EEL 5764: Graduate Computer Architecture

Appendix A - Pipelining Review

Ann Gordon-Ross
Electrical and Computer Engineering
University of Florida

http://www.ann.ece.ufl.edu/

These slides are provided by:
David Patterson
Electrical Engineering and Computer Sciences, University of California, Berkeley
Modifications/additions have been made from the originals

What is Pipelining?

- Overlapping execution to produce faster results
  - Washing and drying dishes
  - Washing and drying laundry
  - Automobile assembly line
  - Chipotle, Quiznos, etc
- Pipelining in computer architecture
  - Multiple instructions are overlapped in execution
  - Exploits parallelism
  - Not visible to programmer
- Each stage is a pipeline “cycle”
  - Each stage happens simultaneously so results are produced only as fast as the *longest* pipeline cycle
  - Determines clock cycle time

Outline

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts

A "Typical" RISC ISA (Load/Store)

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

Example: MIPS (- MIPS)

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

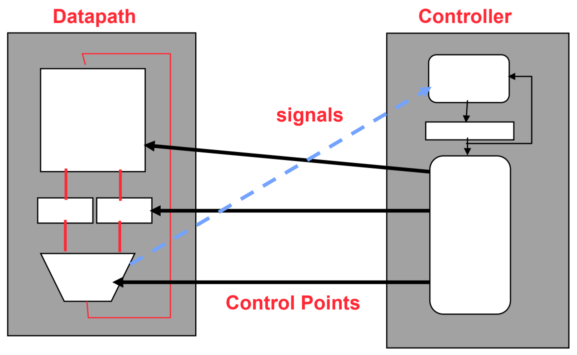
Branch

31 26	25 21	20 16	15 0
Op	Rs1	Rs2/Opx	immediate

Jump / Call

31 26	25 0
Op	target

Datapath vs Control (FSM+D)



- 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

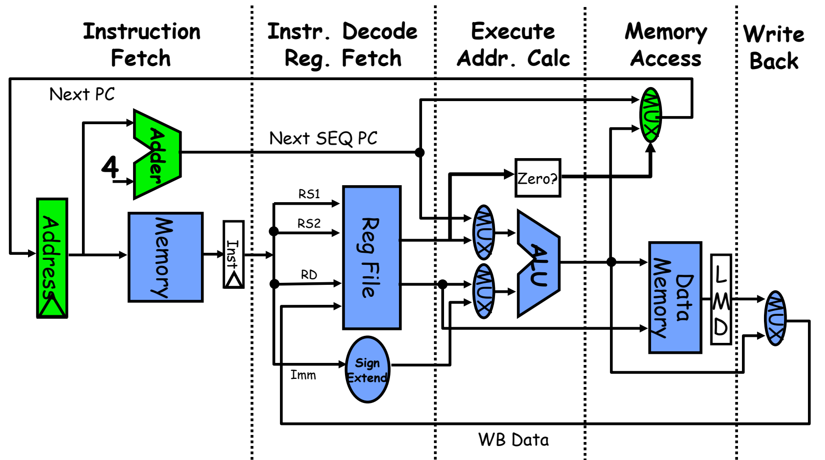
- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on *architected registers* and memory
- Given technology constraints assemble adequate datapath
  - 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
- Implement controller (Finite State Machine (FSM))

Outline

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts

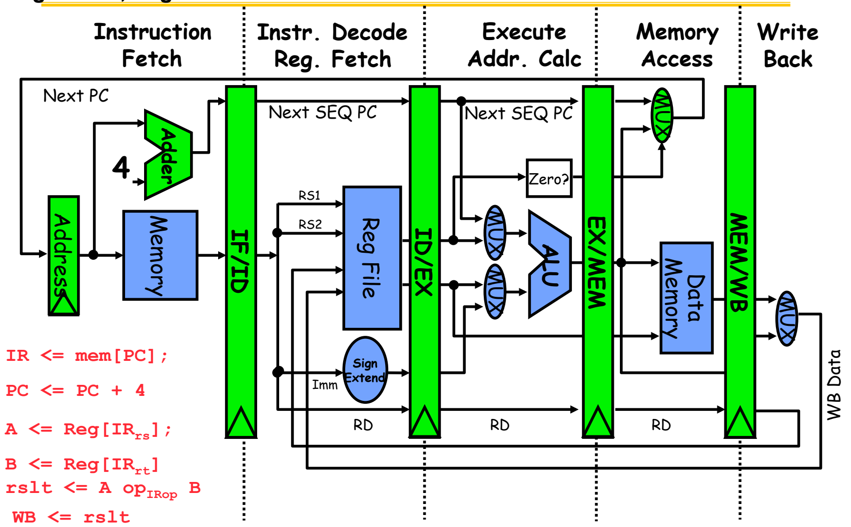
5 Steps of MIPS Datapath

Figure A.2, Page A-8



5 Steps of MIPS Datapath

Figure A.3, Page A-9



$\text{IR} \Leftarrow \text{mem[PC]};$

$\text{PC} \Leftarrow \text{PC} + 4$

$\text{A} \Leftarrow \text{Reg}[\text{IR}_{rs}];$

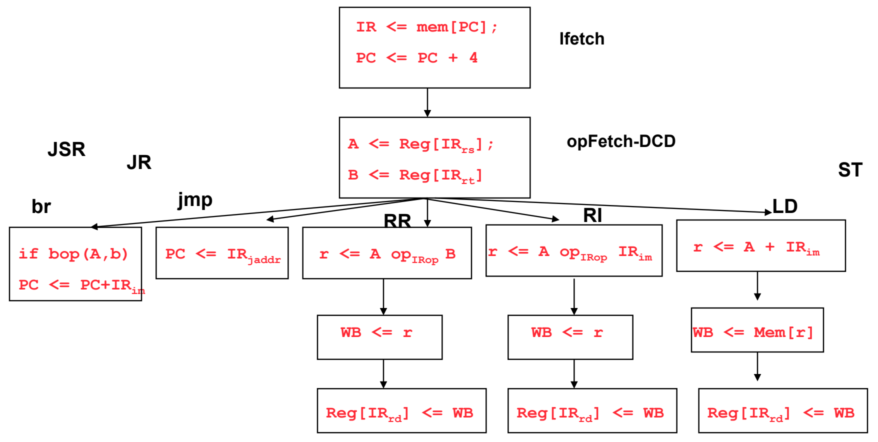
$\text{B} \Leftarrow \text{Reg}[\text{IR}_{rt}]$

$\text{rslt} \Leftarrow \text{A op}_{\text{IRop}} \text{ B}$

$\text{WB} \Leftarrow \text{rslt}$

$\text{Reg}[\text{IR}_{rd}] \Leftarrow \text{WB}$

Inst. Set Processor Controller



$\text{IR} \Leftarrow \text{mem[PC]};$
$\text{PC} \Leftarrow \text{PC} + 4$

$\text{A} \Leftarrow \text{Reg}[\text{IR}_{rs}];$
$\text{B} \Leftarrow \text{Reg}[\text{IR}_{rt}]$

if bop(A,b)
$\text{PC} \Leftarrow \text{PC+IR}_{im}$

$\text{PC} \Leftarrow \text{IR}_{jaddr}$

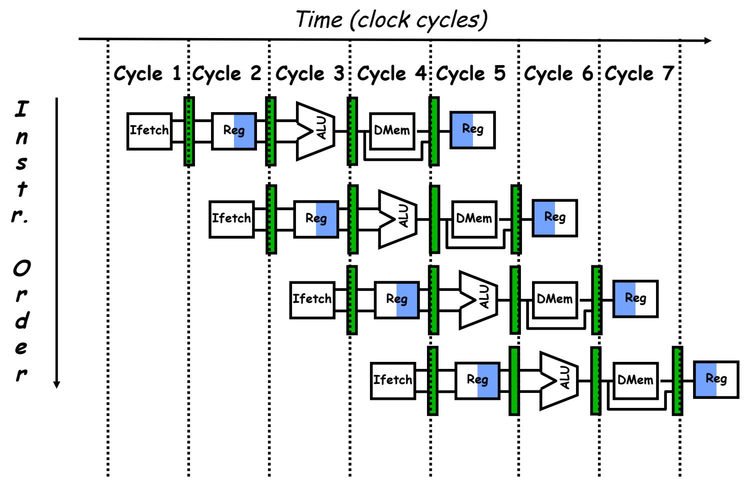
$\text{r} \Leftarrow \text{A op}_{\text{IRop}} \text{ B}$
$\text{WB} \Leftarrow \text{r}$
$\text{Reg}[\text{IR}_{rd}] \Leftarrow \text{WB}$

$\text{r} \Leftarrow \text{A op}_{\text{IRop}} \text{ IR}_{im}$
$\text{WB} \Leftarrow \text{r}$
$\text{Reg}[\text{IR}_{rd}] \Leftarrow \text{WB}$

$\text{r} \Leftarrow \text{A} + \text{IR}_{im}$
$\text{WB} \Leftarrow \text{Mem[r]}$
$\text{Reg}[\text{IR}_{rd}] \Leftarrow \text{WB}$

Visualizing Pipelining

Figure A.2, Page A-8

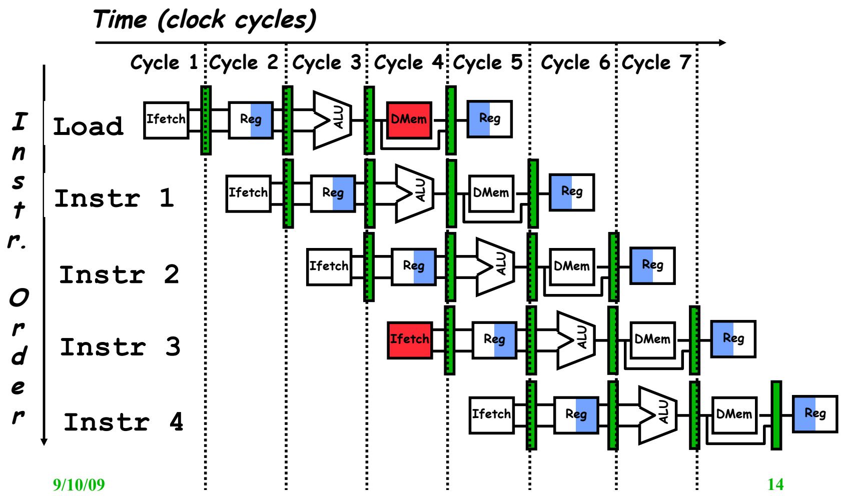


Pipelining is not quite that easy!

- 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).

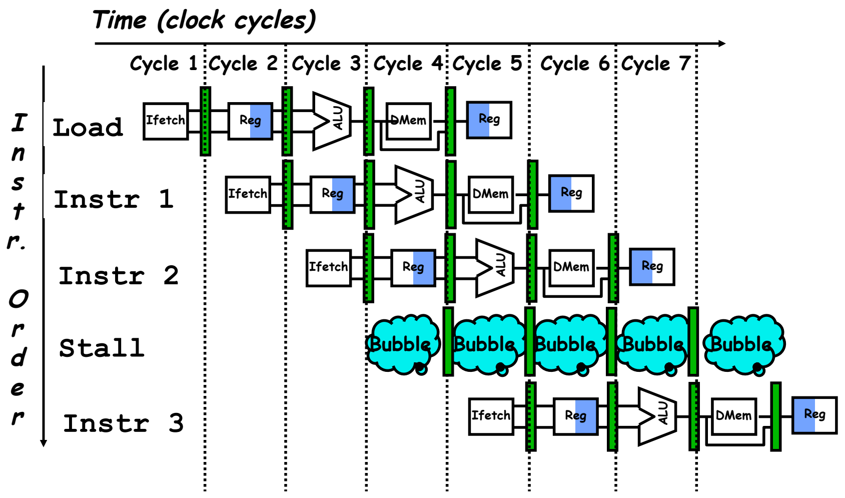
One Memory Port/Structural Hazards

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?

Performance of Pipelines with Stalls

- Ideal CPI speedup is simply the pipeline depth
  - Assumes no stalls, perfect execution
- But, pipelining causes stalls and changes the clock cycle time

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined x Clock cycle unpipelined}}{\text{CPI pipelined x Clock cycle time pipelined}}$$

- Ideal CPI is 1

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI } + \text{Pipeline stall clock cycles per instruction}\\ &= 1 + \text{Pipeline stall clock cycles per instruction}\\ \text{CPI unpipelined} &= \text{Ideal CPI x pipeline depth}\\ &= \text{Pipeline depth}\end{aligned}$$

Performance of Pipelines with Stalls

- **Lets ignore cycle time overhead for pipelining and assume all stages are balanced, thus cycle times for each are equal**

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}$$

$$= \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

- **Assuming no pipeline stalls, speedup is equal to pipeline depth.**
- **But, pipelining changes the clock cycle time too….**

Performance of Pipelines with Stalls

- **Pipelining reduces clock cycle time (increases frequency) – less work to do in each stage**
- **CPI unpipelined is 1**

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined x Clock cycle unpipelined}}{\text{CPI pipelined x Clock cycle time pipelined}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \text{ x } \frac{\text{Clock cycle time unpipelined}}{\text{Clock cycle time pipelined}}$$

- **If all pipeline stages are balanced:**

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

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

Performance of Pipelines with Stalls

$$\text{Speedup from pipelining} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \text{ x } \frac{\text{Clock cycle time unpipelined}}{\text{Clock cycle time pipelined}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \text{ x Pipeline Depth}$$

- **And again, if no stalls, ideal speedup is equal to the pipeline depth**

Example: Dual-port vs. Single-port

- **Machine A: Dual ported memory (“Harvard Architecture”)**
- **Machine B: Single ported memory and the clock rate is 1.05 times faster**
- **Ideal CPI = 1 for both**
- **Loads are 40% of instructions executed**

$$\text{Average instruction time}_A = \text{CPI x Clock cycle time} = \text{Clock cycle time}$$

$$\text{Average instruction time}_B = \text{CPI x Clock cycle time} = (1 + 0.4 \text{ x } 1) \text{ x } \frac{\text{Clock cycle time}}{1.05}$$

$$= 1.3 \text{ x Clock cycle time}$$

- **Machine A is 1.3 times faster**

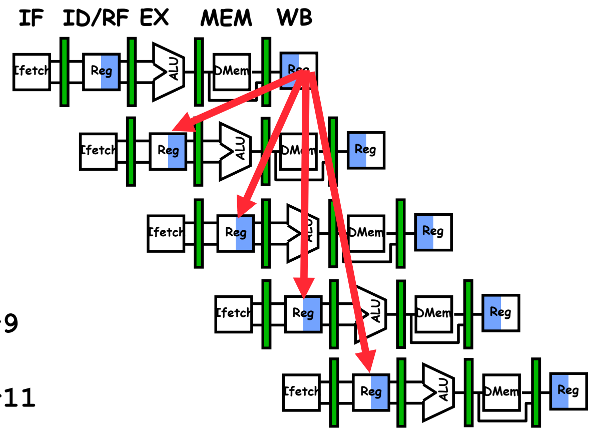
Data Hazard on R1

Figure A.6, Page A-17

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
```



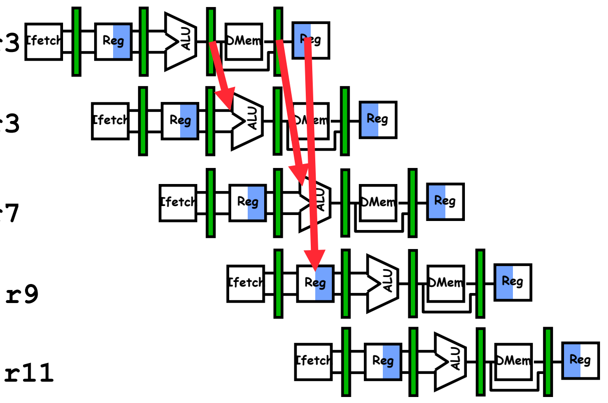
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19

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

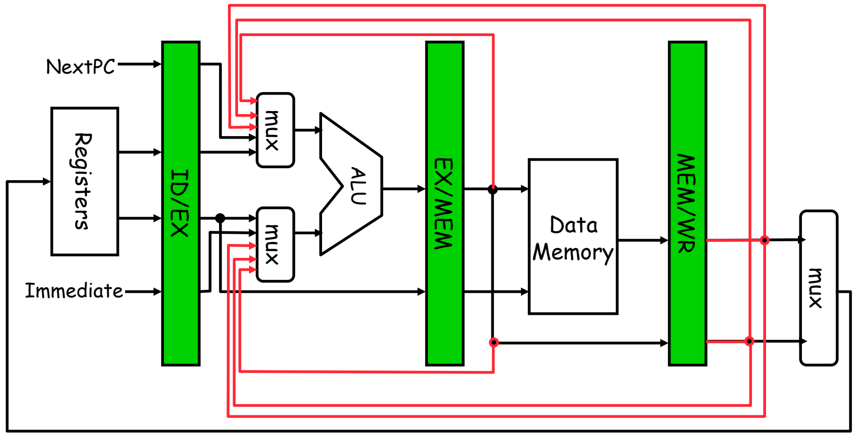
- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.

```
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 more complicated pipes

HW Change for Forwarding

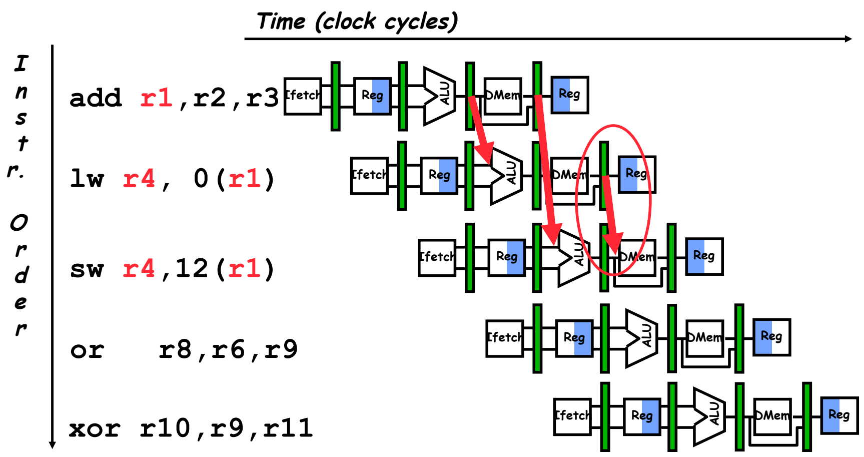
Figure A.23, Page A-37



What circuit detects and resolves this hazard?

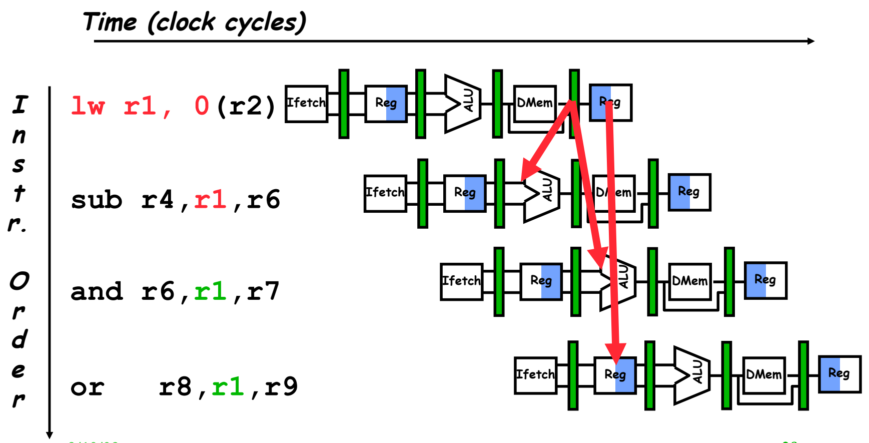
Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20

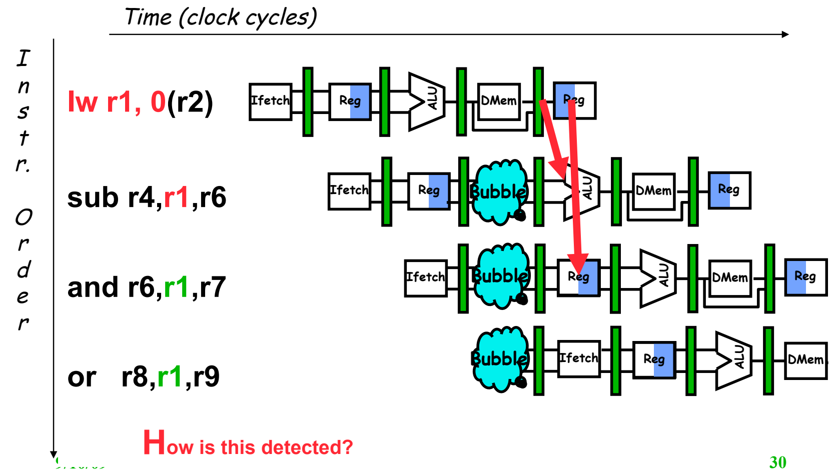


Data Hazard Even with Forwarding

Figure A.9, Page A-21



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.

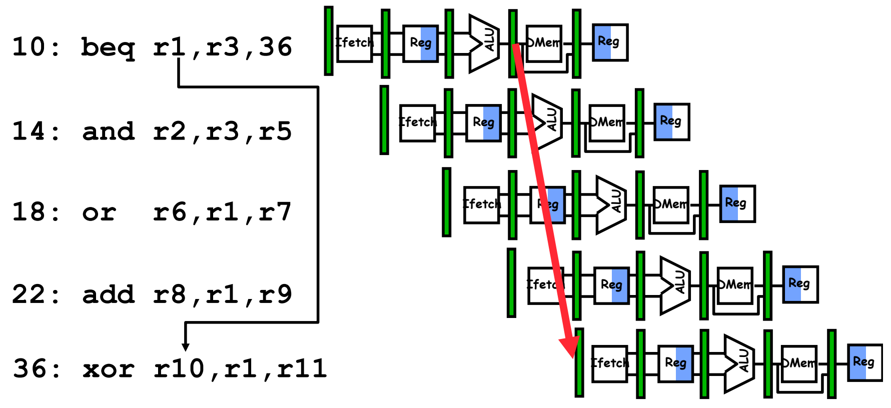
	Slow code:		Fast code:	
	LW	Rb,b	LW	Rb,b
	LW	Rc,c	LW	Rc,c
Stall	ADD	Ra,Rb,Rc	LW	Re,e
	SW	a,Ra	ADD	Ra,Rb,Rc
	LW	Re,e	LW	Rf,f
	LW	Rf,f	SW	a,Ra
Stall	SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
	SW	d,Rd	SW	d,Rd

Compiler optimizes for performance. Hardware checks for safety.

Outline

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion

Control Hazard on Branches Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

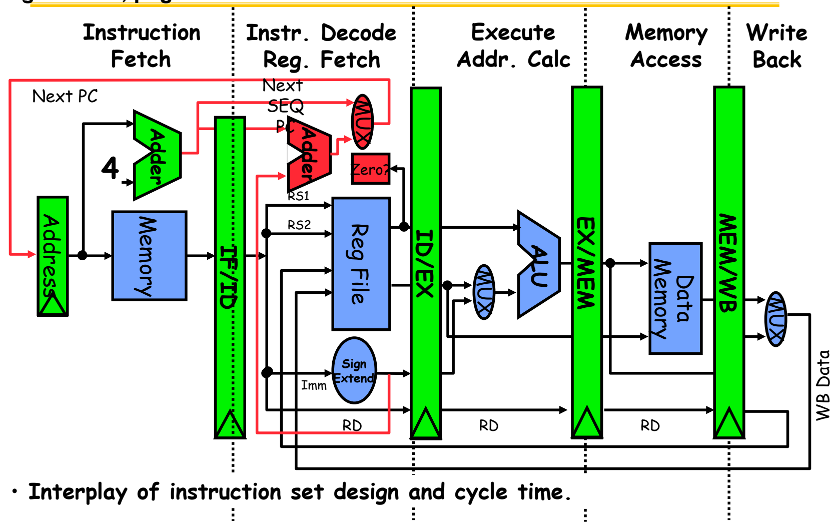
Where is the “commit”?

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

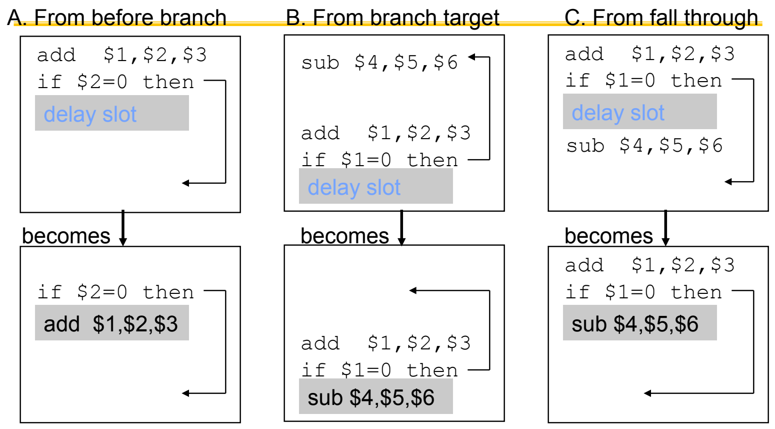
Four Branch Hazard Alternatives

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

```
branch instruction
  sequential successor1
  sequential successor2     Branch delay of length n
  ........
  sequential successorn
branch target if taken
```

- 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