 3400 - 6800 FIT or 150,000 - 300,000 hour MTTF [Gray & van Ingen 05]
  - 3% to 7% of ATA drives fail per year
    - » 3400 - 8000 FIT or 125,000 - 300,000 hour MTTF [Gray & van Ingen 05]

Outline

- Quantify and summarize performance
  - Ratios, Geometric Mean, Multiplicative Standard Deviation
  - Fallacies & Pitfalls: Benchmarks age, disks fail, 1 point fail danger
- Pipelining
  - MIPS: an ISA for Pipelining
  - 5 stage pipelining
  - Structural and Data Hazards
  - Forwarding
  - Branch Schemes
  - Exceptions and Interrupts
- Conclusion

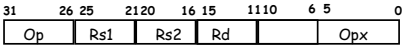
A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

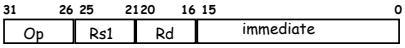
see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Example: MIPS

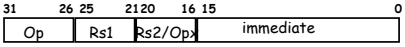
Register-Register



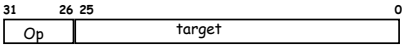
Register-Immediate



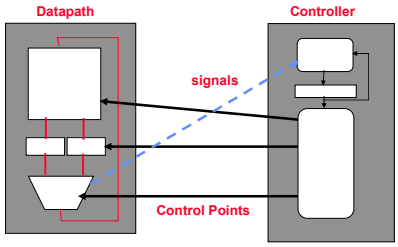
Branch



Jump / Call



Datapath vs Control



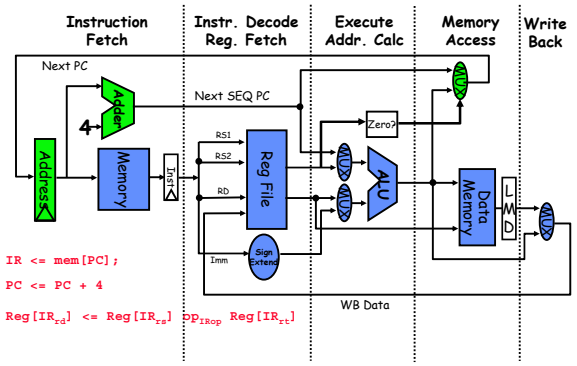
- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
  - Based on desired function and signals

Approaching an ISA

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by the register transfer language (RTL) on *architected registers* and memory
- Given technology constraints assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MBR, ...)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

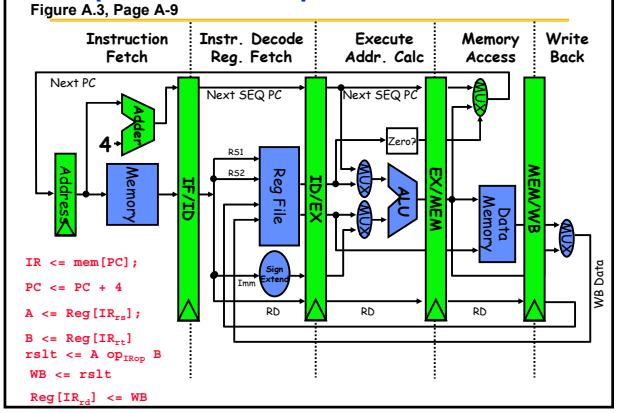
5 Steps of MIPS Datapath

Figure A.2, Page A-8

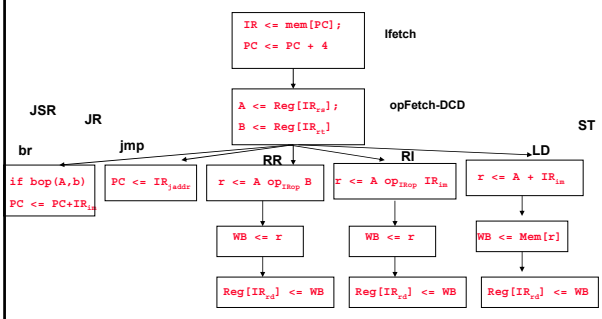


5 Steps of MIPS Datapath

Figure A.3, Page A-9

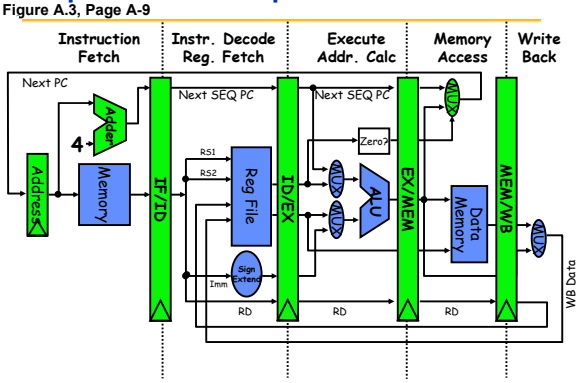


Inst. Set Processor Controller



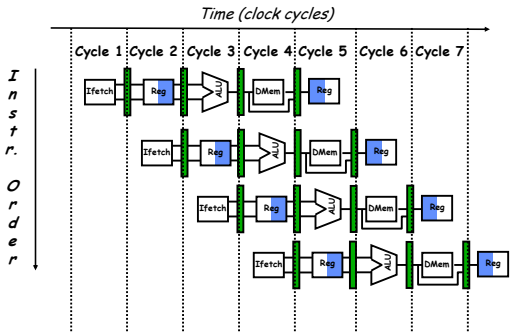
5 Steps of MIPS Datapath

Figure A.3, Page A-9



Visualizing Pipelining

Figure A.3, Page A-9

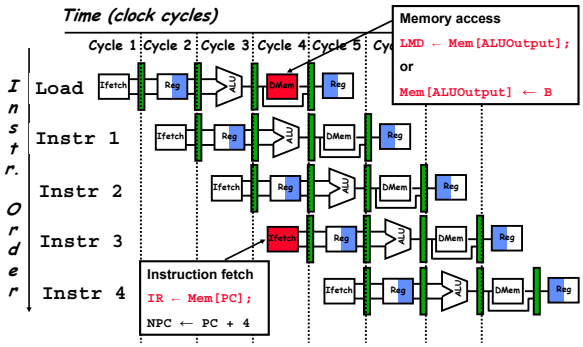


Pipelining is not quite that easy!

- 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).

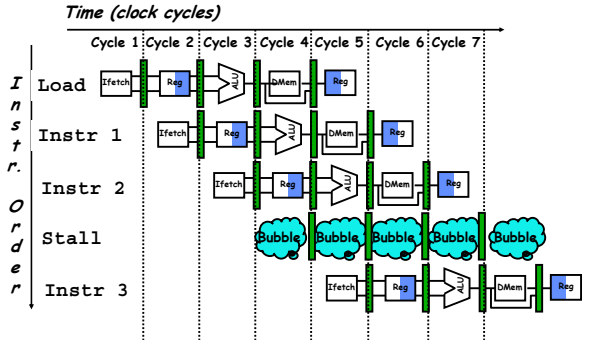
One Memory Port/Structural Hazards

Figure A.4, Page A-14



One Memory Port/Structural Hazards

(Similar to Figure A.5, Page A-15)



How do you "bubble" the pipe?

Speed Up Equation for Pipelining

$$\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}$$

$$CPI_{pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

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

For simple RISC pipeline, Ideal CPI = 1:

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

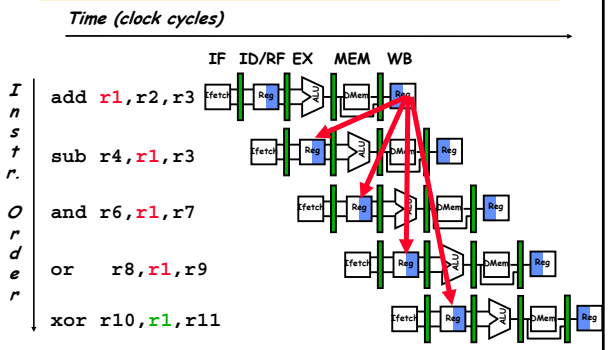
Example: Dual-port vs. Single-port

- 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 ($clock_{unpipe}$/$clock_{pipe}$)
  = Pipeline Depth
  $SpeedUp_B$ = Pipeline Depth/(1 + 0.4 x 1) x ($clock_{unpipe}$/($clock_{unpipe}$ / 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

Data Hazard on R1

Figure A.6, Page A-17



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

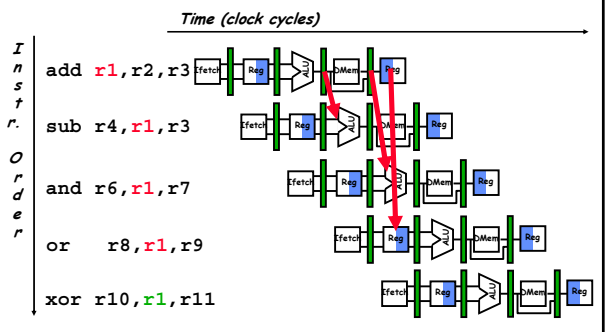
- Write After Write (WAW)
  $Instr_J$ writes operand *before* $Instr_I$ writes it.

```
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 more complicated pipes

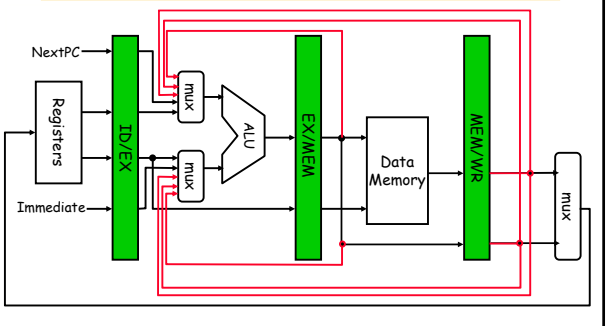
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19



HW Change for Forwarding

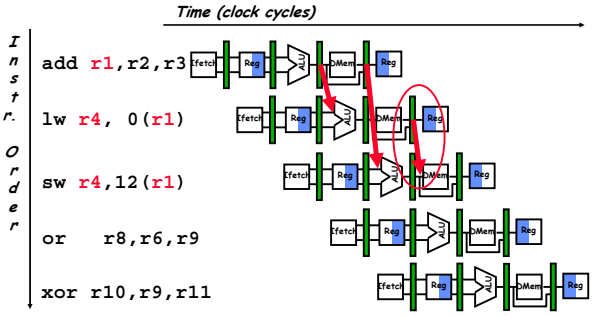
Figure A.23, Page A-37



What circuit detects and resolves this hazard?

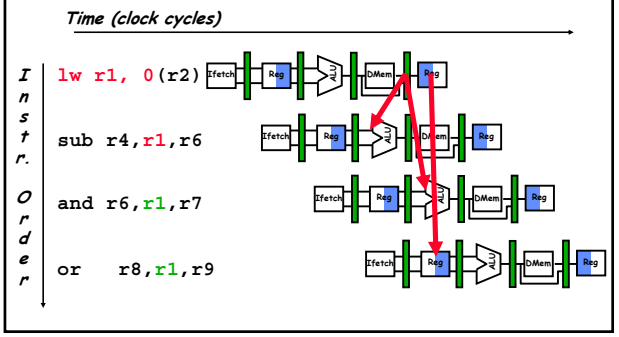
Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20



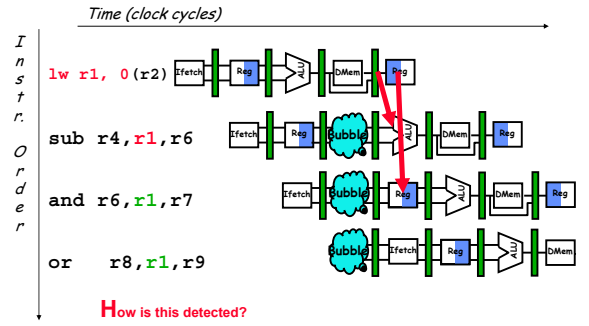
Data Hazard Even with Forwarding

Figure A.9, Page A-20



Data Hazard Even with Forwarding

(Similar to Figure A.10, Page A-21)



How is this detected?

Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

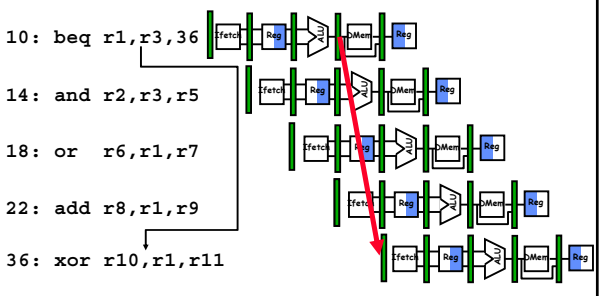
d = e – f;

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

Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

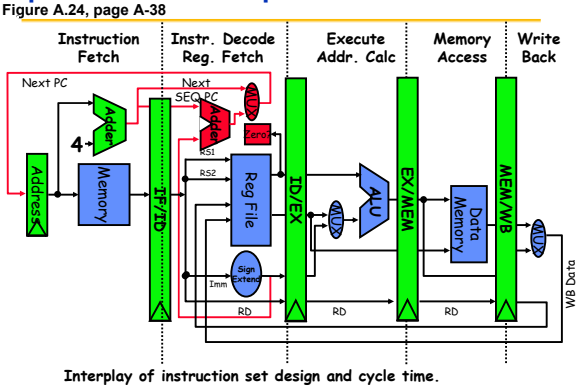
Where is the "commit"?

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- 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

Figure A.24, page A-38



Interplay of instruction set design and cycle time.

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

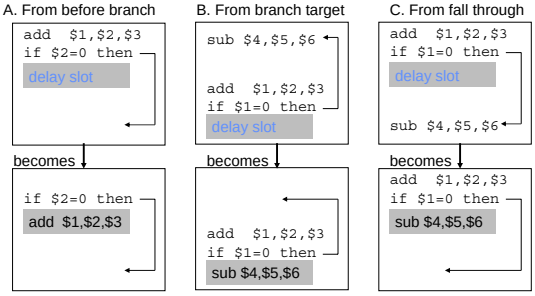
- Define branch to take place AFTER a following instruction

```
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

Scheduling Branch Delay Slots (Fig A.14)



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Delayed Branch

- 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: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

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

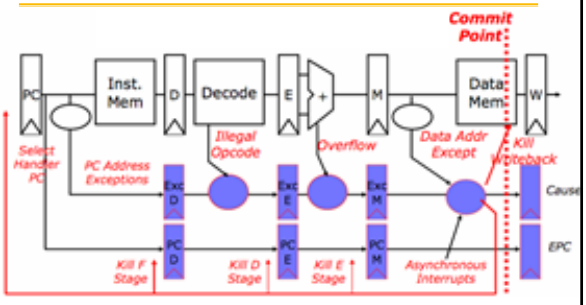
Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

Problems with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- **Interrupt**: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)
- Problem: It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
  - The effect of all instructions up to and including I_i is totalling complete
  - No effect of any instruction after I_i can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.

And In Conclusion:

- Quantify and summarize performance
  - Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail,1 point fail danger
- Control via State Machines and Microprogramming
- 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
- Exceptions, Interrupts add complexity

Reading

- This lecture: appendix A *Pipelining*
- Next lecture: appendix C *Memory Hierarchy*