Pipelining

Anne Bracy CS 3410

Computer Science
Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

See P&H Chapter: 4.5-4.8

Single Cycle → Multi-Cycle → Pipelining

Single-cycle

insn0.fetch, dec, exec

insn1.fetch, dec, exec

Multi-cycle

insn0.fetch insn0.dec insn0.exec

insn1.fetch insn1.dec insn1.exec

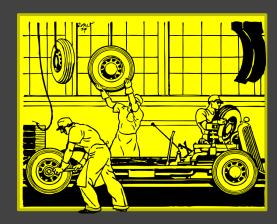
Pipelined

insn0.fetchinsn0.decinsn0.execinsn1.fetchinsn1.decinsn1.exec

Agenda

5-stage Pipeline

- Implementation
- Working Example





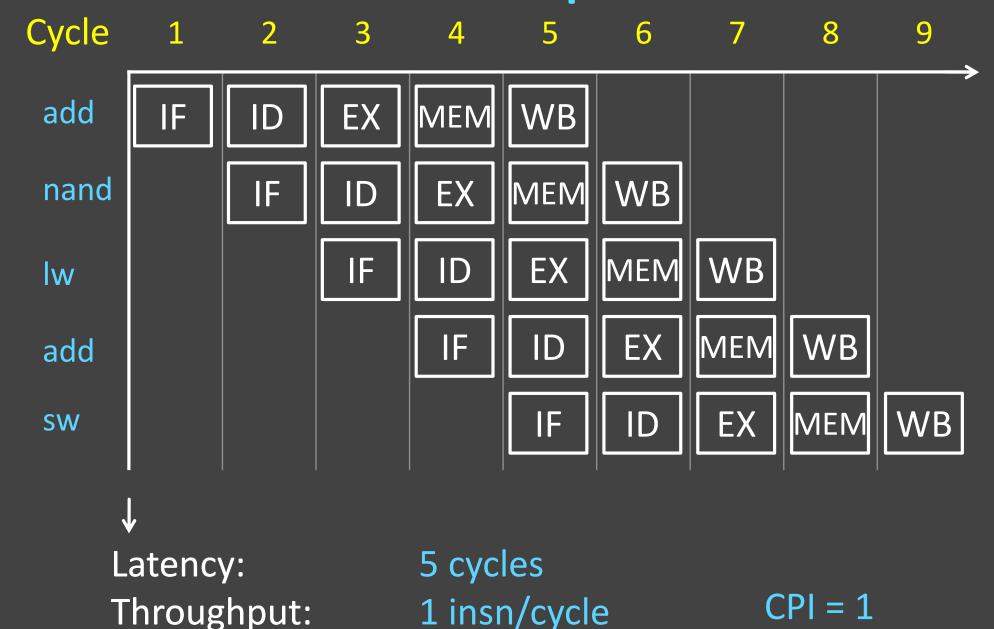


Hazards

- Structural
- Data Hazards
- Control Hazards

Pipelined Processor 4 memory register alu file മ addr inst PC d_{out} $d_{\text{in}} \\$ Σ В control memory compute jump/branch new imm extend targets рс Write-Instruction Instruction ctrl ctrl Memory Execute Decode Back Fetch IF/ID ID/EX EX/MEM MEM/WB

Time Graphs



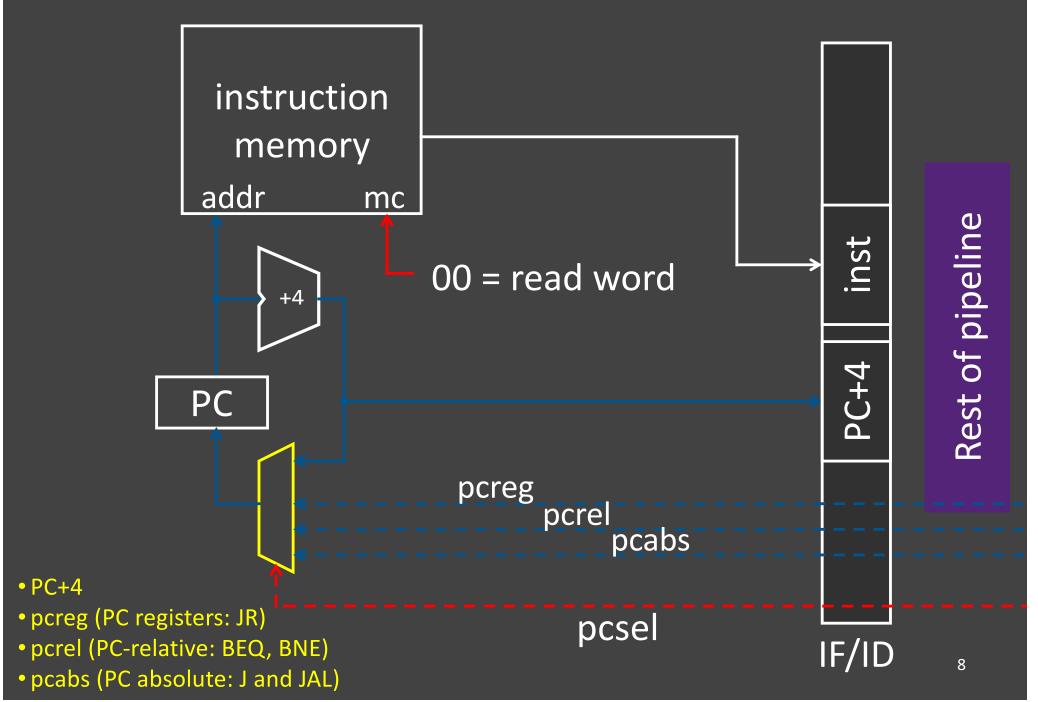
Principles of Pipelined Implementation

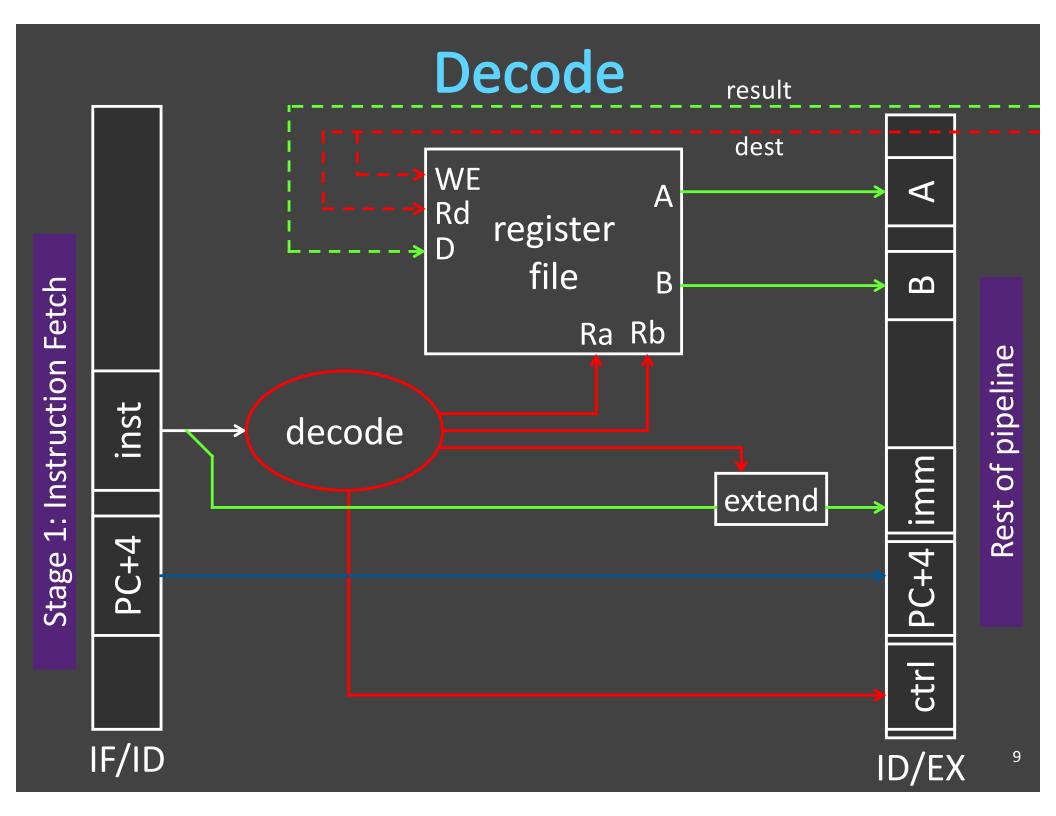
- Break datapath into multiple cycles (here 5)
 - Parallel execution increases throughput
 - Balanced pipeline very important
 - Slowest stage determines clock rate
 - Imbalance kills performance
- Add pipeline registers (flip-flops) for isolation
 - Each stage begins by reading values from latch
 - Each stage ends by writing values to latch
- Resolve hazards

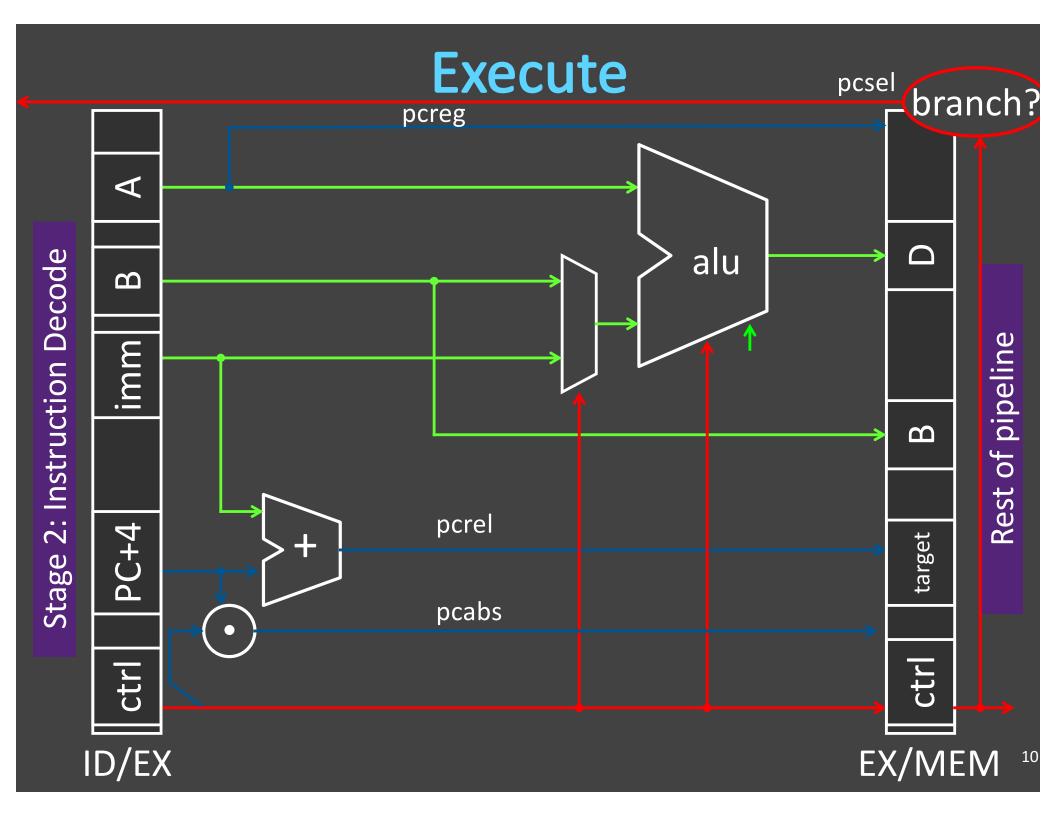
Pipeline Stages

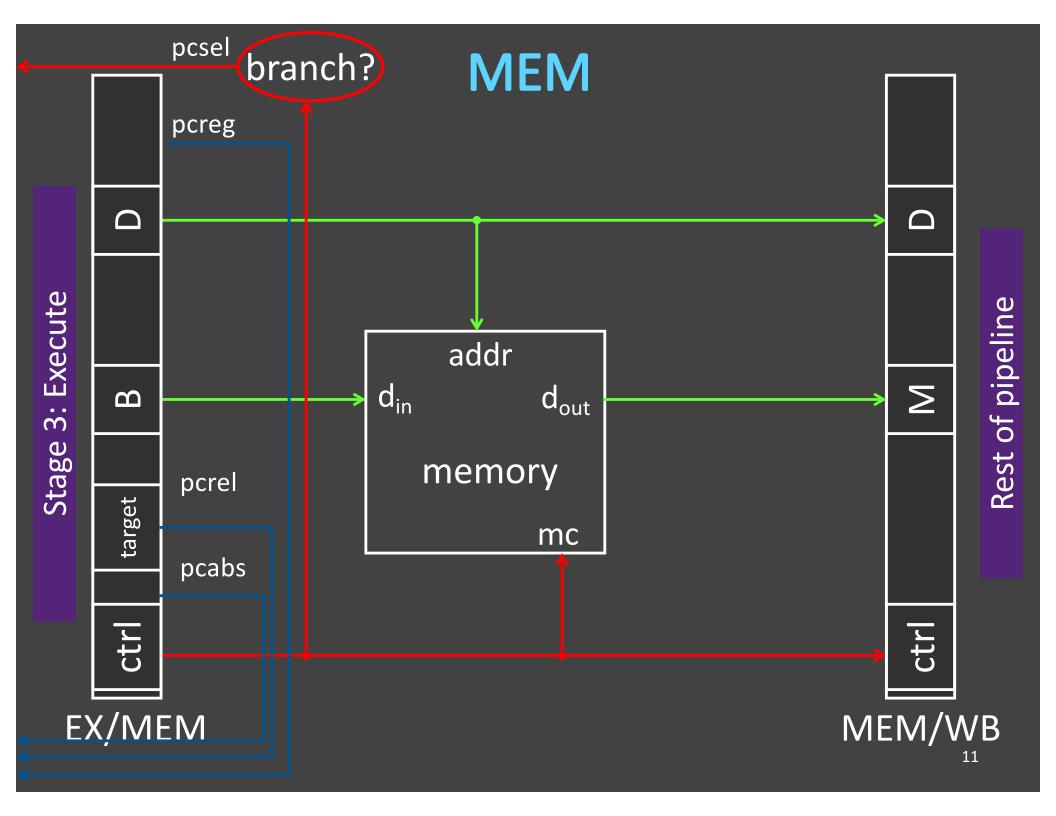
Stage	Perform Functionality	Latch values of interest
Fetch	Use PC to index Program Memory, increment PC	Instruction bits (to be decoded) PC + 4 (to compute branch targets)
Decode	Decode instruction, generate control signals, read register file	Control information, Rd index, immediates, offsets, register values (Ra, Rb), PC+4 (to compute branch targets)
Execute	Perform ALU operation Compute targets (PC+4+offset, etc.) in case this is a branch, decide if branch taken	Control information, Rd index, etc. Result of ALU operation, value in case this is a store instruction
Memory	Perform load/store if needed, address is ALU result	Control information, Rd index, etc. Result of load, pass result from execute
Writeback	Select value, write to register file	

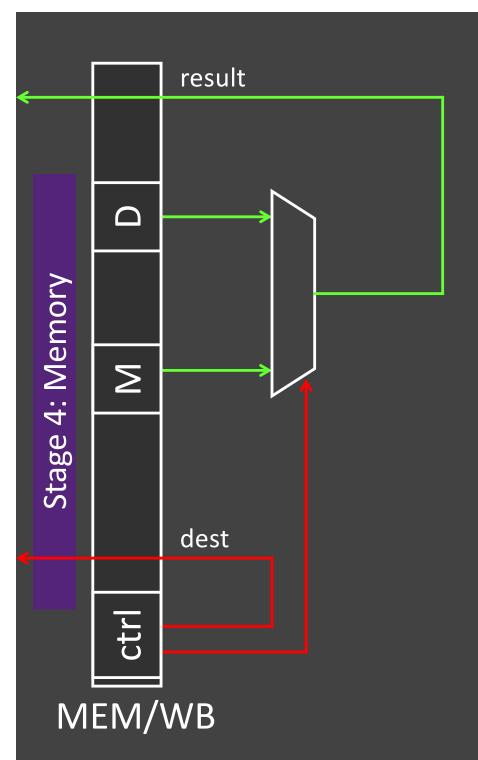
Instruction Fetch

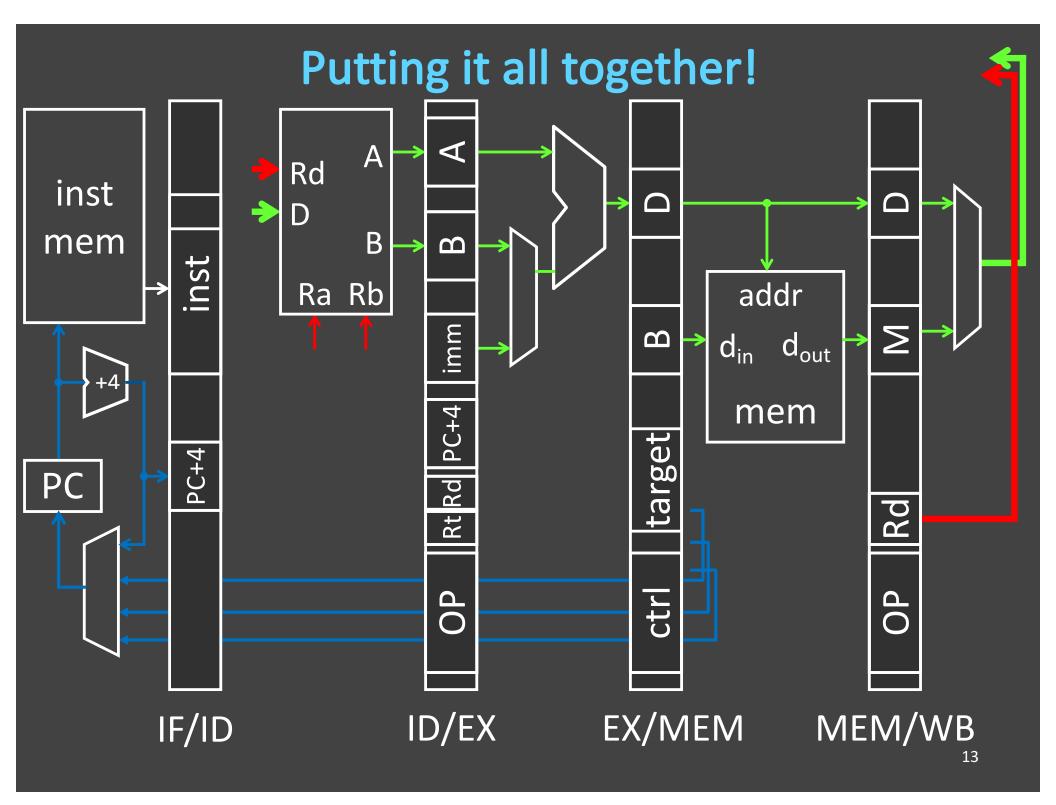












MIPS designed for pipelining

- Instructions same length
 - 32 bits, easy to fetch and then decode

- 3 types of instruction formats
 - Easy to route bits between stages
 - Can read a register source before even knowing what the instruction is
- Memory access through lw and sw only
 - Access memory after ALU

iClicker Question

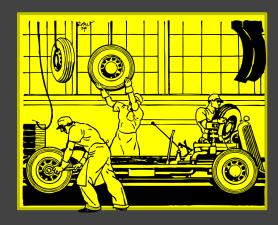
Consider a non-pipelined processor with clock period C (e.g., 50 ns). If you divide the processor into N stages (e.g., 6), your new clock period will be:

- A. C
- B. N
- C. less than C/N
- D. C/N
- E. greater than C/N

Agenda

5-stage Pipeline

- Implementation
- Working Example







Hazards

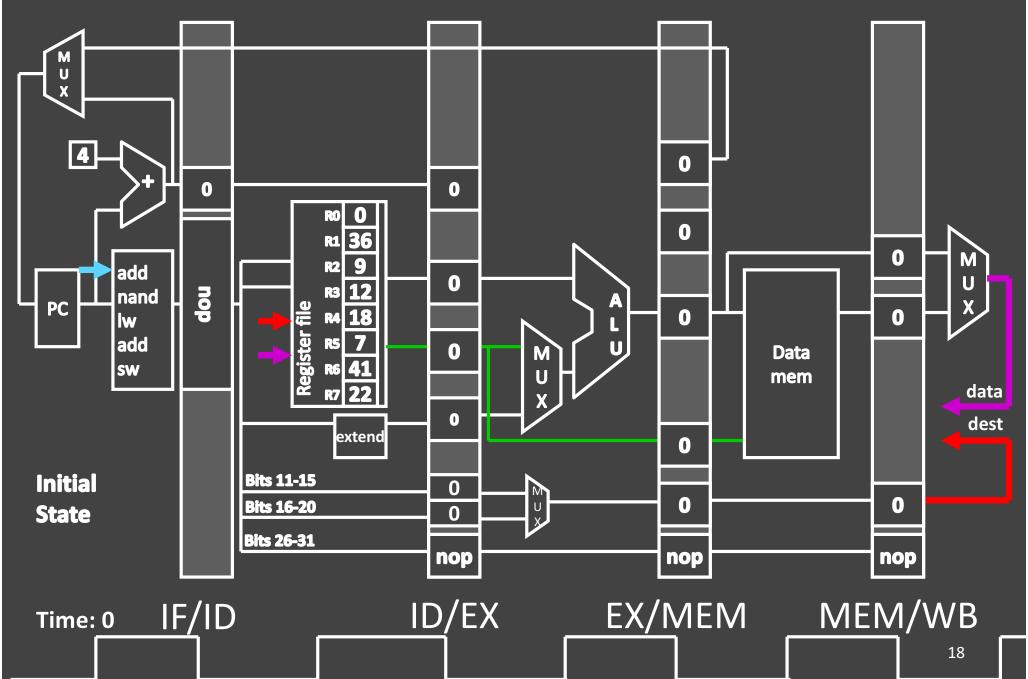
- Structural
- Data Hazards
- Control Hazards

Example: : Sample Code (Simple)

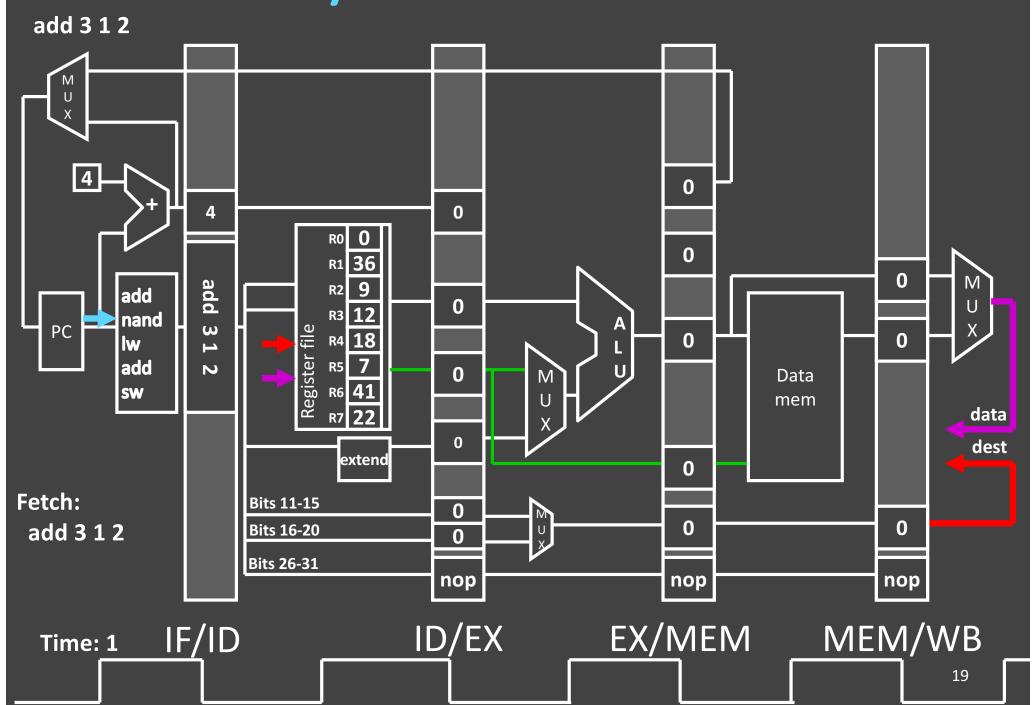
```
add r3 \leftarrow r1, r2
nand r6 \leftarrow r4, r5
lw r4 \leftarrow 20(r2)
add r5 \leftarrow r2, r5
sw r7 \rightarrow 12(r3)
```

Assume 8-register machine

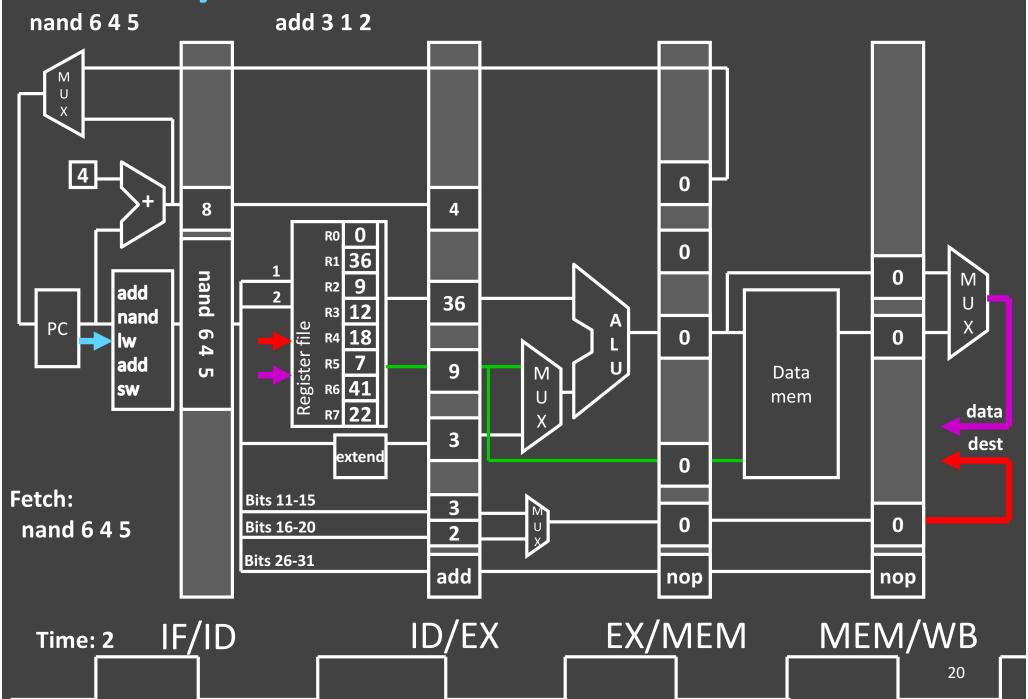
Example: Start State @ Cycle 0



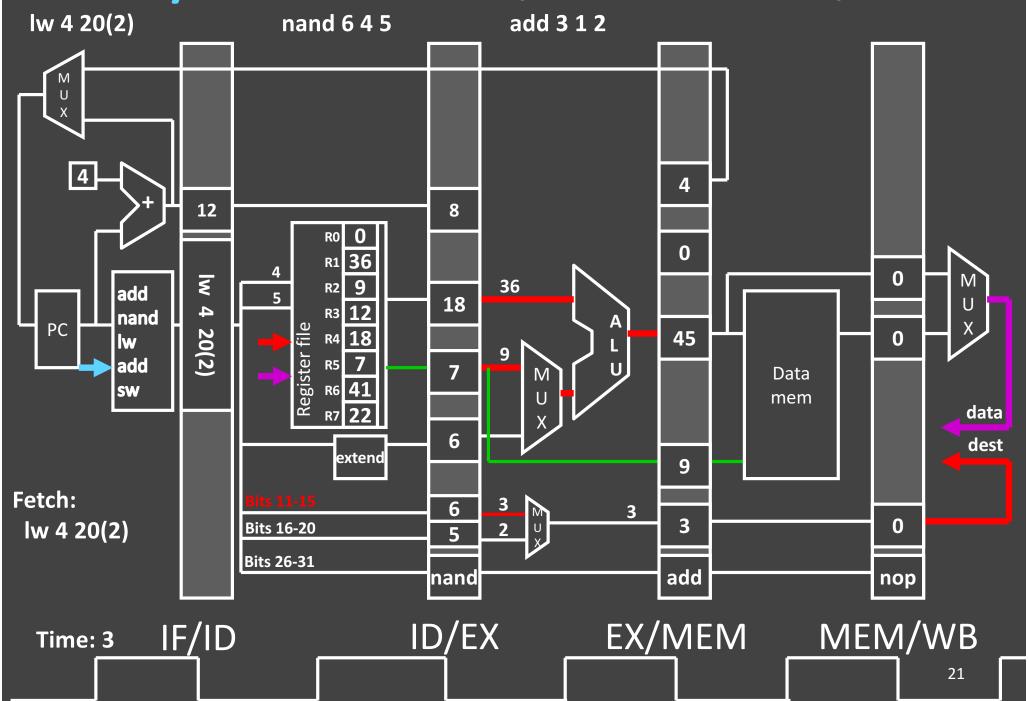
Cycle 1: Fetch add



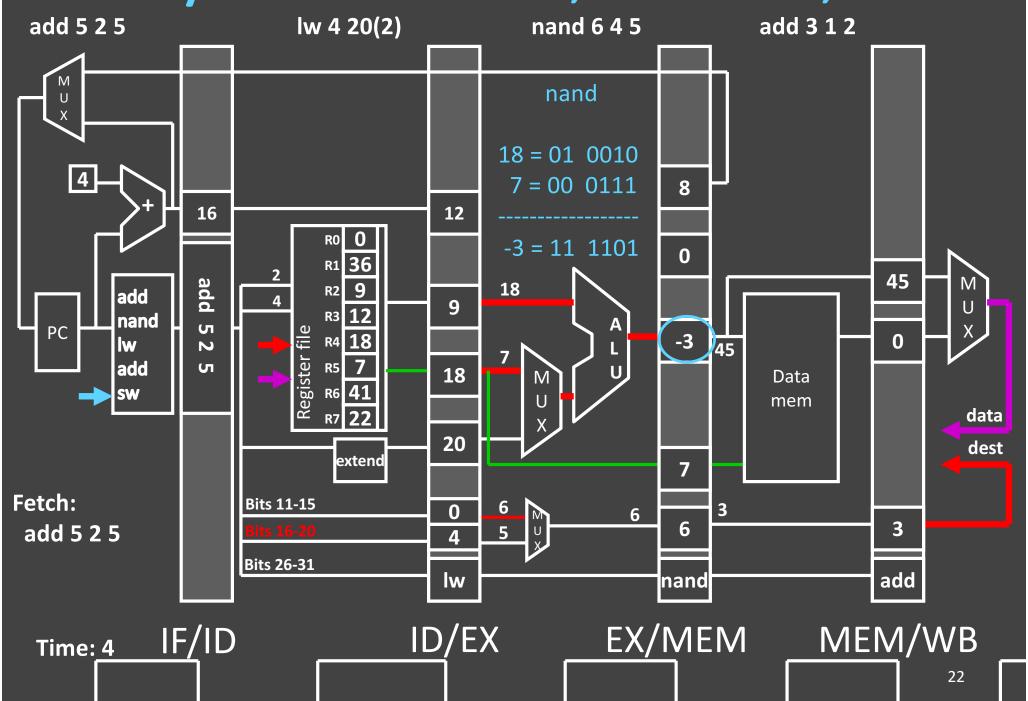
Cycle 2: Fetch nand, Decode add



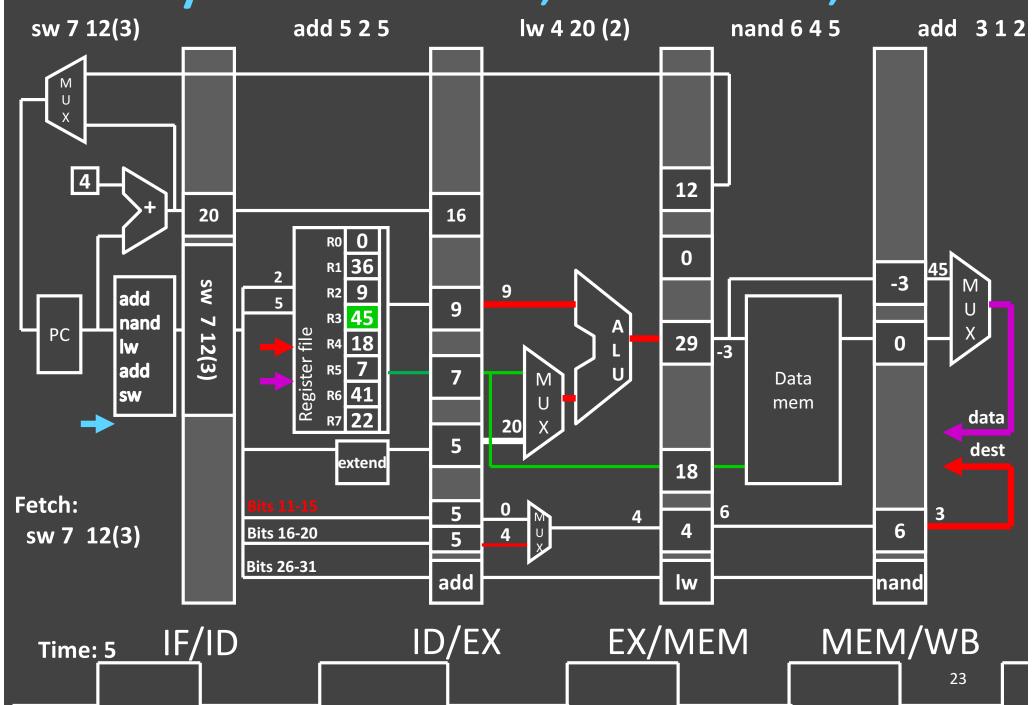
Cycle 3: Fetch lw, Decode nand, ...

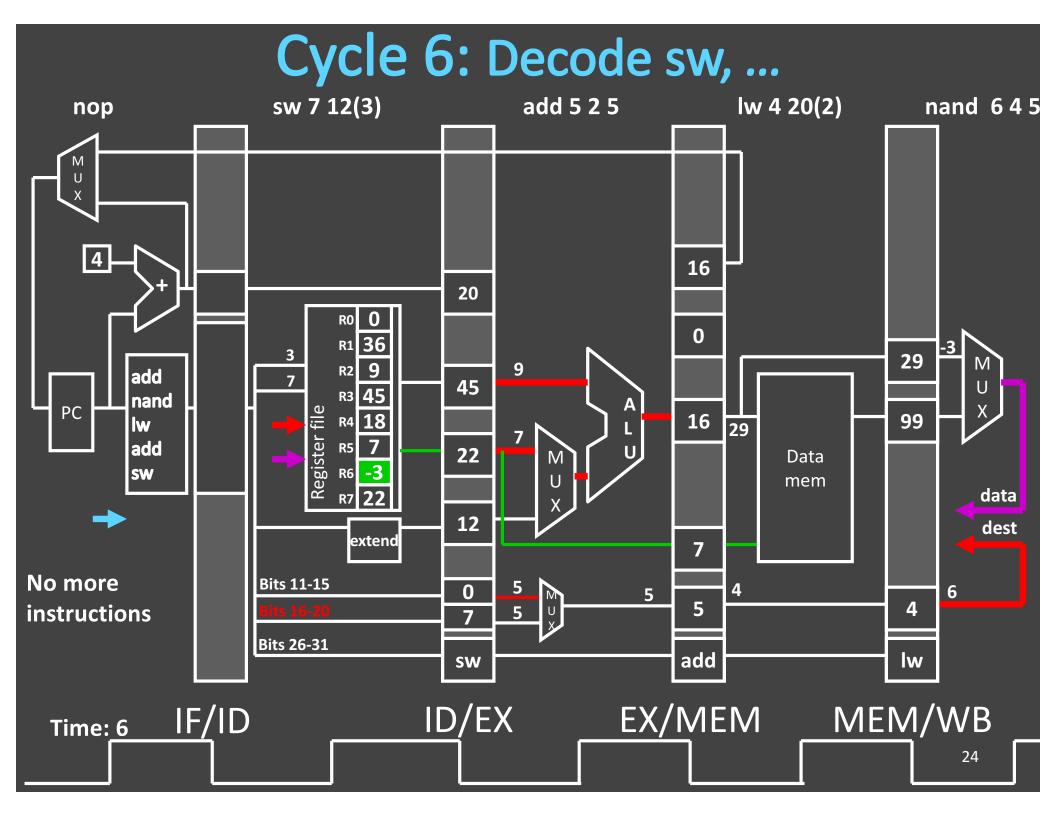


Cycle 4: Fetch add, Decode lw, ...

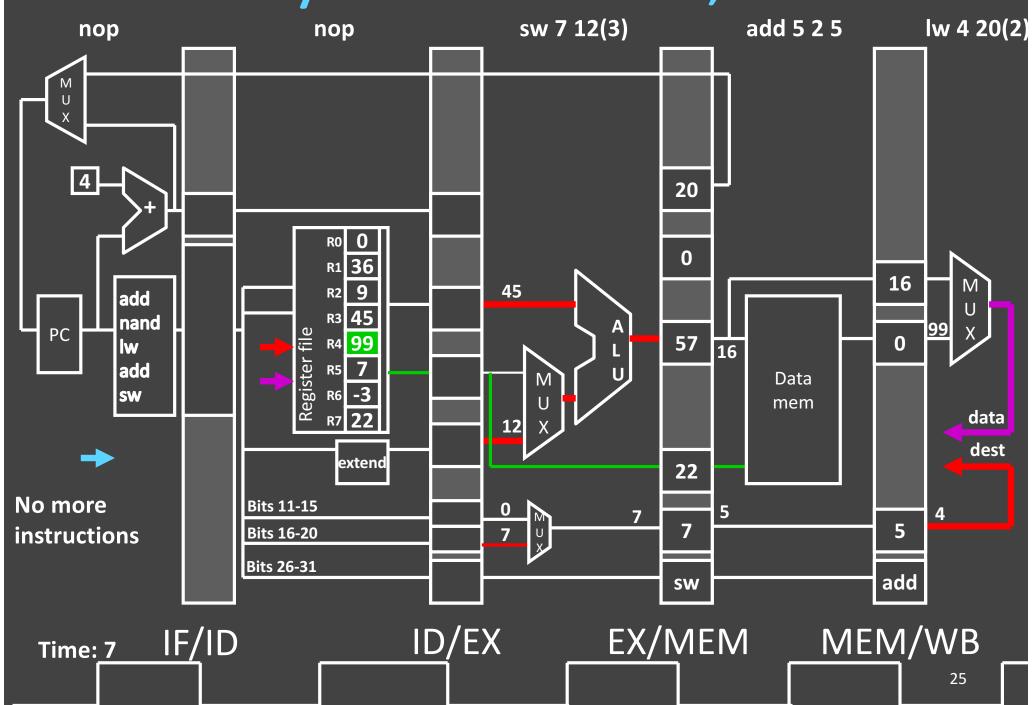


Cycle 5: Fetch sw, Decode add, ...

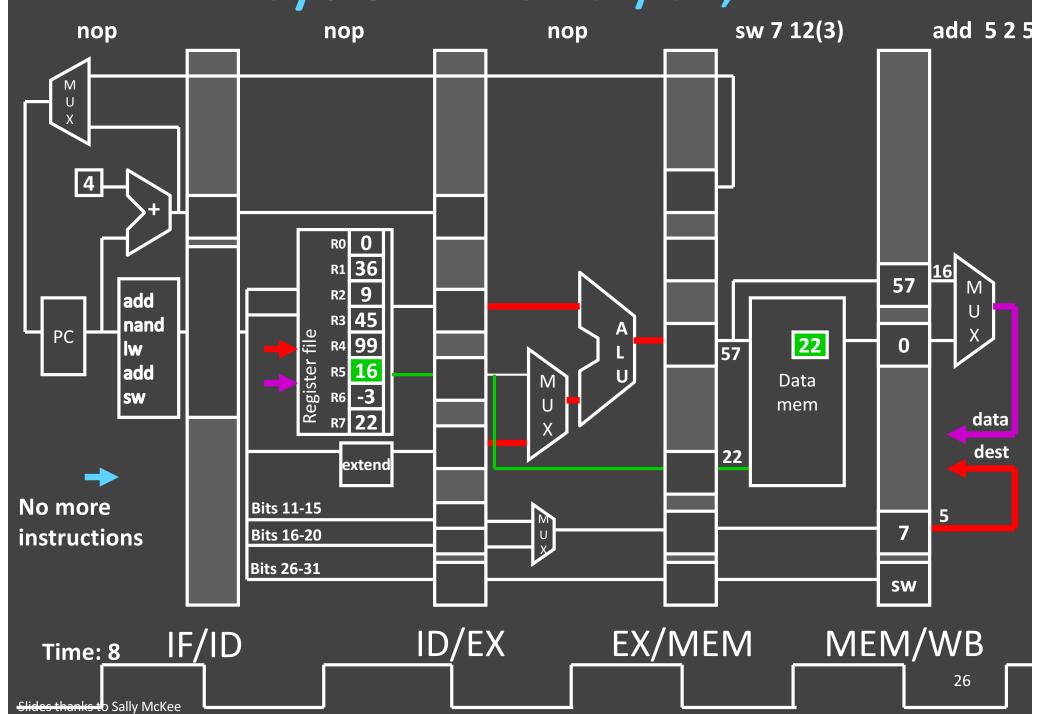




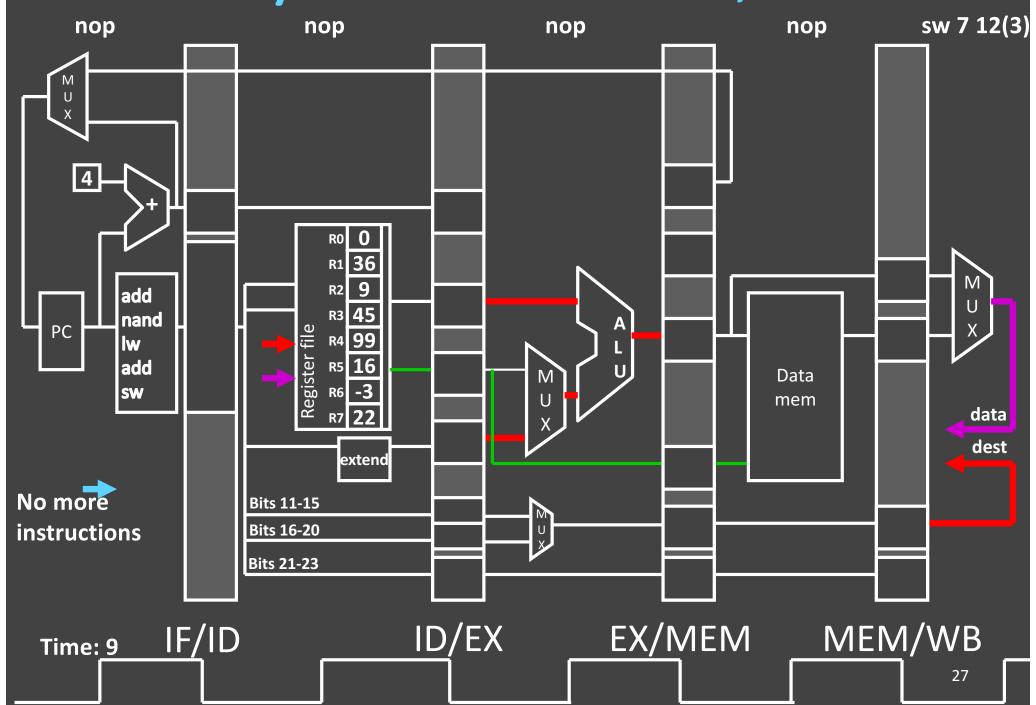
Cycle 7: Execute sw, ...



Cycle 7: Memory sw, ...



Cycle 7: Writeback sw, ...



iClicker Question

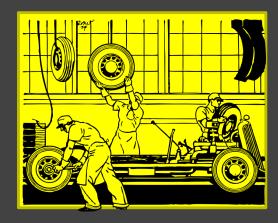
Pipelining is great because:

- A. You can fetch and decode the same instruction at the same time.
- B. You can fetch two instructions at the same time.
- C. You can fetch one instruction while decoding another.
- D. Instructions only need to visit the pipeline stages that they require.

Agenda

5-stage Pipeline

- Implementation
- Working Example







Hazards

- Structural
- Data Hazards
- Control Hazards

Hazards

Correctness problems associated w/processor design

1. Structural hazards

Same resource needed for different purposes at the same time (Possible: ALU, Register File, Memory)

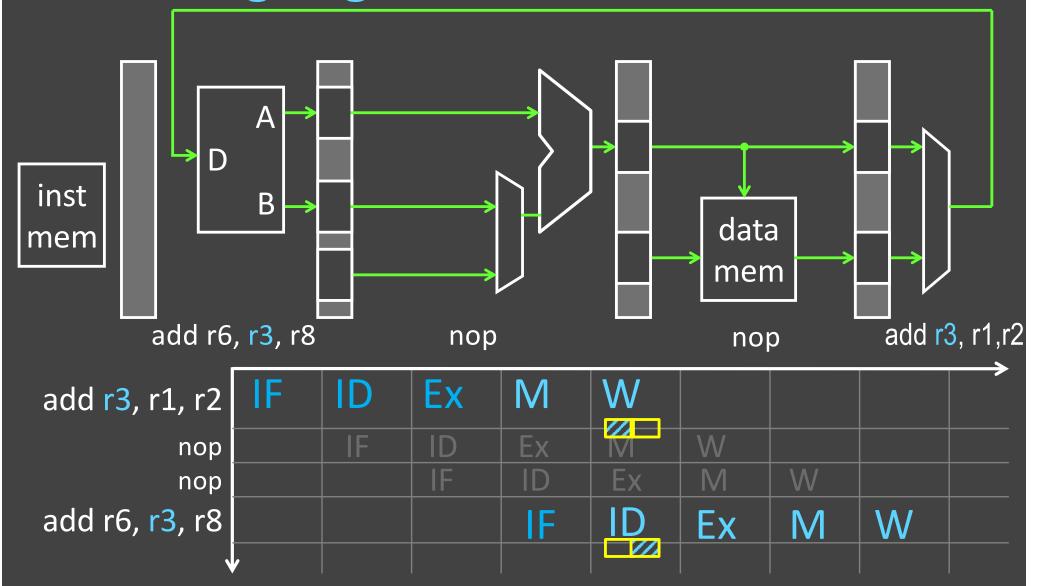
2. Data hazards

Instruction output needed before it's available

3. Control hazards

Next instruction PC unknown at time of Fetch

Resolving Register File Structural Hazard



Problem: Need to read a value that is currently being written

Solution: negate RF clock: write first half, read second half

Dependences and Hazards

Dependence: relationship between two insns

- Data: two insns use same storage location
- Control: 1 insn affects whether another executes at all
- Not a bad thing, programs would be boring otherwise
- Enforced by making older insn go before younger one
 - Happens naturally in single-/multi-cycle designs
 - But not in a pipeline

Hazard: dependence & possibility of wrong insn order

- Effects of wrong insn order cannot be externally visible
- Hazards are a bad thing: most solutions either complicate the hardware or reduce performance

iClicker Question

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

```
add r3 \leftarrow r1, r2 sub r5 \leftarrow r3, r4
```

- 1. Is there a dependence?
- 2. Is there a hazard?

$$Yes = A$$
 $No = B$

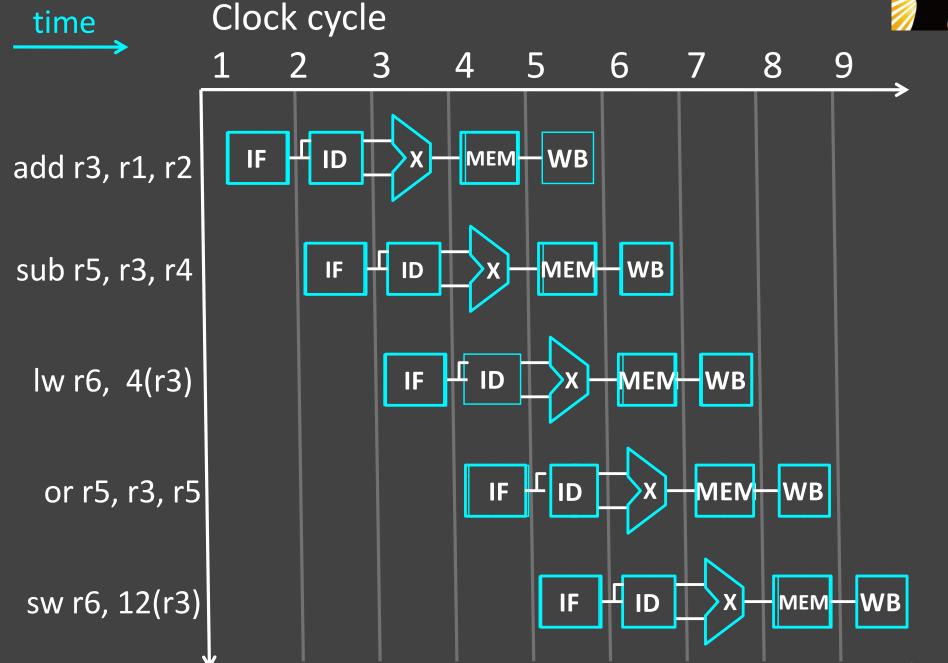
iClicker Follow-up Question

Which of the following statements is true?

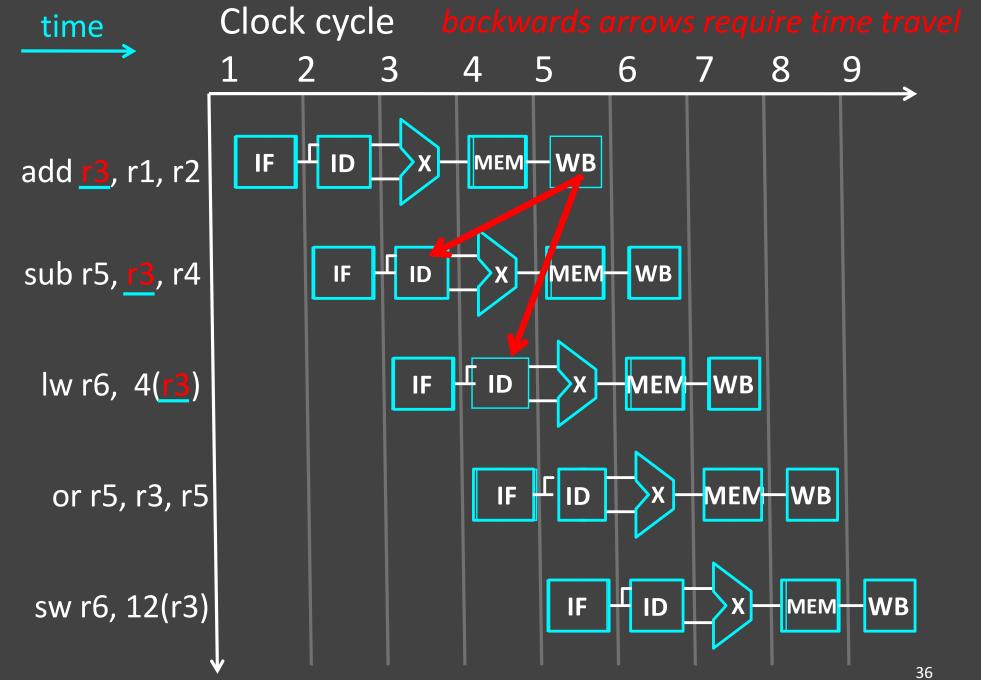
- A. Whether there is a data dependence between two instructions depends on the machine the program is running on.
- B. Whether there is a data hazard between two instructions depends on the machien the program is running on.
- C. Both A & B
- D. Neither A nor B

Where are the Data Hazards?

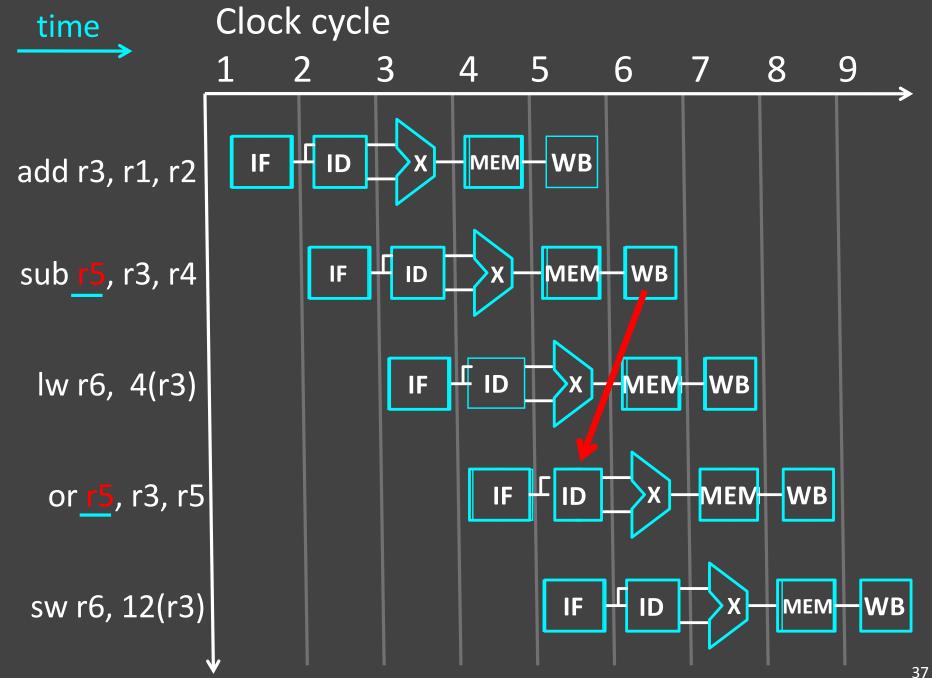




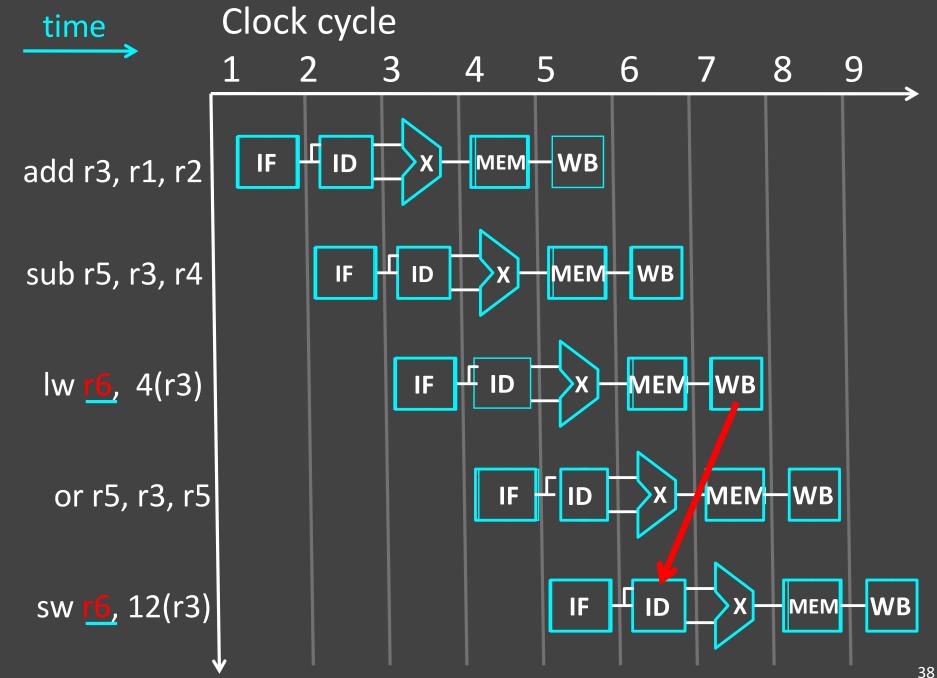
Visualizing Data Hazards (1)



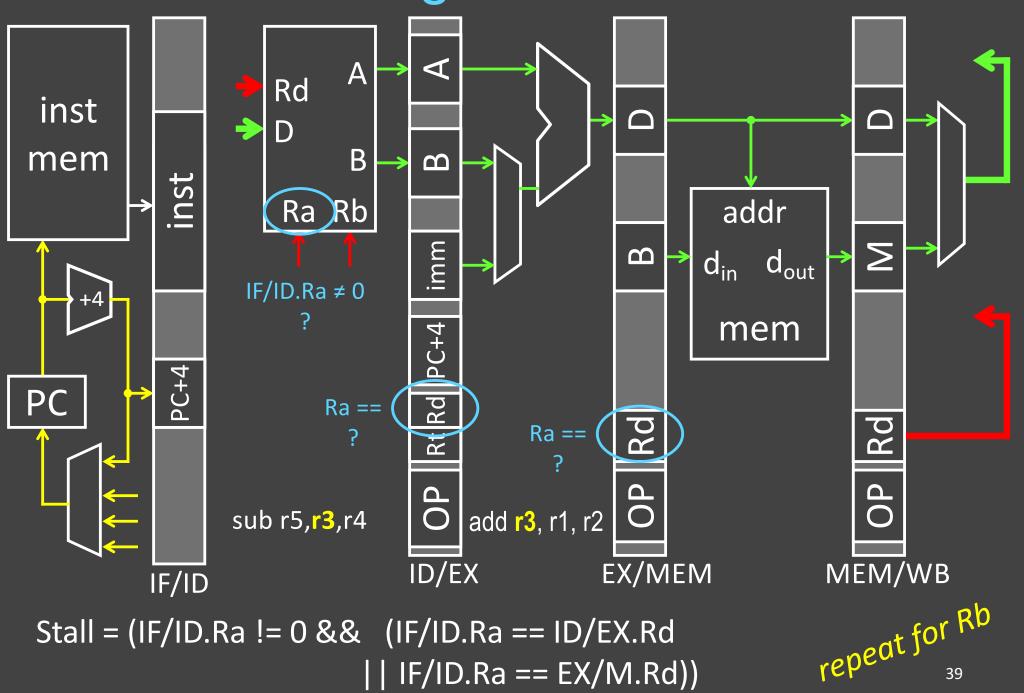
Visualizing Data Hazards (2)



Visualizing Data Hazards (3)



Detecting Data Hazards



Possible Responses to Data Hazards

1. Do Nothing

- Change the ISA to match implementation
- "Hey compiler: don't create code w/data hazards!"
 (We can do better than this)

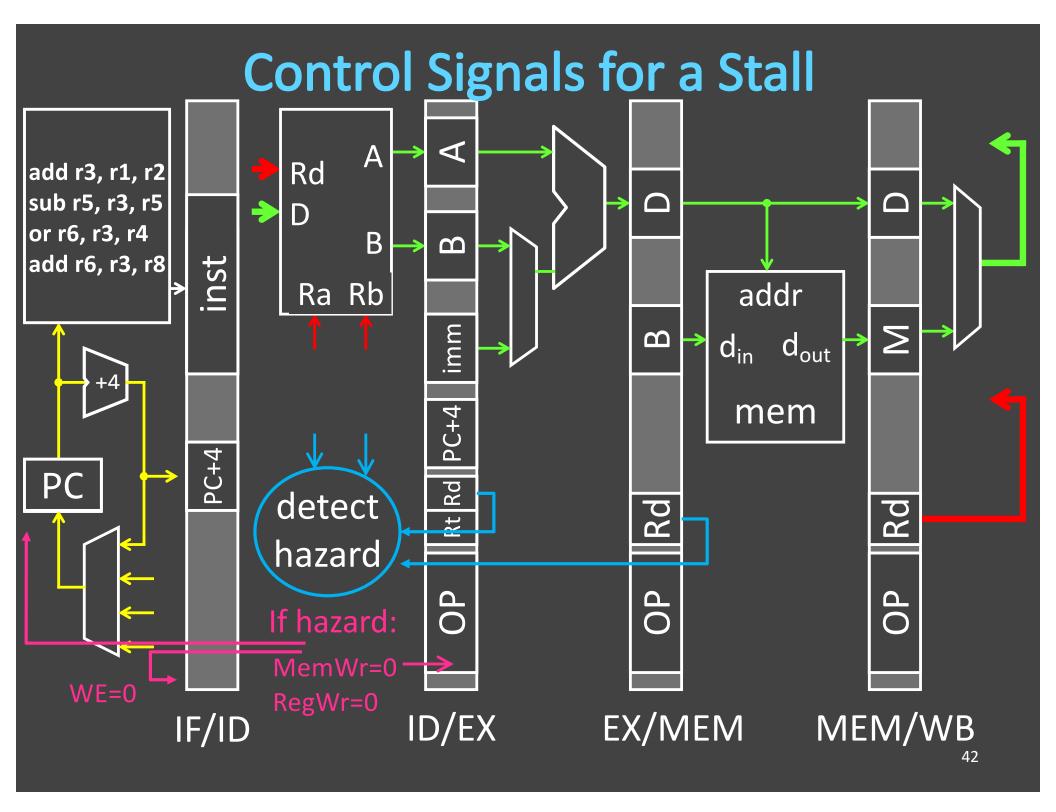
2. Stall

- Pause current and subsequent instructions till safe
- 3. Forward/bypass
 - Forward data value to where it is needed
 (Only works if value actually exists already)

Stalling

How to stall an instruction in ID stage

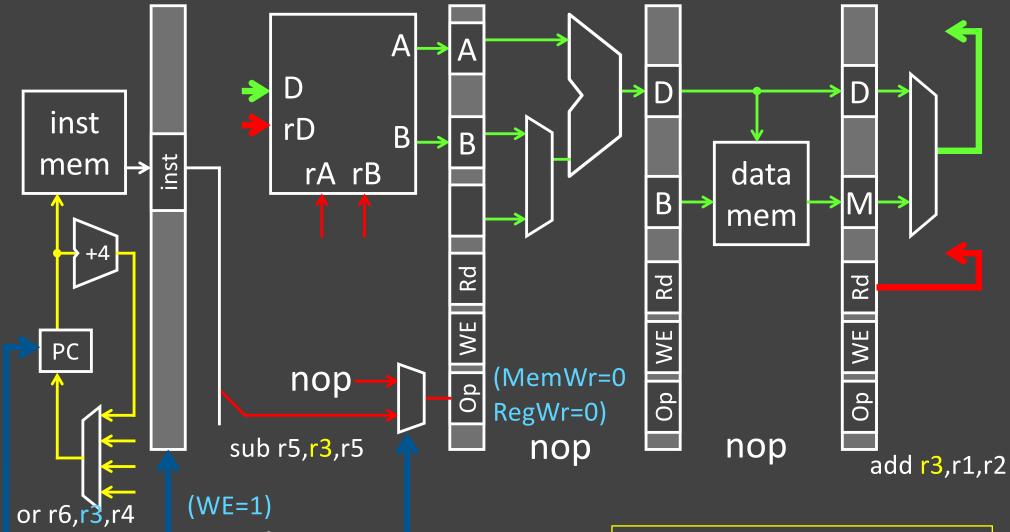
- prevent IF/ID pipeline register update
 - stalls the ID stage instruction
- convert ID stage insn into nop for later stages
 - innocuous "bubble" passes through pipeline
- prevent PC update
 - stalls the next (IF stage) instruction



Detecting the Hazard D inst rD В В inst mem rA rB data B mem (MemWr=0 WE RegWr=0) PC nop sub r5,**r3**,**r**5 add r3,r1,r2 (WE=0) or r6,r3,r4 /stall $NOP = If(IF/ID.rA \neq 0 \&\&$ (IF/ID.rA==ID/Ex.Rd ← STALL CONDITION MET IF/ID.rA==Ex/M.Rd)) 43

First Stall Cycle (nop in X) D inst rD В В mem inst data rA rB B mem (MemWr=0 RegWr=0) PC (MemWr=0 nop RegWr=0) sub r5,**r3**,r5 add r3,r1,r2 nop (WE=0) or r6,r3,r4 /stall NOP = $If(IF/ID.rA \neq 0 \&\&$ (IF/ID.rA==ID/Ex.Rd IF/ID.rA==Ex/M.Rd)) <- STALL CONDITION MET

Second Stall Cycle (nop in X, MEM)



/stall NOP = If(IF/ID.rA ≠ 0 && (IF/ID.rA==ID/Ex.Rd IF/ID.rA==Ex/M.Rd))

sub allowed to leave decode stage

NO STALL CONDITION MET:

Stalling



time	Clock cycle								
<u>time</u> →	1	2	3	4	5	6	7	8	>
add r3 , r1, r2									
sub r5, r3, r5									
or r6, r3, r4									
add r6, r3 , r8									
	,								46

Stalling

time	Clock cycle								
<u> </u>	1	2	3	4	5	6	7	8	
r3 = 10									
add r3, r1, r2	IF.			M	W				
r3 = 20					Z				
				2 Sta	II Cyc	les			
sub r5, r3 , r5				ID*	ID*	Ex	M	W	
							_		
or r6, r3, r4				IF*	IF*	ID	Ex	M	
							10	_	
add r6, r3 , r8						IF	ID	Ex	
									47

Possible Responses to Data Hazards

1. Do Nothing

- Change the ISA to match implementation
- "Compiler: don't create code with data hazards!"
 (Nice try, we can do better than this)

2. Stall

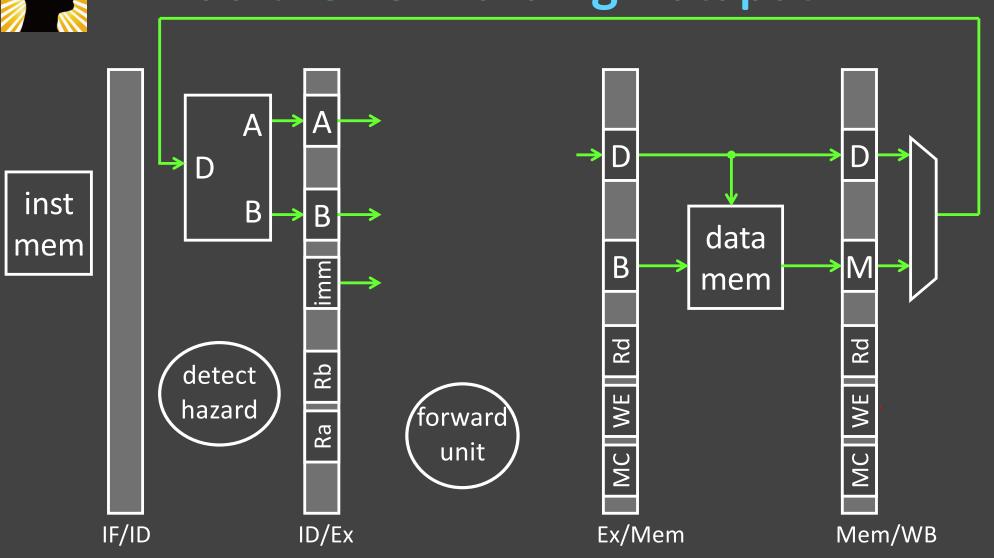
Pause current and subsequent instructions till safe

3. Forward/bypass

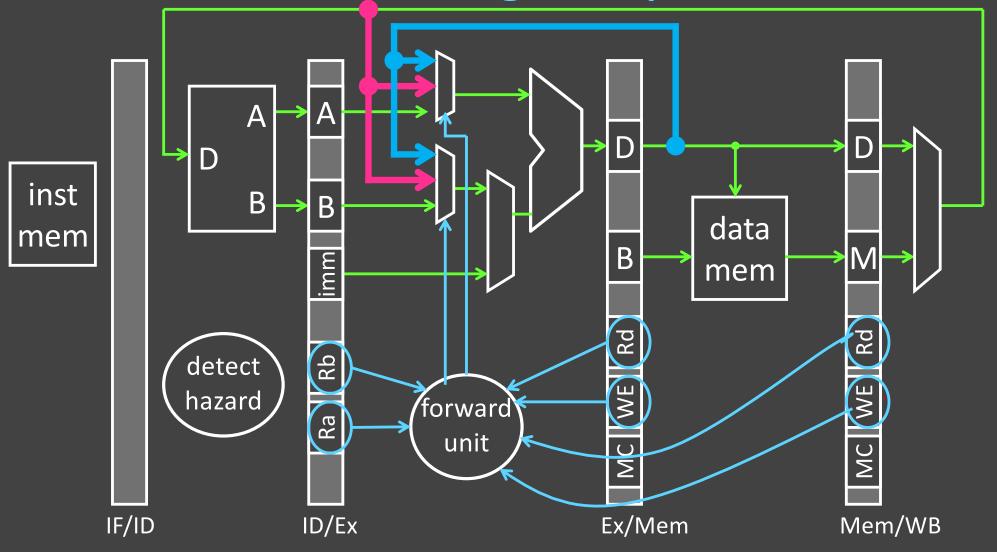
Forward data value to where it is needed
 (Only works if value actually exists already)



Add the Forwarding Datapath



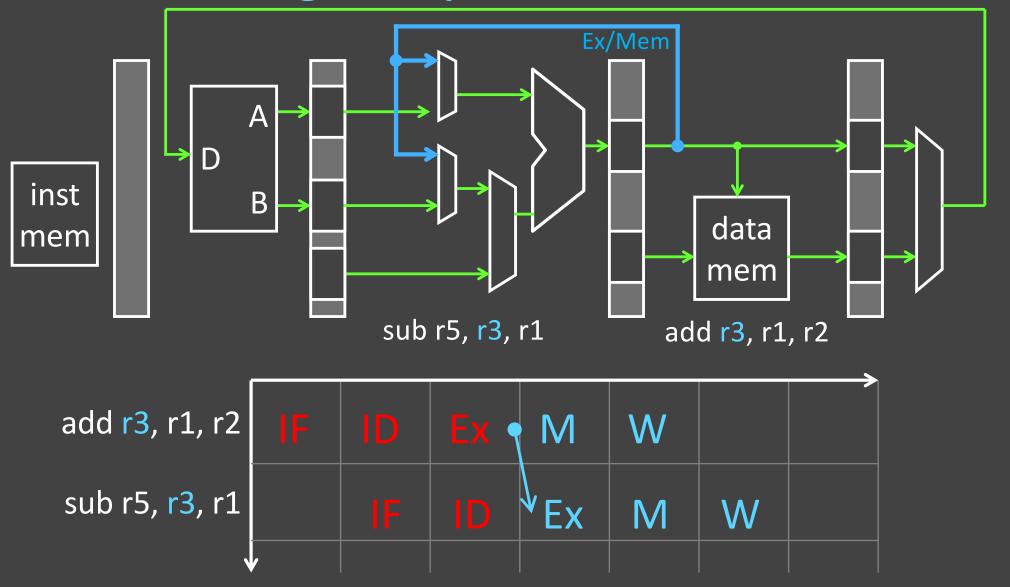
Forwarding Datapath



Two types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage $(M \rightarrow Ex)$
- Forwarding from Mem/WB register to Ex stage (W \rightarrow Ex)

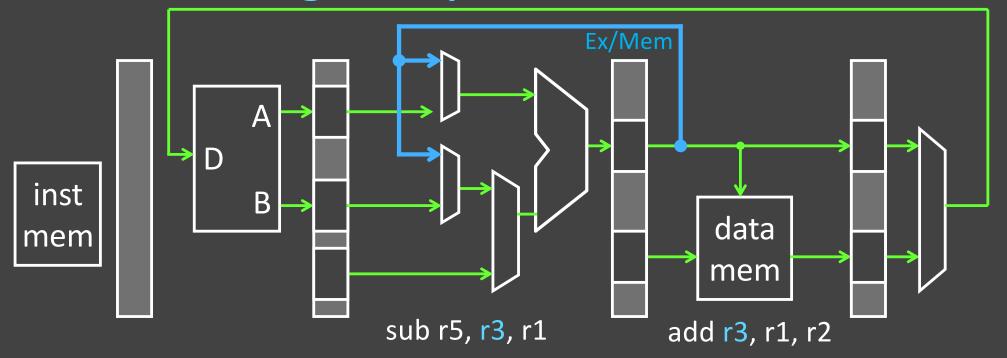
Forwarding Datapath 1: Ex/MEM → EX



Problem: EX needs ALU result that is in MEM stage

Solution: add a bypass from EX/MEM.D to start of EX

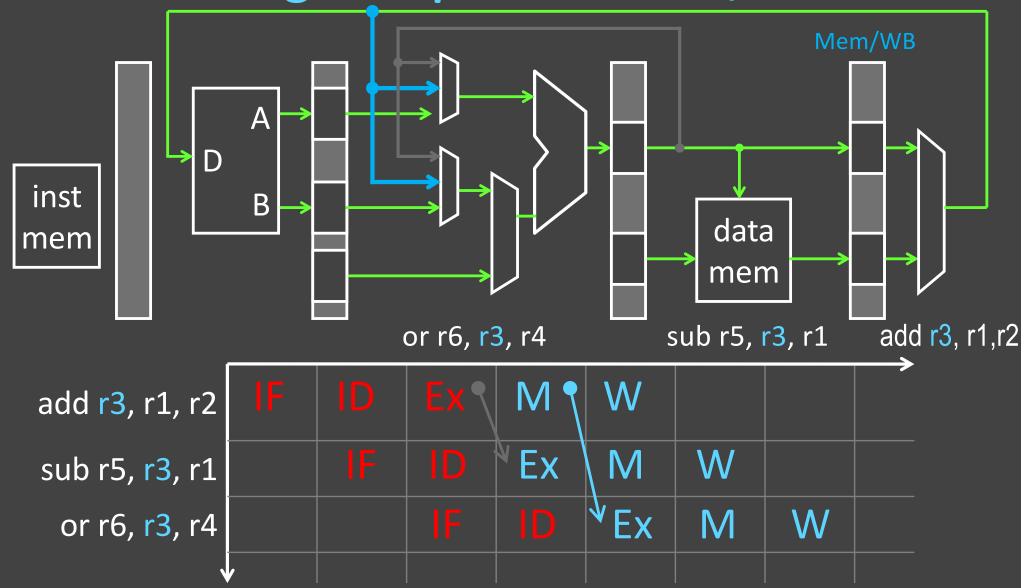
Forwarding Datapath 1: Ex/MEM -> EX



Detection Logic in Ex Stage:

```
forward = (Ex/M.WE && EX/M.Rd != 0 &&
ID/Ex.Ra == Ex/M.Rd)
|| (same for Rb)
```

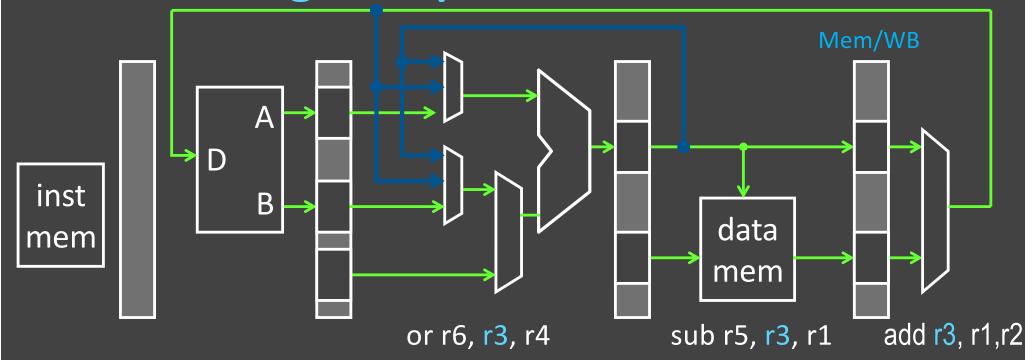
Forwarding Datapath 2: Mem/WB → EX



Problem: EX needs value being written by WB

Solution: Add bypass from WB final value to start of EX

Forwarding Datapath 2: Mem/WB -> EX



Detection Logic:

Forwarding Example 2 Clock cycle time 3 2 6 7 8 add r3, r1, r2 sub r5, r3, r4 lw r6, 4(r3) or r5, r3, r5 sw r6, 12(r3) 55

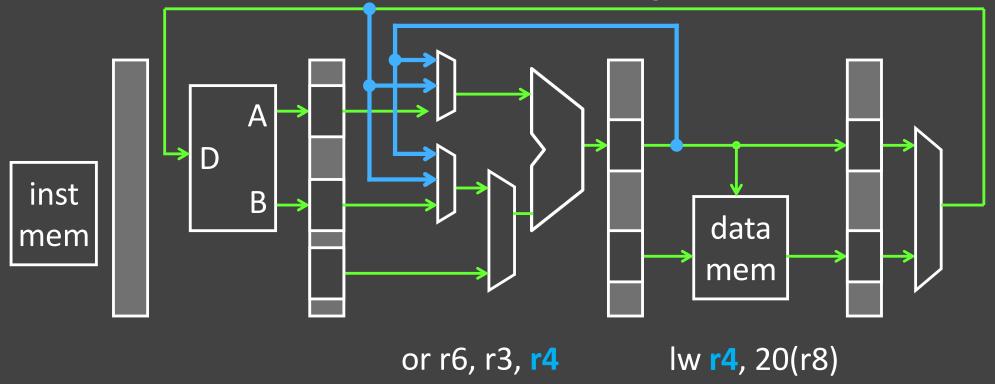
Forwarding Example 2 Clock cycle 1 2 3 5 6 7 8 IF ID W Ex M IF ID Ex M W M W



time

add r3, r1, r2

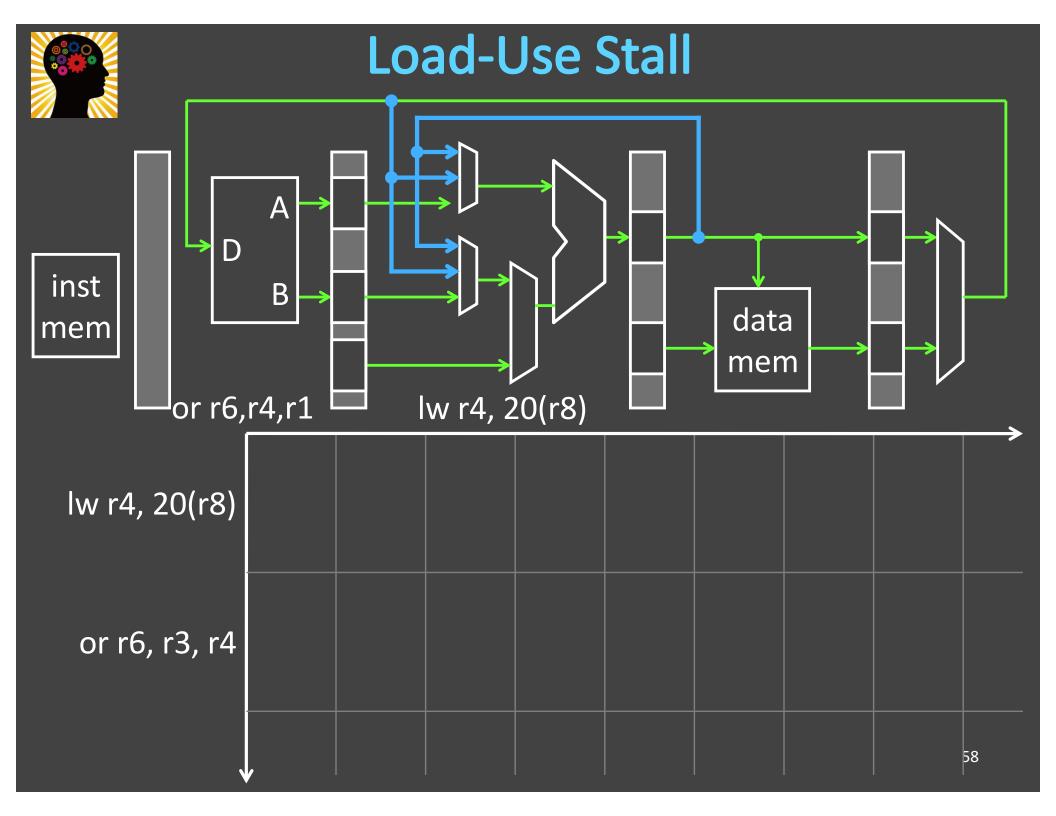
Load-Use Hazard Explained



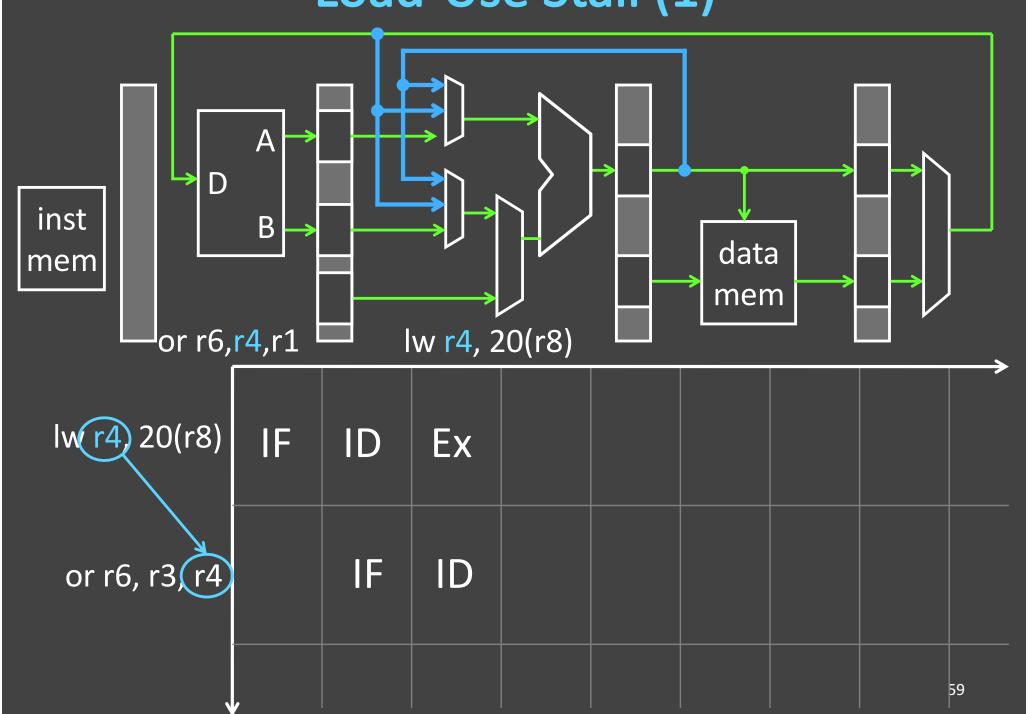
Data dependency after a load instruction:

- Value not available until after the M stage
- → Next instruction cannot proceed if dependent

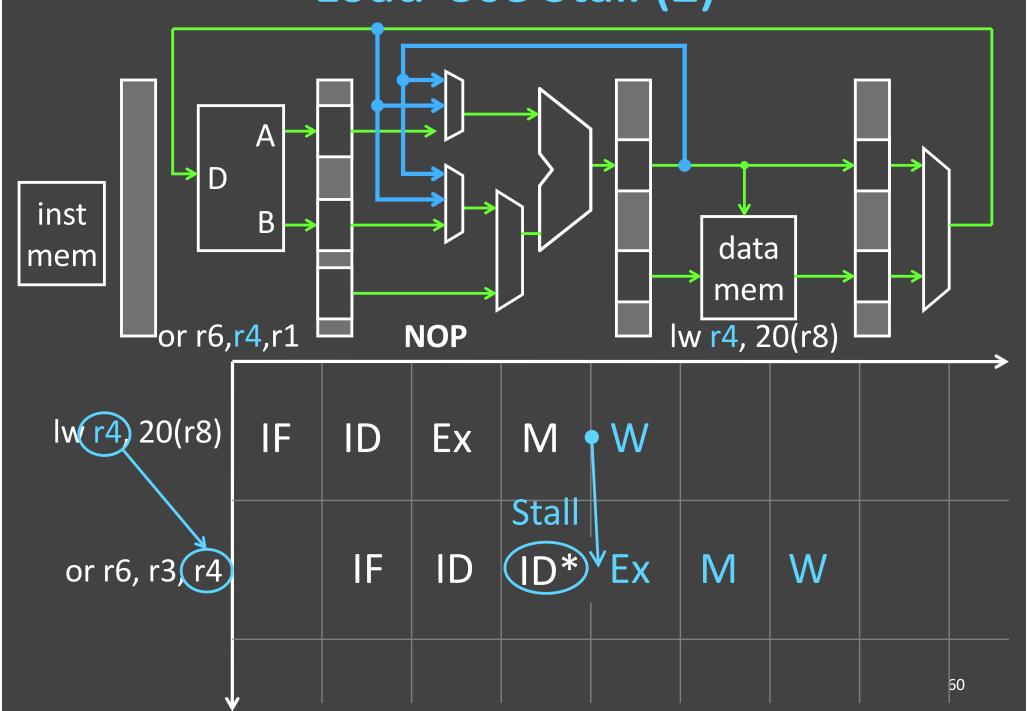
THE KILLER HAZARD



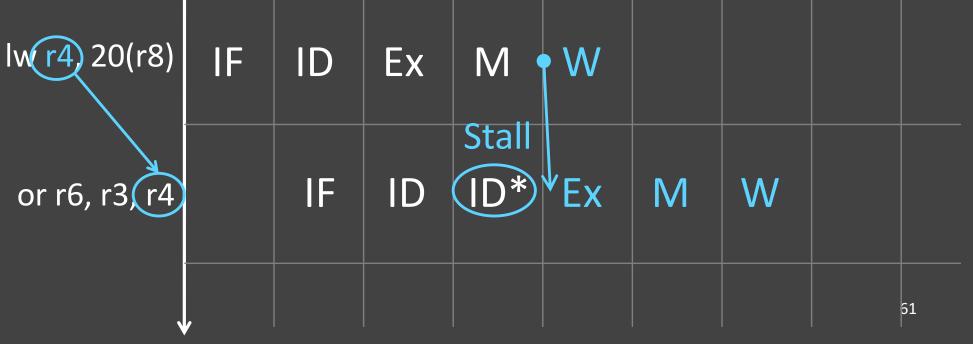
Load-Use Stall (1)



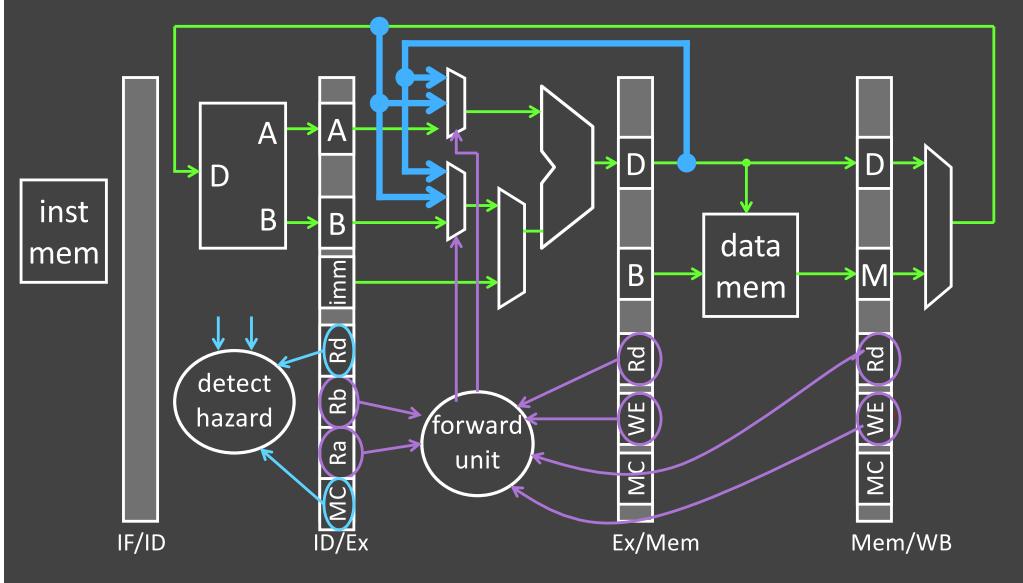
Load-Use Stall (2)



Load-Use Stall (3) inst B data mem mem lw r4, 20 NOP or r6,r4,r1 lw(r4) 20(r8) IF ID Ex M Stall

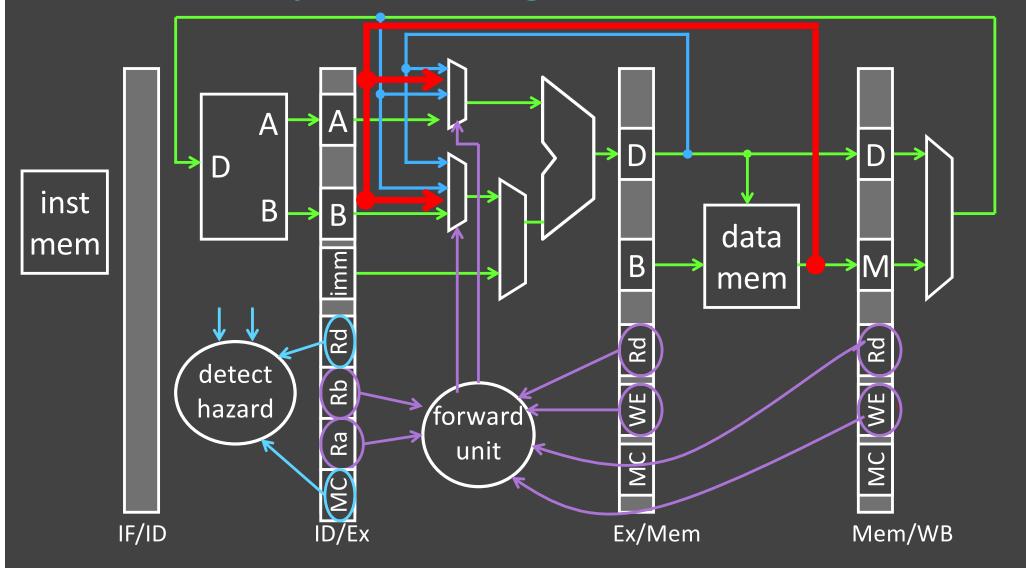


Load-Use Detection



Stall = If(ID/Ex.MemRead && IF/ID.Ra == ID/Ex.Rd

Incorrectly Resolving Load-Use Hazards



Most frequent 3410 **non-solution** to load-use hazards Why is this "solution" so so so so so awful?

iClicker Question

Forwarding values directly from Memory to the Execute stage without storing them in a register first:

- A. Does not remove the need to stall.
- B. Adds one too many possible inputs to the ALU.
- C. Will cause the pipeline register to have the wrong value.
- D. Halves the frequency of the processor.
- E. Both A & D

Resolving Load-Use Hazards

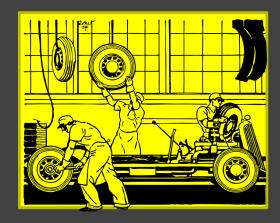
Two MIPS Solutions:

- MIPS 2000/3000: delay slot
 - ISA says results of loads are not available until one cycle later
 - Assembler inserts nop, or reorders to fill delay slot
- MIPS 4000 onwards: stall
 - But really, programmer/compiler reorders to avoid stalling in the load delay slot

Agenda

5-stage Pipeline

- Implementation
- Working Example







Hazards

- Structural
- Data Hazards
- Control Hazards

Control Hazards

Control Hazards

- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)
- → next PC not known until *2 cycles after* branch/jump

0x10: beq r1, r2, L

0x14: add r3, r0, r3

0x18: sub r5, r4, r6

0x1C: L: or r3, r2, r4

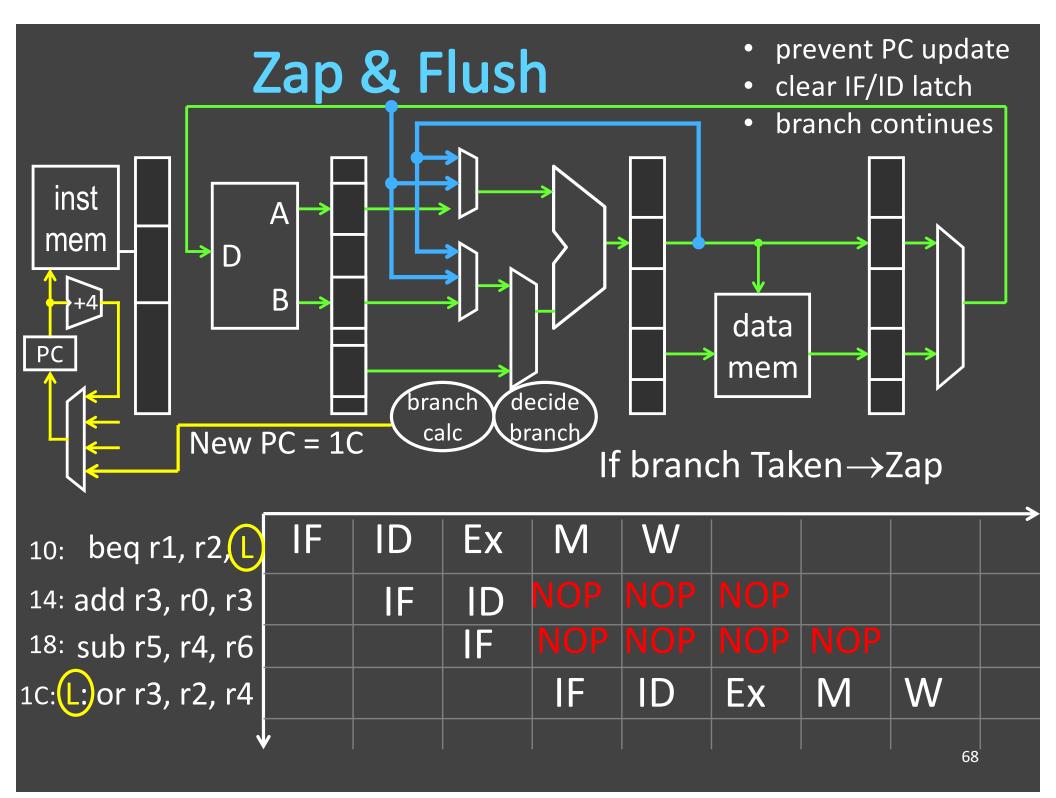
Branch *not* taken?

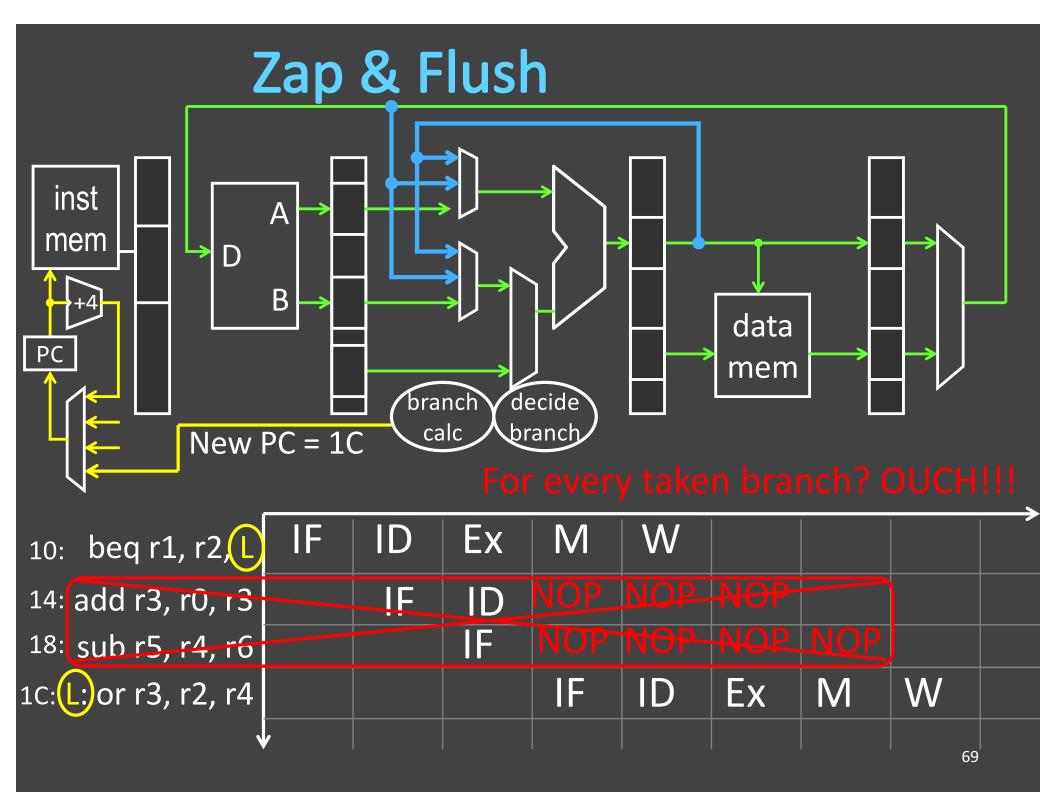
No Problem!

Branch taken?

Just fetched add, sub...

→ Zap & Flush





Reducing the cost of control hazard

1. Delay Slot

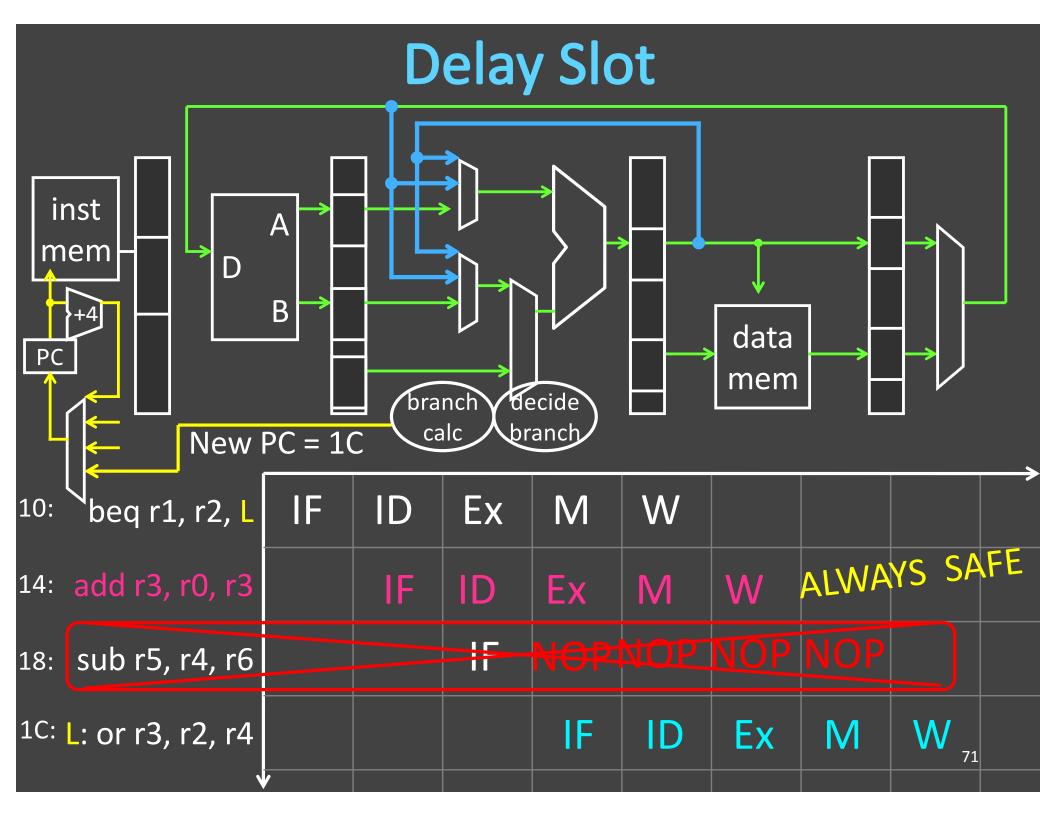
- You MUST do this
- MIPS ISA: 1 insn after ctrl insn always executed
 - Whether branch taken or not

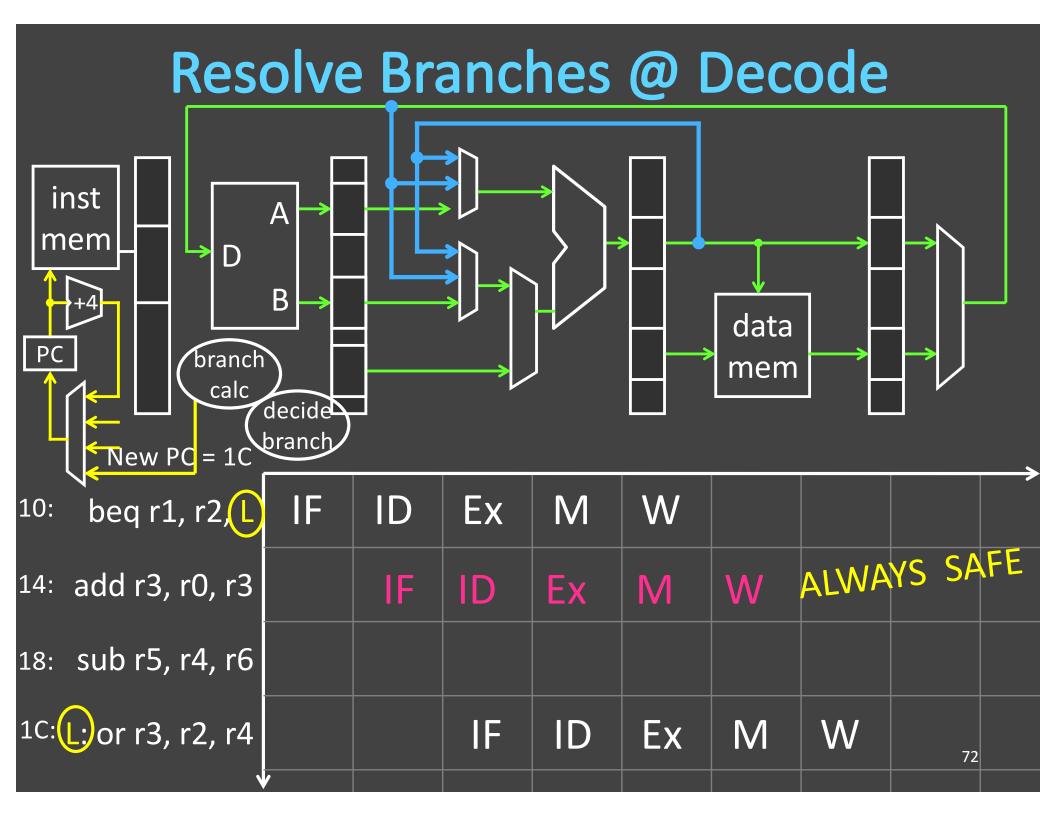
2. Resolve Branch at Decode

- Some groups do this for Project 2, your choice
- Move branch calc from EX to ID
- Alternative: just zap 2nd instruction when branch taken

3. Branch Prediction

 Not in 3410, but every processor worth anything does this (no offense!)





Branch Prediction

Most processor support Speculative Execution

- Guess direction of the branch
 - Allow instructions to move through pipeline
 - Zap them later if guess turns out to be wrong
- A must for long pipelines

Data Hazard Takeaways

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. Pipelined processors need to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards. Stalling introduces NOPs ("bubbles") into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Nops significantly decrease performance.

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register). Better performance than stalling.

Control Hazard Takeaways

Control hazards occur because the PC following a control instruction is not known until control instruction is executed. If branch is taken \rightarrow need to zap instructions. 1 cycle performance penalty.

Delay Slots can potentially increase performance due to control hazards. The instruction in the delay slot will *always* be executed. Requires software (compiler) to make use of delay slot. Put nop in delay slot if not able to put useful instruction in delay slot.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage. With a delay slot, this removes the need to flush instructions on taken branches.