## Pipelining

Hakim Weatherspoon 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.

#### **Announcements**

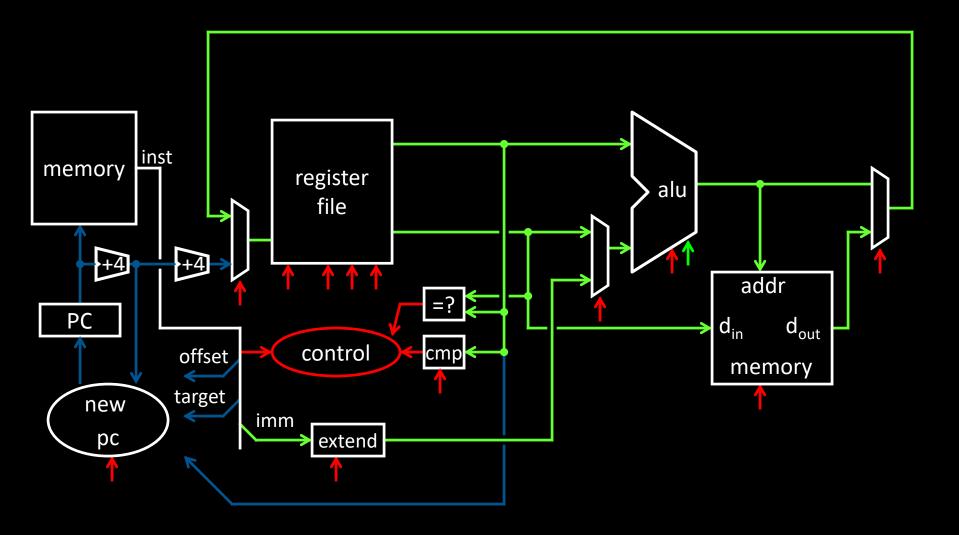
C Programming Practice Assignment due next Tuesday Short. Do not wait till the end.

Project 2 design doc

Critical to do this, else Project 2 will be hard

## Single Cycle vs Pipelined Processor

## Review: Single Cycle Processor



#### Review: Single Cycle Processor

#### Advantages

Single cycle per instruction make logic and clock simple

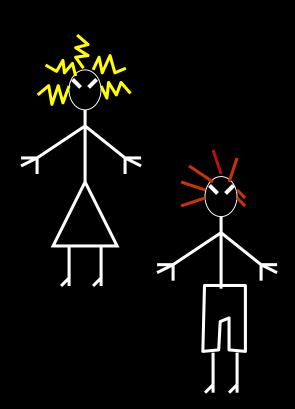
#### Disadvantages

- Since instructions take different time to finish, memory and functional unit are not efficiently utilized
- Cycle time is the longest delay
  - Load instruction
- Best possible CPI is 1 (actually < 1 w parallelism)</li>
  - However, lower MIPS and longer clock period (lower clock frequency); hence, lower performance

#### The Kids

Alice

Bob

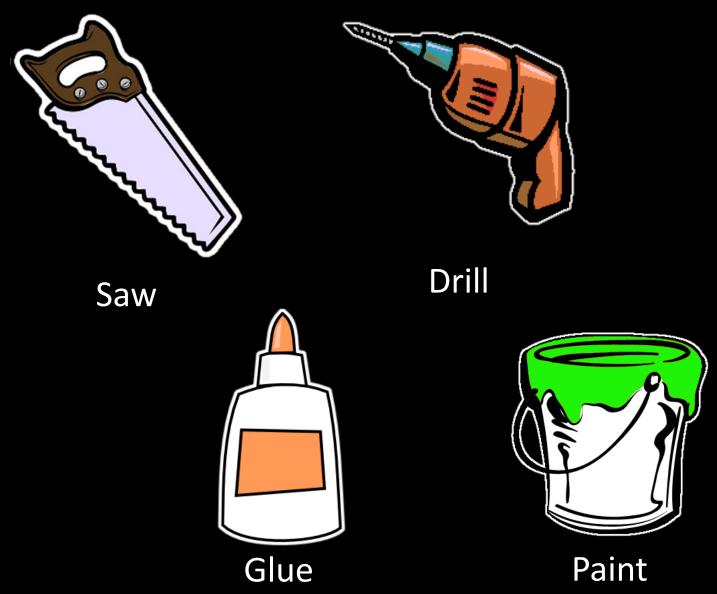


They don't always get along...

## The Bicycle

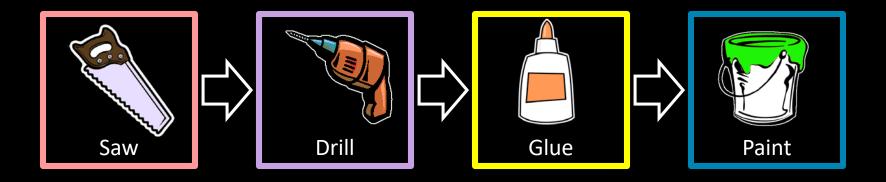


## The Materials

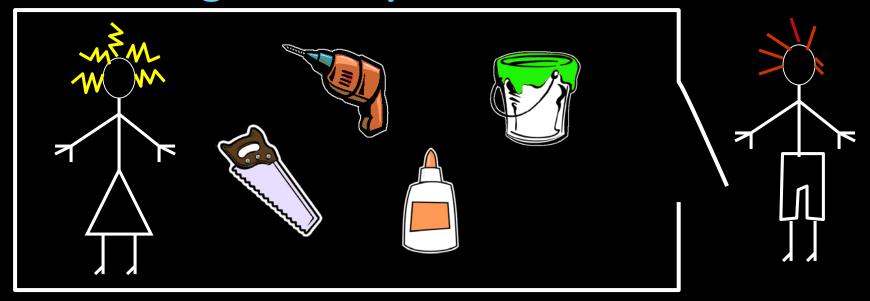


#### The Instructions

N pieces, each built following same sequence:



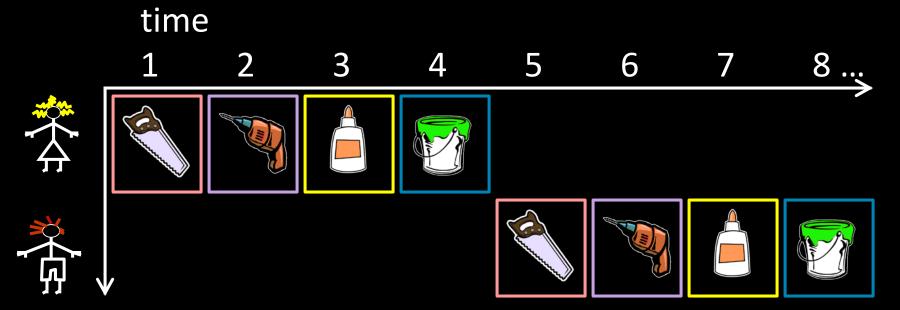
#### Design 1: Sequential Schedule



Alice owns the room

Bob can enter when Alice is finished
Repeat for remaining tasks
No possibility for conflicts

#### **Sequential Performance**



Latency:

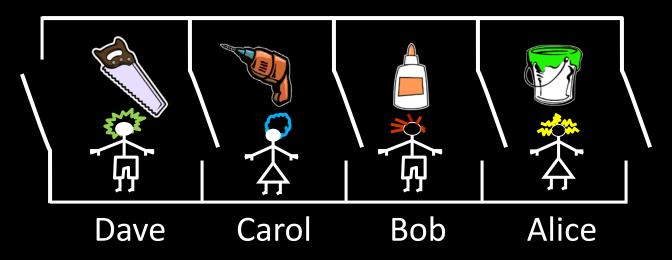
Throughput:

**Concurrency:** 

Can we do better?

#### Design 2: Pipelined Design

Partition room into stages of a pipeline



One person owns a stage at a time

4 stages

4 people working simultaneously

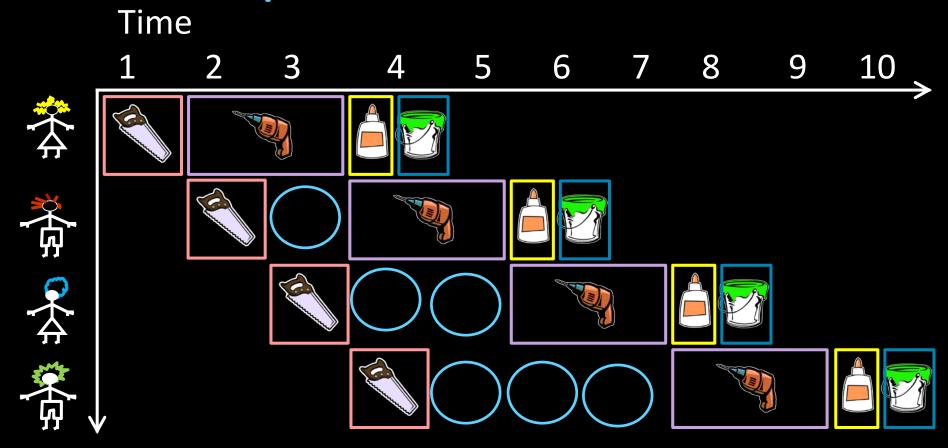
Everyone moves right in lockstep

It still takes all four stages for one job to complete

#### Pipelined Performance

time 6 7... Latency: Throughput: Concurrency: CPI =

#### Pipelined Performance



What if drilling takes twice as long, but gluing and paint take ½ as long?

Latency:

Throughput:

CPI =

#### Lessons

#### Principle:

Throughput increased by parallel execution Balanced pipeline very important

Else slowest stage dominates performance

#### Pipelining:

- Identify pipeline stages
- Isolate stages from each other
- Resolve pipeline hazards (next lecture)

## Single Cycle vs Pipelined Processor

## Single Cycle → Pipelining

#### Single-cycle

insn0.fetch, dec, exec insn1.fetch, dec, exec

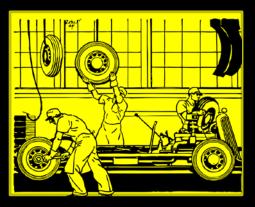
#### **Pipelined**

insn0.fetch	insn0.dec	insn0.exec	
	insn1.fetch	insn1.dec	insn1.exec

## Agenda

#### 5-stage Pipeline

- Implementation
- Working Example





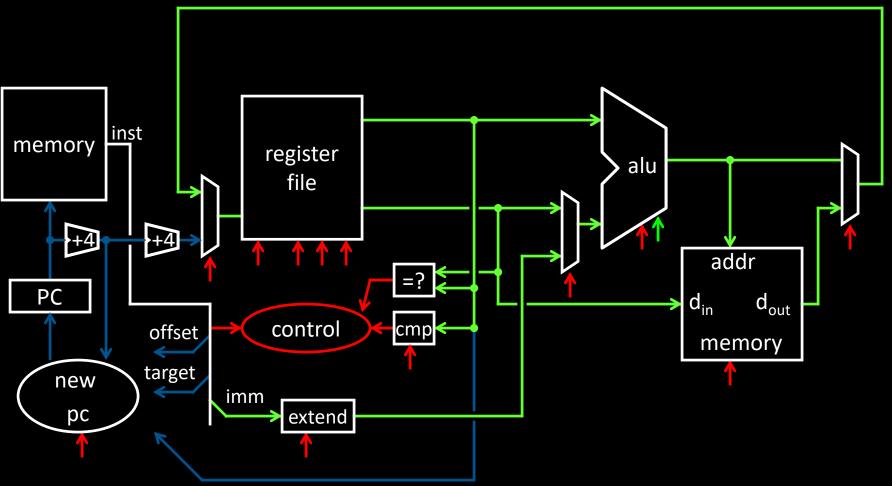


#### Hazards

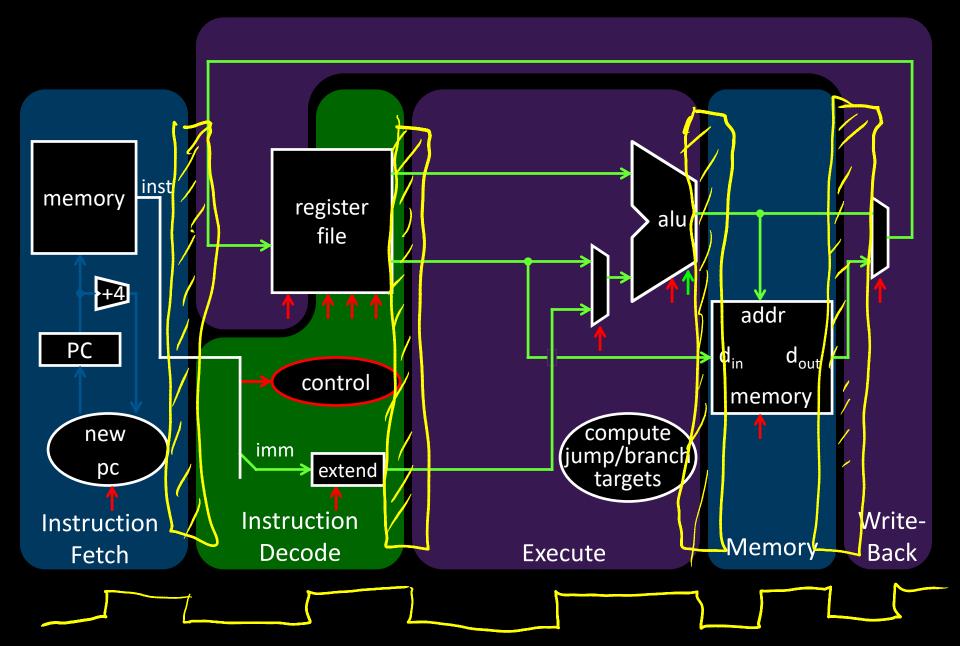
- Structural
- Data Hazards
- Control Hazards

#### A Processor

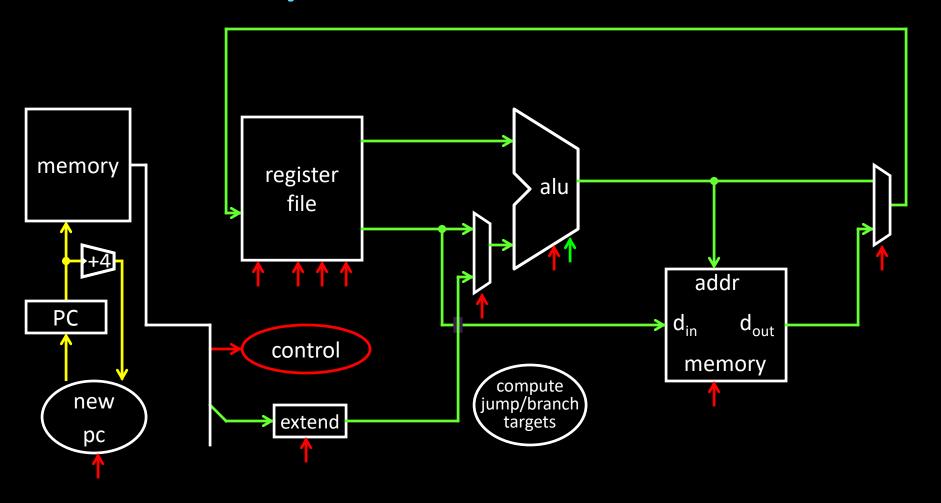
Review: Single cycle processor

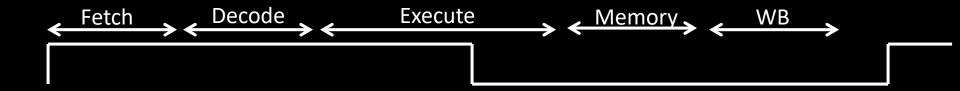


#### **A Processor**



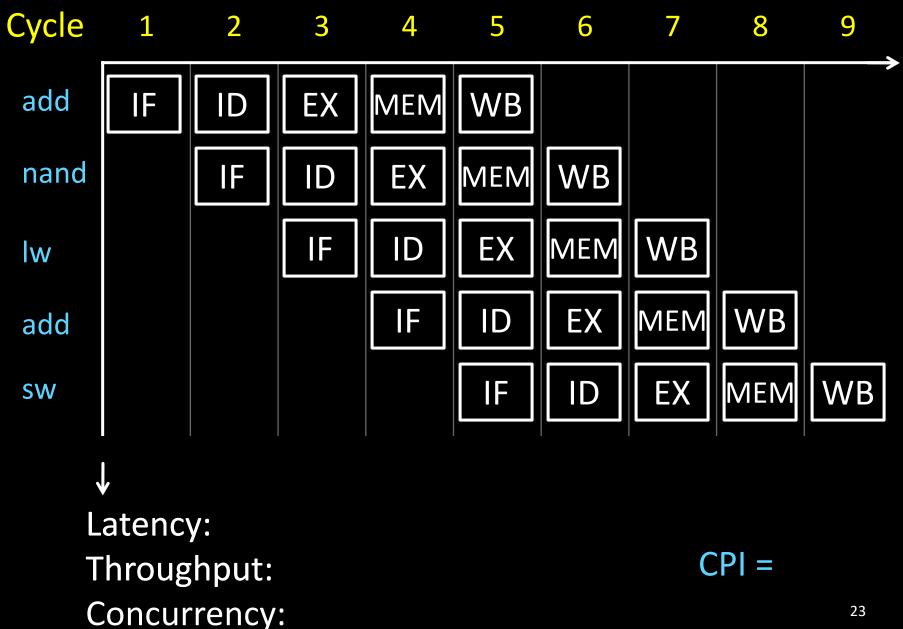
## **Pipelined Processor**





#### Dinalinad Drocaccor 4 memory register alu file $\mathbf{B}$ addr inst $\mathsf{d}_{\mathsf{in}}$ $d_{out}$ В control memory compute jump/branch new imm extend targets рс Write-Instruction Instruction ctrl Memory Decode Execute Back Fetch IF/ID MEM/WB ID/EX EX/MEM

### **Time Graphs**

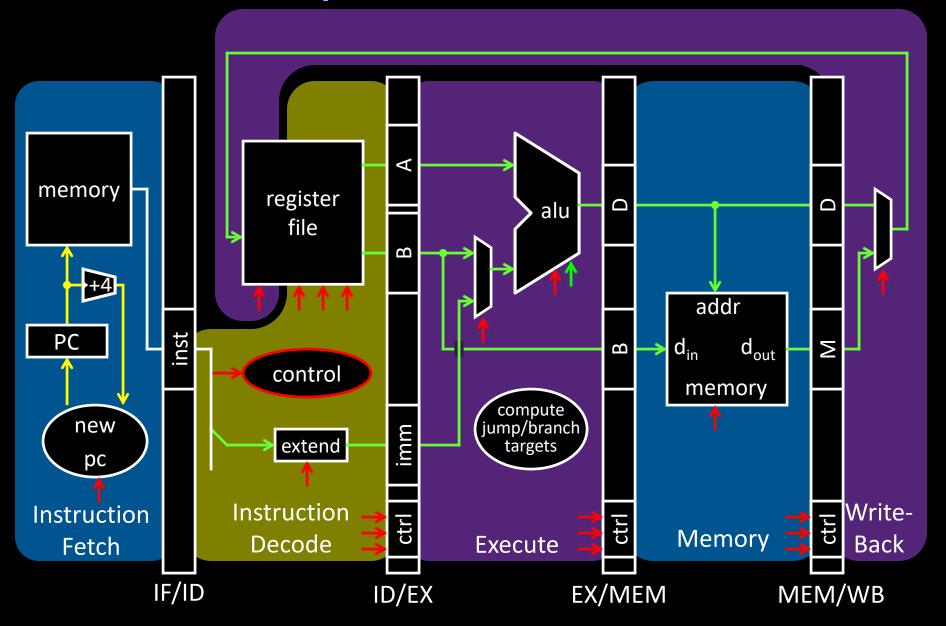


23

## 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

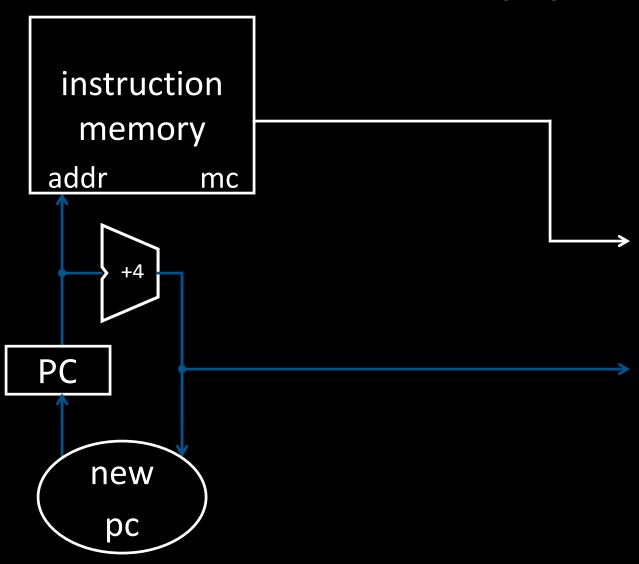
#### Pipelined Processor



## 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 (IF)



IF/ID

WE Rd register D file B Rb Ra

 $\mathbf{\Omega}$ 

Rest of pipeline

A

ctrl ||PC+4||imm

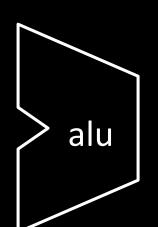
ID/EX

A

 $\mathbf{B}$ 

imm

Execute (EX)



Rest of pipeline  $\mathbf{\Omega}$ target ctrl

EX/MEM

ID/EX

ctrl

Stage 3: Execute

target B D

addr d<sub>in</sub> d<sub>out</sub>

memory

**MEM** 

mc

Rest of pipeline

MEM/WB

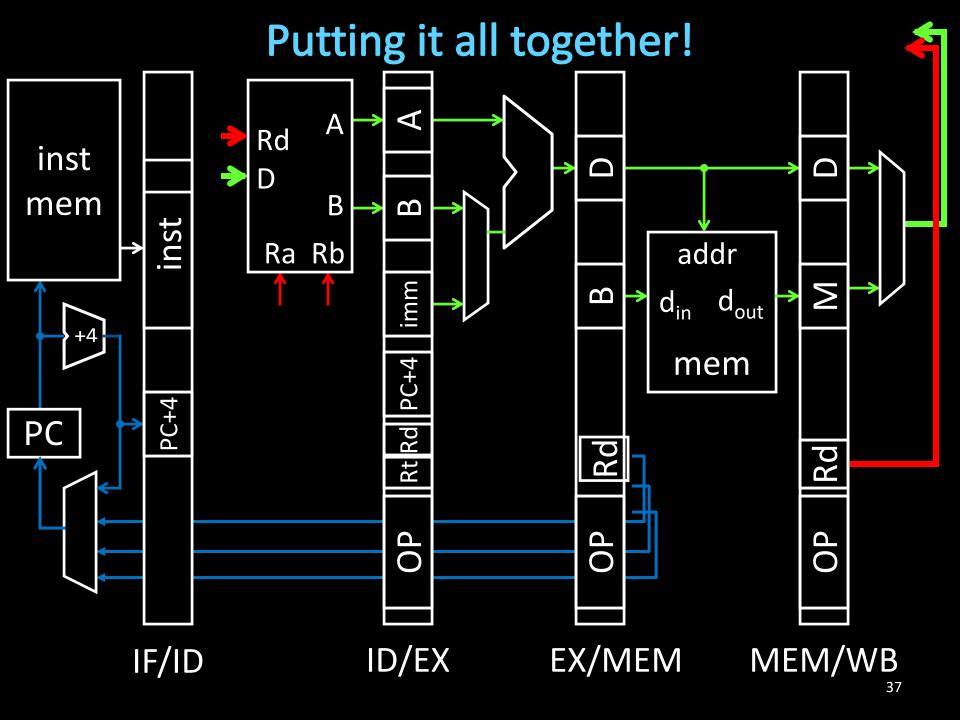
ctrl

 $\boxtimes$ 

EX/MEM

ctrl

4: Memory  $\boxtimes$ Stage MEM/WB WB



#### **Takeaway**

# Pipelining is a powerful technique to mask latencies and increase throughput

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel
  - Instruction level parallelism

#### Abstraction promotes decoupling

Interface (ISA) vs. implementation (Pipeline)

## MIPS is designed for pipelining

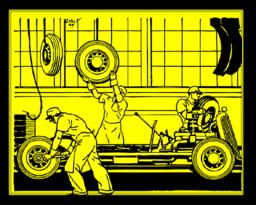
- Instructions same length
  - 32 bits, easy to fetch and then decode

- 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

## Agenda

#### 5-stage Pipeline

- Implementation
- Working Example







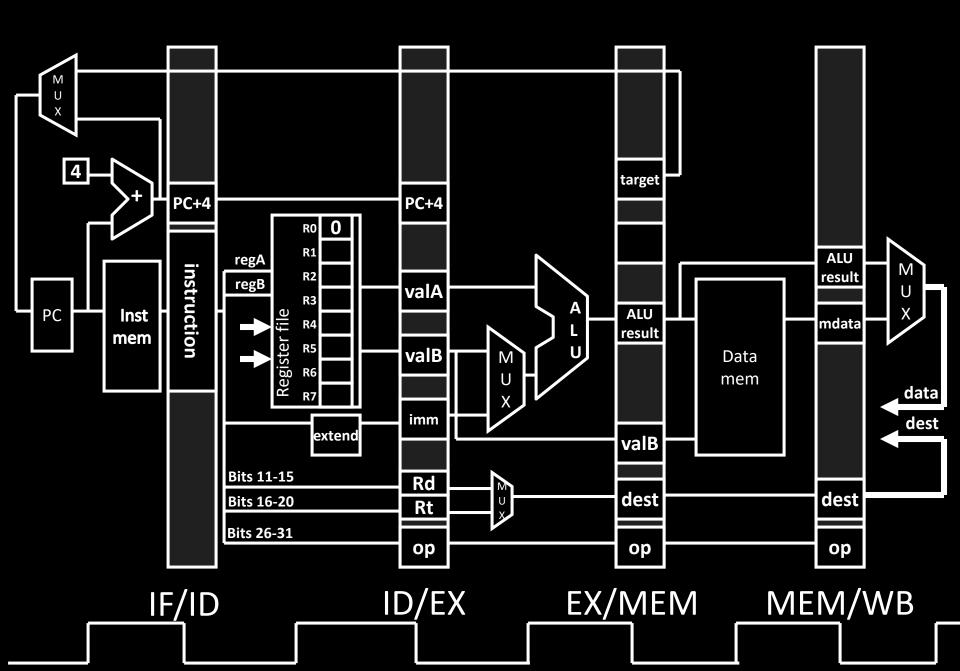
#### Hazards

- 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

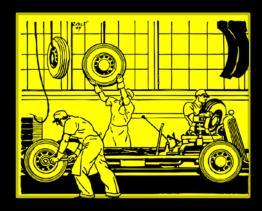


Fetch Example: Start State @ Cycle 0 add r3 r1 r2 M U 0 0 R1 36 R2 9 R3 12 R4 18 0 0 M add 0 nop nand 0 0 lw add 0 M **Data** SW mem data 0 dest extend 0 **Initial** Bits 11-15 0 Bits 16-20 **State** 0 Bits 26-31 nop nop nop MEM/WB ID/EX EX/MEM IF/ID Time: 0

# 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

# 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

### **Data Hazards**

#### 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
  - i.e instruction may need values that are being computed further down the pipeline
  - in fact, this is quite common

# Where are the Data Hazards?



time	CIOCK Cycle									
	1	2	3	4	5	6	7	8	9	
add r3, r1, r2										
sub r5, r3, r4										
lw r6, 4(r3)										
or r5, r3, r5										
sw r6, 12(r3)										

#### **Data Hazards**

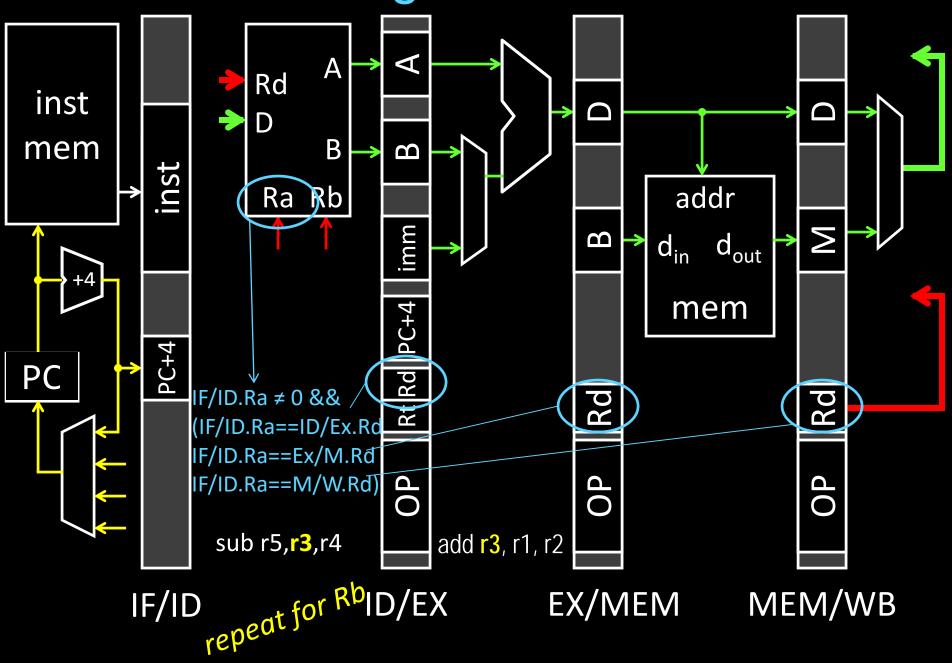
#### **Data Hazards**

- 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

```
i.e. add r3, r1, r2 sub r5, r3, r4
```

How to detect?

# **Detecting Data Hazards**



### **Data Hazards**

#### 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

### How to detect? Logic in ID stage:

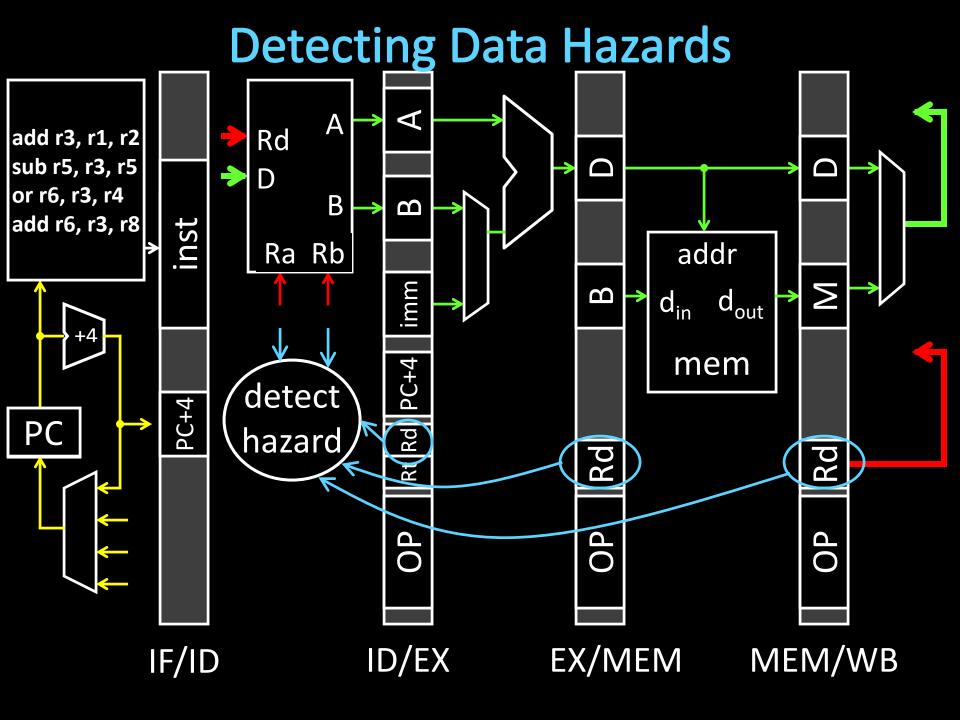
```
stall = (IF/ID.Ra != 0 &&

(IF/ID.Ra == ID/EX.Rd ||

IF/ID.Ra == EX/M.Rd ||

IF/ID.Ra == M/WB.Rd))

|| (same for Rb)
```



### **Takeaway**

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

### **Next Goal**

What to do if data hazard detected?

### Possible Responses to Data Hazards

#### 1. Do Nothing

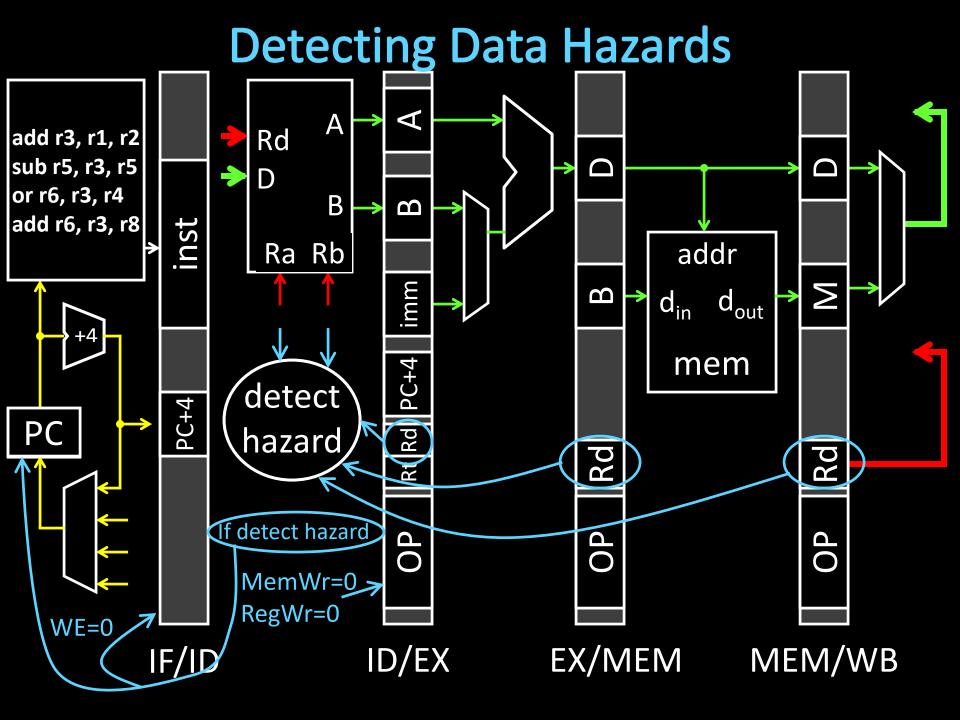
- Change the ISA to match implementation
- "Hey compiler: don't create code w/data hazards!"
   (We can do better than this)

#### 2. Stall

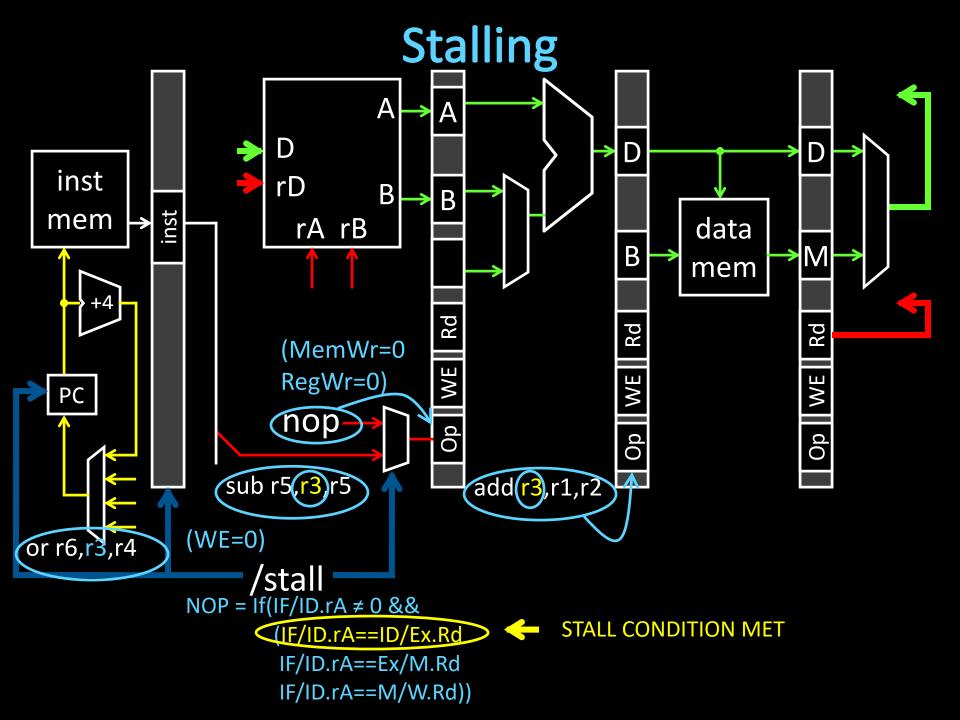
- Pause current and subsequent instructions till safe
- 3. Forward/bypass
  - Forward data value to where it is needed
     (Only works if value actually exists already)

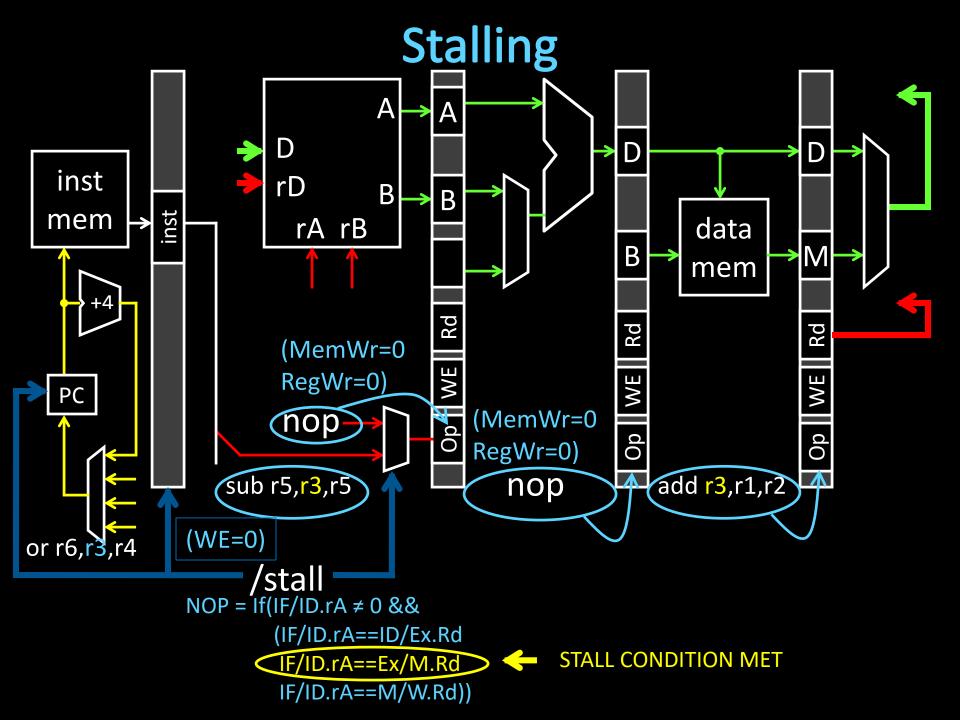
#### How to stall an instruction in ID stage

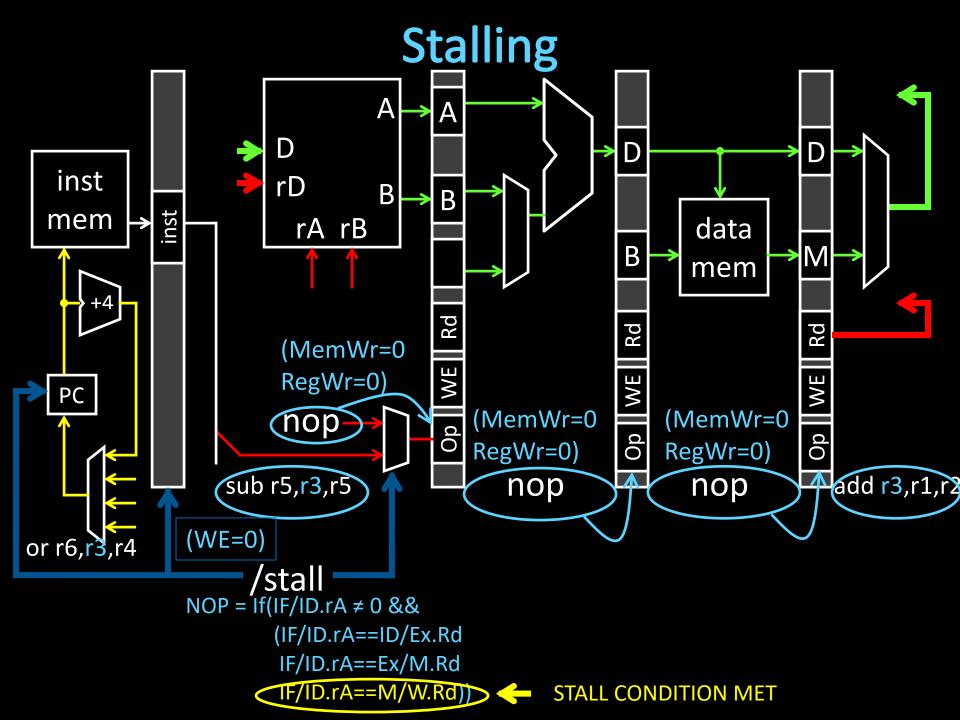
- prevent IF/ID pipeline register update
  - stalls the ID stage instruction
- convert ID stage instr into nop for later stages
  - innocuous "bubble" passes through pipeline
- prevent PC update
  - stalls the next (IF stage) instruction



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										
	,									









time	Clock cycl				
<del></del>	1	2			

3

6

8

r3 = 10add r3, r1,

r3 = 20

sub r5, r3,

or r6, r3,

add r6, r3,

, r2					
, r5					
, r4					
, r8					

#### How to stall an instruction in ID stage

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

### **Takeaway**

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs 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.

\*Bubbles in pipeline significantly decrease performance.

### 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)

### **Forwarding**

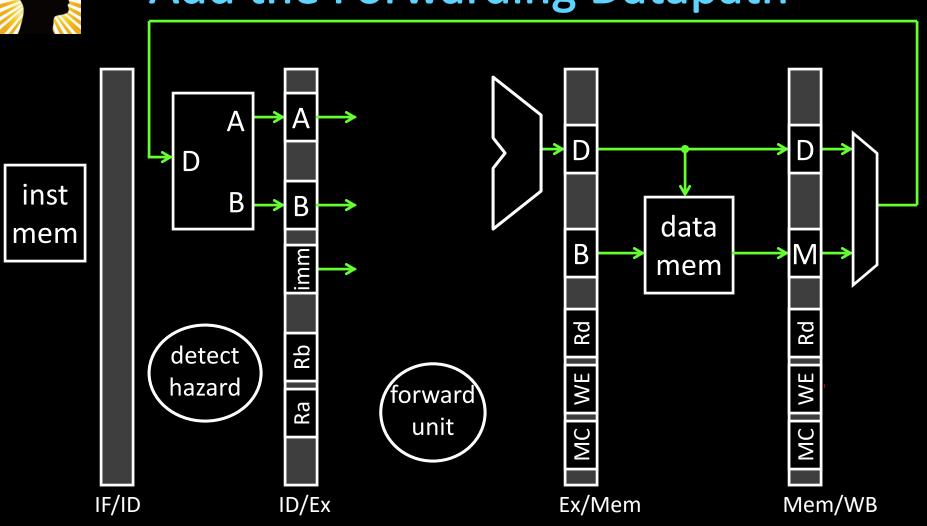
Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).

### Three types of forwarding/bypass

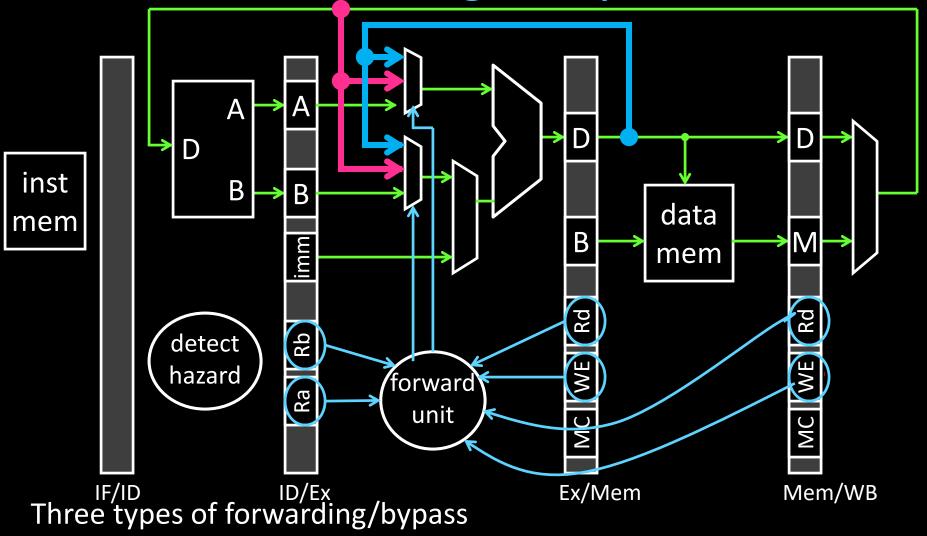
- Forwarding from Ex/Mem registers to Ex stage  $(M\rightarrow Ex)$
- Forwarding from Mem/WB register to Ex stage ( $W\rightarrow Ex$ )
- RegisterFile Bypass



## Add the Forwarding Datapath

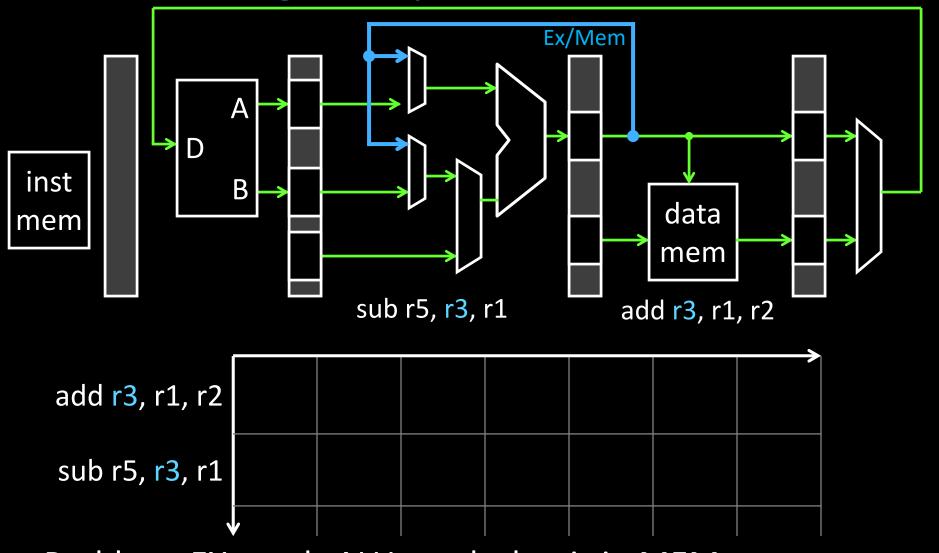


## **Forwarding Datapath**



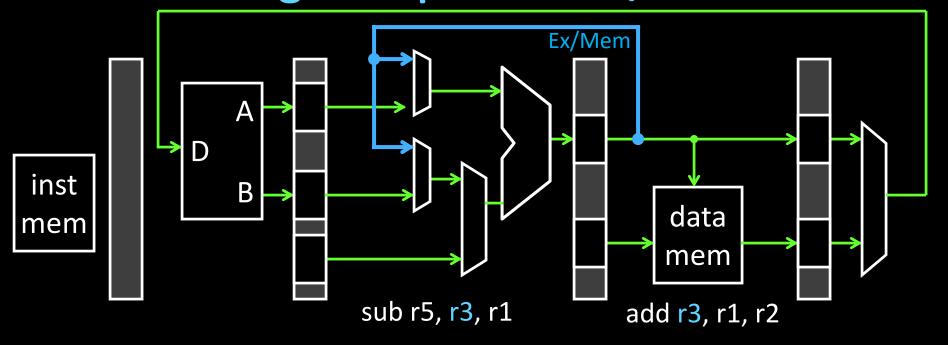
- Forwarding from Ex/Mem registers to Ex stage  $(M \rightarrow Ex)$
- Forwarding from Mem/WB register to Ex stage (W  $\rightarrow$  Ex)
- RegisterFile Bypass

# 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

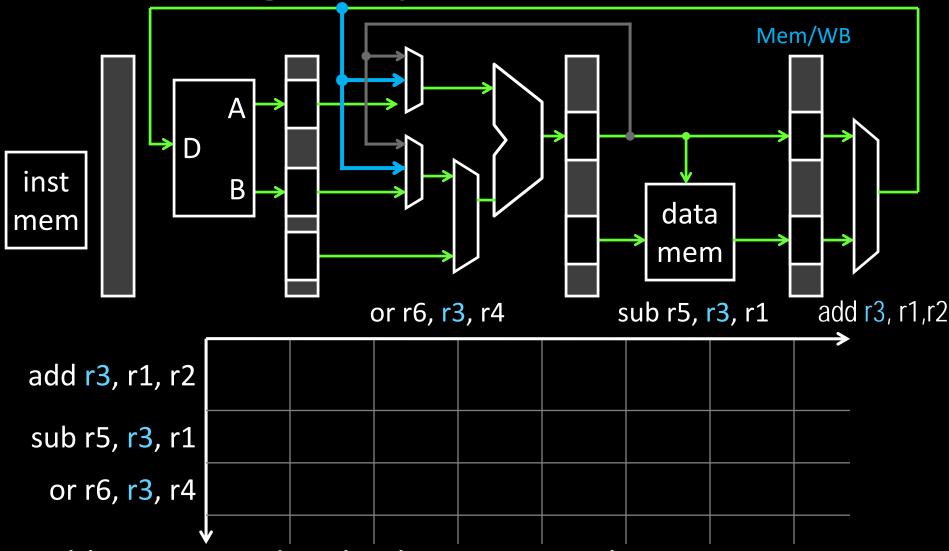
# Forwarding Datapath 1: Ex/MEM -> EX



#### Detection Logic in Ex Stage:

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

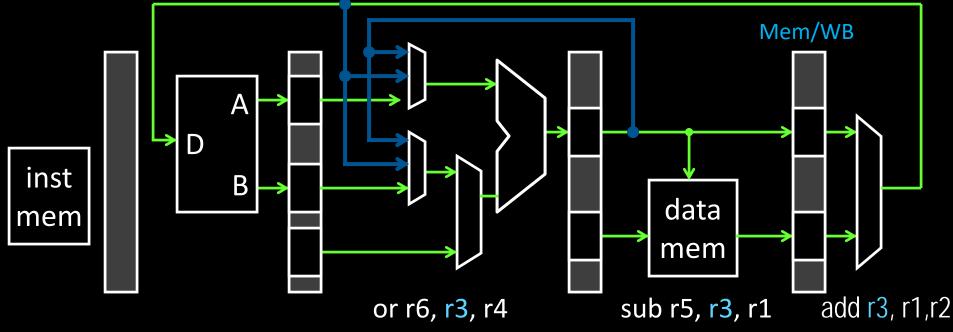
# Forwarding Datapath 2: Mem/WB -> EX



Problem: EX needs value being written by WB

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

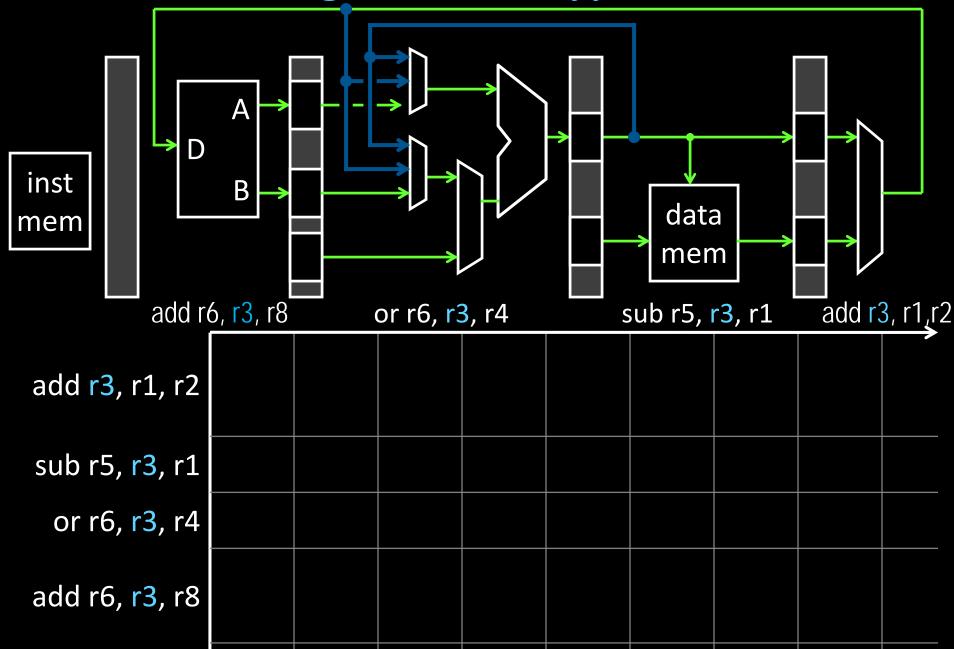
## Forwarding Datapath 2: Mem/WB -> EX



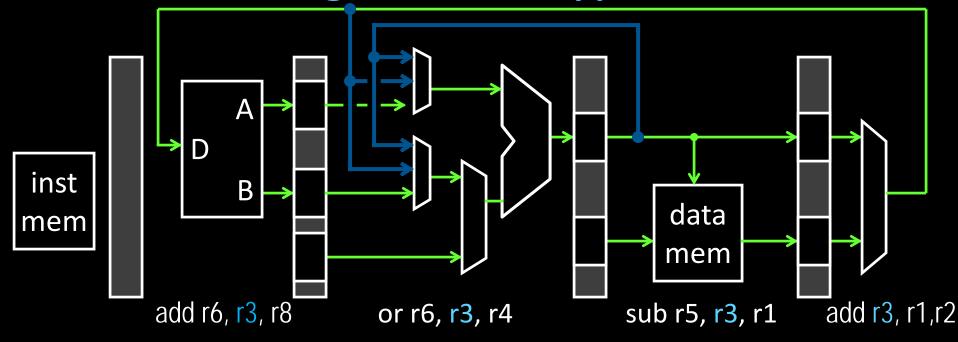
#### **Detection Logic:**

```
forward = (M/WB.WE && M/WB.Rd != 0 &&
ID/Ex.Ra == M/WB.Rd &&
not (ID/Ex.WE && Ex/M.Rd != 0 &&
ID/Ex.Ra == Ex/M.Rd)
[] (same for Rb)
```

# Register File Bypass



## Register File Bypass



Problem: Reading a value that is currently being written

Solution: just negate register file clock

- writes happen at end of first half of each clock cycle
- reads happen during second half of each clock cycle

# Forwarding Example 2



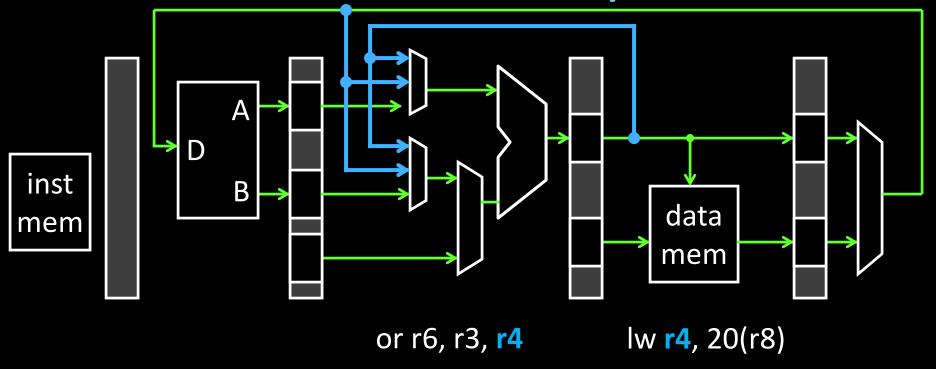
75

time	Clock cycle								
•••••	1	2	3	4	5	6	7	8	
add r3, r1, r2									
sub r5, r3, r4									
lw r6, 4(r3)									
or r5, r3, r5									
sw r6, 12(r3)									

# Forwarding Example 2

time	Clock cycle								
	1	2	3	4	5	6	7	8	<b>&gt;</b>
add r3, r1, r2	IF	ID	Ex	M	W				
sub r5, r3, r4		IF	ID	Ex	M	W			
[lw r6, 4(r3)									
or r5, r3, r5									
sw r6, 12(r3)									

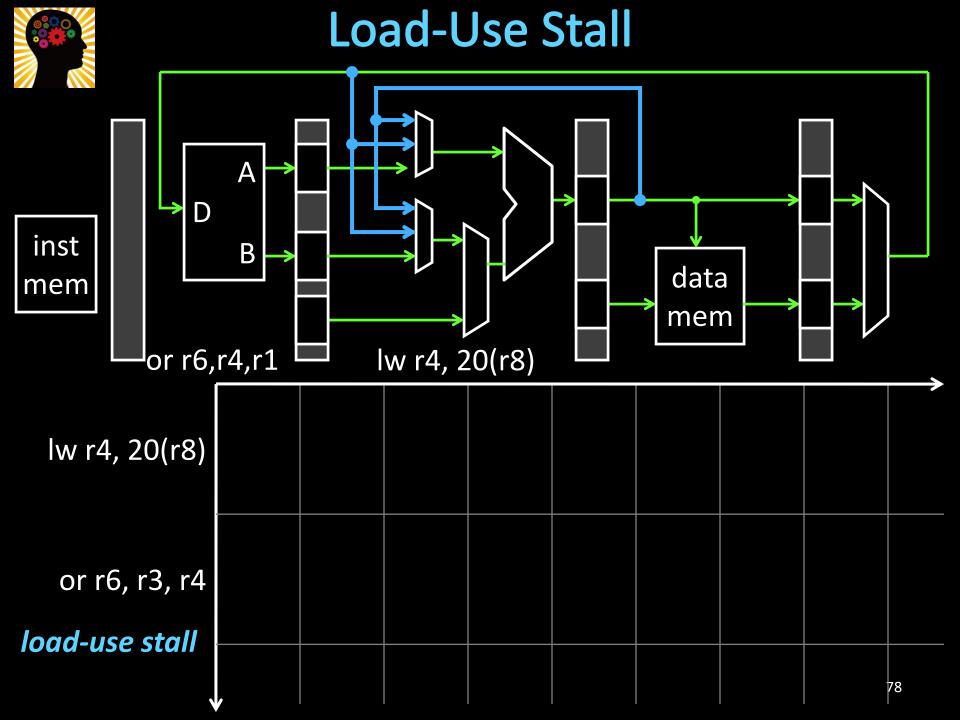
## Load-Use Hazard Explained



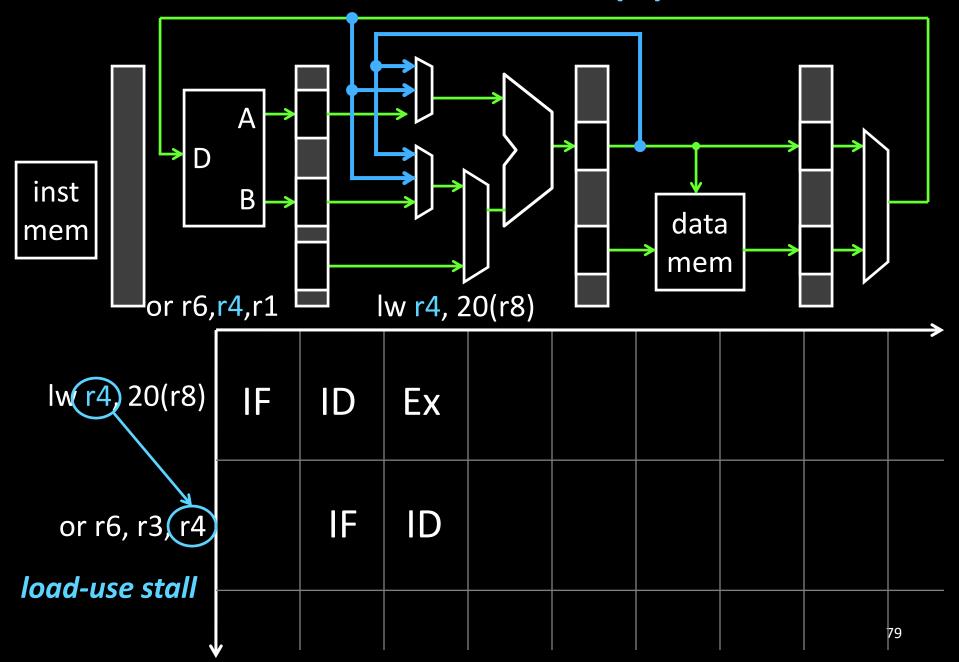
#### Data dependency after a load instruction:

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

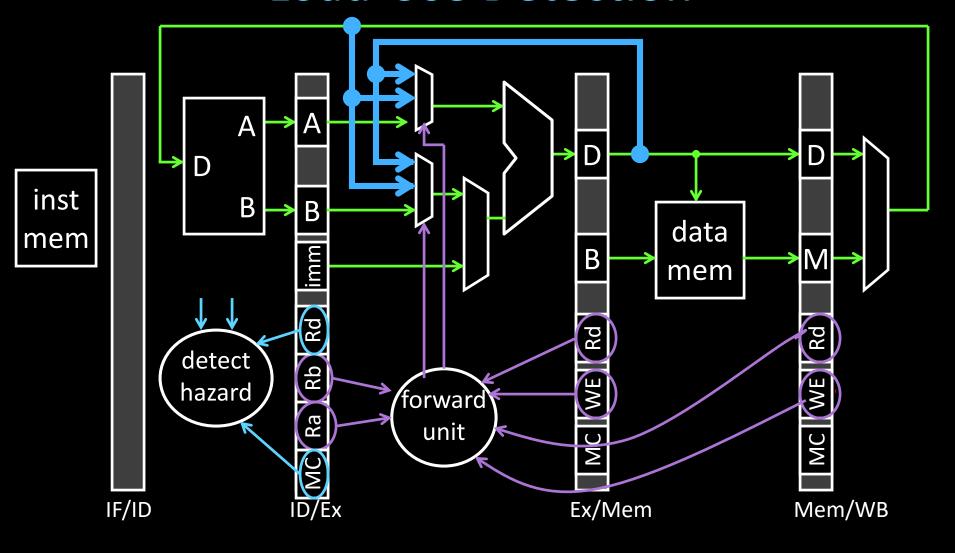
THE KILLER HAZARD



## Load-Use Stall (1)



## Load-Use Detection



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

## 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

## **Takeaway**

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs 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. Bubbles (nops) in pipeline significantly decrease performance.

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

### Quiz

Find all hazards, and say how they are resolved:

```
add r3, r1, r2
nand r5, r3, r4
add r2, r6, r3
lw r6, 24(r3)
sw r6, 12(r2)
```

## Data Hazard Recap

### Delay Slot(s)

Modify ISA to match implementation

#### Stall

Pause current and all subsequent instructions

### Forward/Bypass

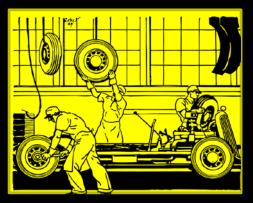
- Try to steal correct value from elsewhere in pipeline
- Otherwise, fall back to stalling or require a delay slot

#### Tradeoffs?

# Agenda

### 5-stage Pipeline

- Implementation
- Working Example







#### Hazards

- Structural
- Data Hazards
- Control Hazards

```
i = 0;
                A bit of Context
do {
   n += 2;
                                           i \rightarrow r1
  <u>i</u>++;
                                           Assume:
} while(i < max)</pre>
                                            n \rightarrow r2
i = 7;
                                            max \rightarrow r3
n--;
```

```
      x10
      addi r1, r0, 0
      # i=0

      x14 Loop: addi r2, r2, 2
      # n += 2

      x18
      addi r1, r1, 1
      # i++

      x1C
      blt r1, r3, Loop # i<max?</td>

      x20
      addi r1, r0, 7
      # i = 7

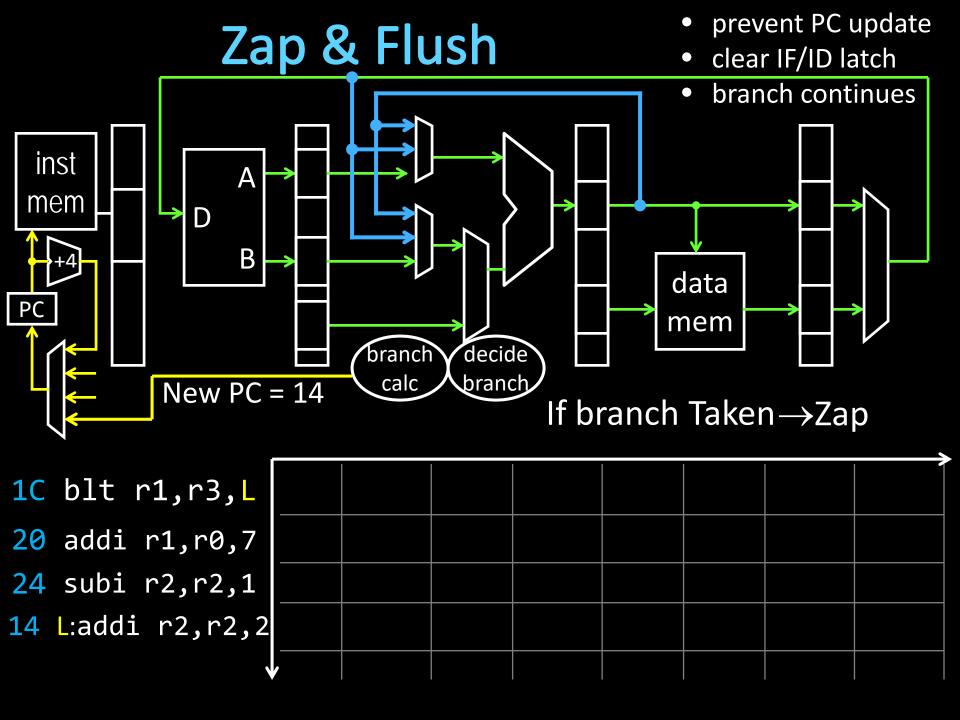
      x24
      subi r2, r2, 1
      # n--
```

### **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

```
x1C blt r1, r3, Loop
x20 addi r1, r0, 7
x24 subi r2, r2, 1 Branch taken?
Branch not taken?
```



## 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

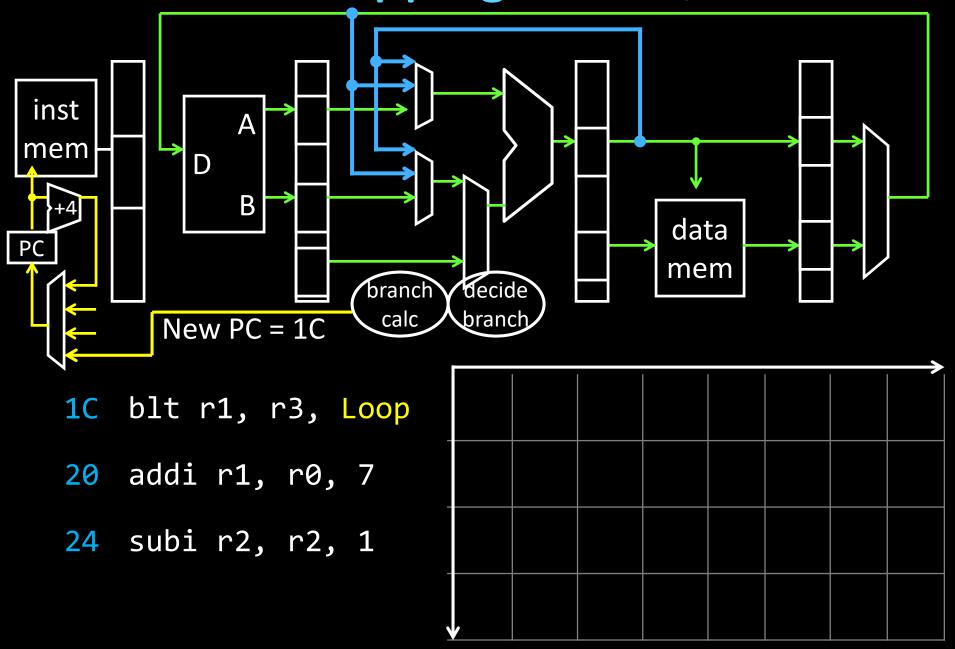
#### 2. Resolve Branch at Decode

- Some groups do this for Project 3, your choice
- Move branch calc from EX to ID
- Alternative: just zap 2<sup>nd</sup> instruction when branch taken

#### 3. Branch Prediction

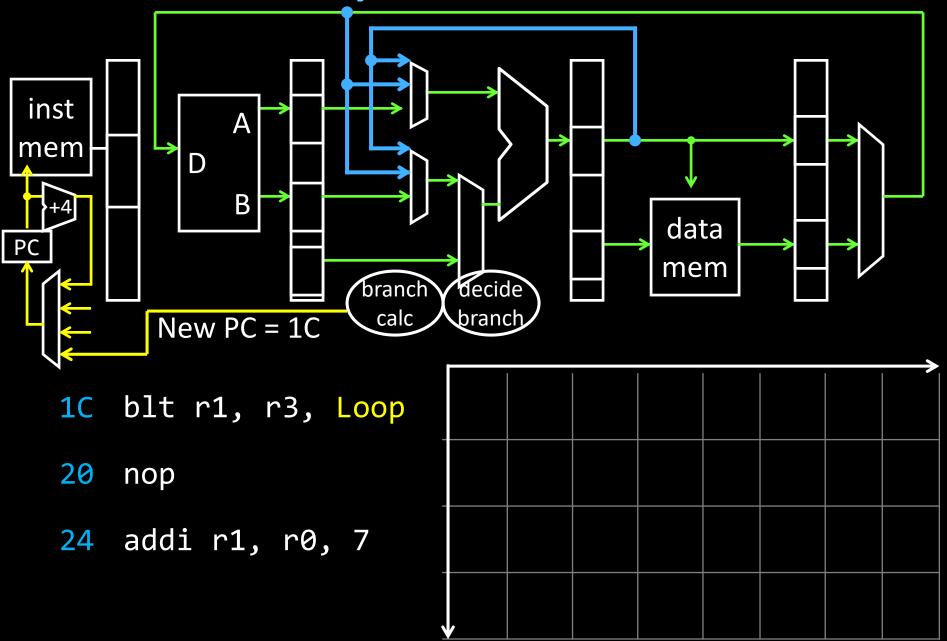
 Not in 3410, but every processor worth anything does this (no offense!)

# Problem: Zapping 2 insns/branch

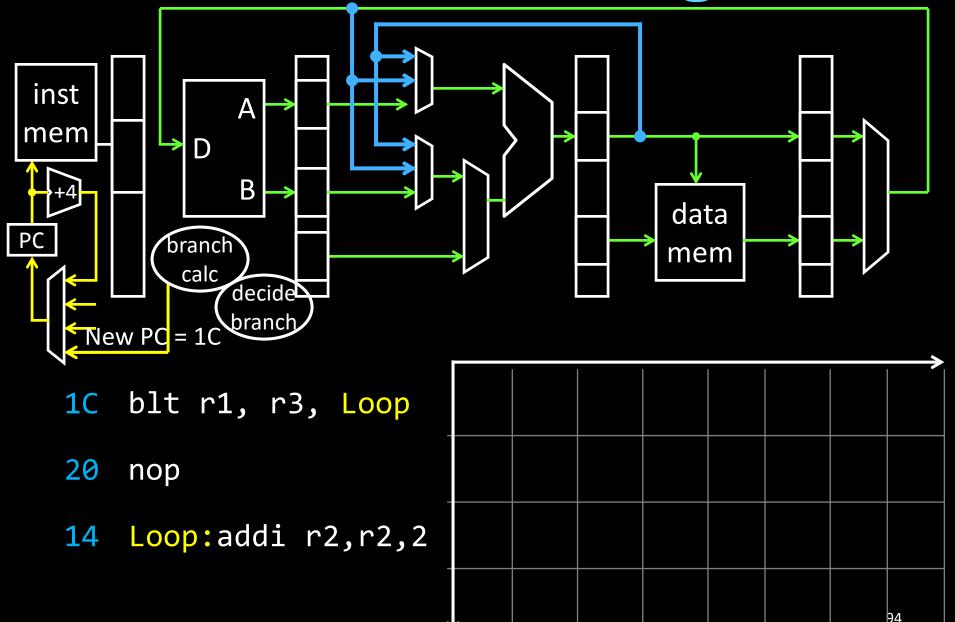


```
i = 0; Solution #1: Delay Slot
do {
  n += 2;
                                   i \rightarrow r1
  i++;
                                   Assume:
} while(i < max)</pre>
                                   n \rightarrow r2
i = 7;
                                   max \rightarrow r3
n--;
x10
            addi r1, r0, 0
                                  # i=0
x14 Loop: addi r2, r2, \overline{2} # n += 2
            addi r1, r1, 1
                                  # i++
x18
            blt r1, r3, Loop # i<max?
x1C
x20
            nop
                                  # i = 7
            addi r1, r0, 7
x24
            subi r2, r2, 1
                                  # n++
x28
```

# **Delay Slot in Action**



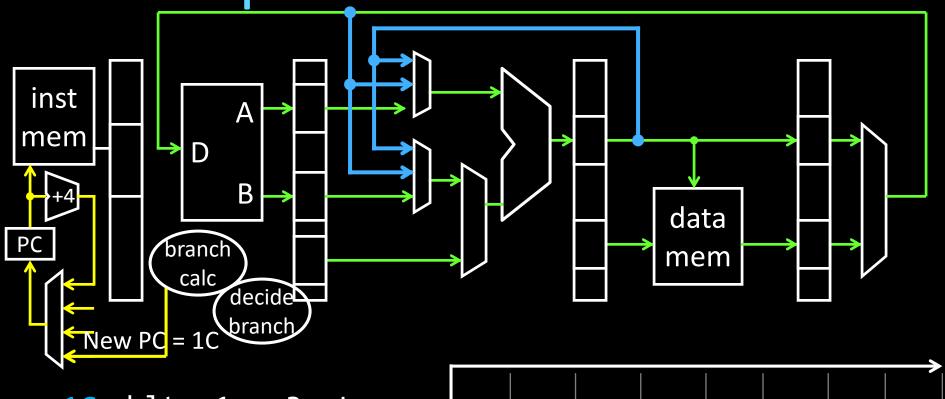
# Soln #2: Resolve Branches @ Decode



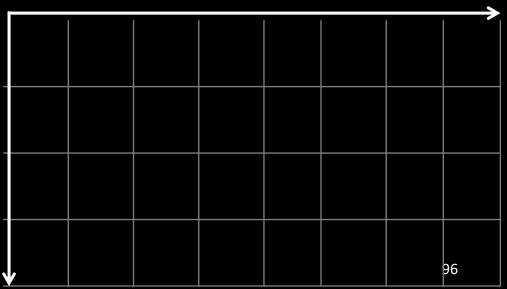
# Optimization: Fill the Delay Slot

```
x10
          addi r1, r0, 0
                             # i=0
   Loop: addi r2, r2, 2 # n += 2
x14
                          # <u>i++</u>
x18
          addi r1, r1, 1
          blt r1, r3, Loop # i<max?
x1C
x20
          nop
               Compiler transforms code
          addi r1, r0, 0
                            # i=0
x10
   Loop: addi r1, r1, 1
                             # i++
x14
          blt r1, r3, Loop # i<max?
x18
          addi r2, r2, 2
x1C
                             # n += 2
```

Optimization In Action!



- 1C blt r1, r3, Loop
- 20 addi r2,r2,2
- 14 Loop:addi r1,r1,1



### **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.