



# Chapter 4

## The Processor

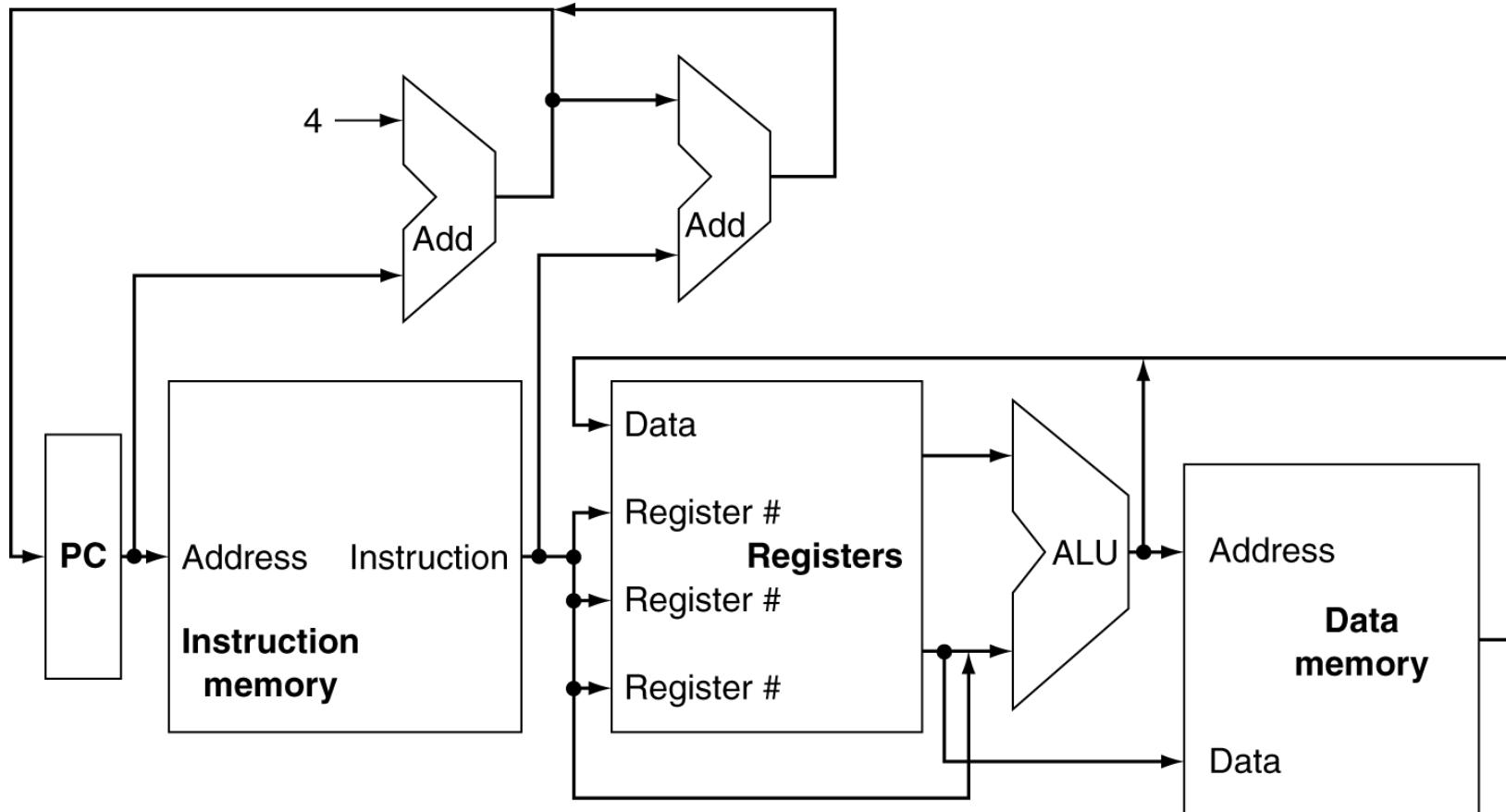
# Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: lw, sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j

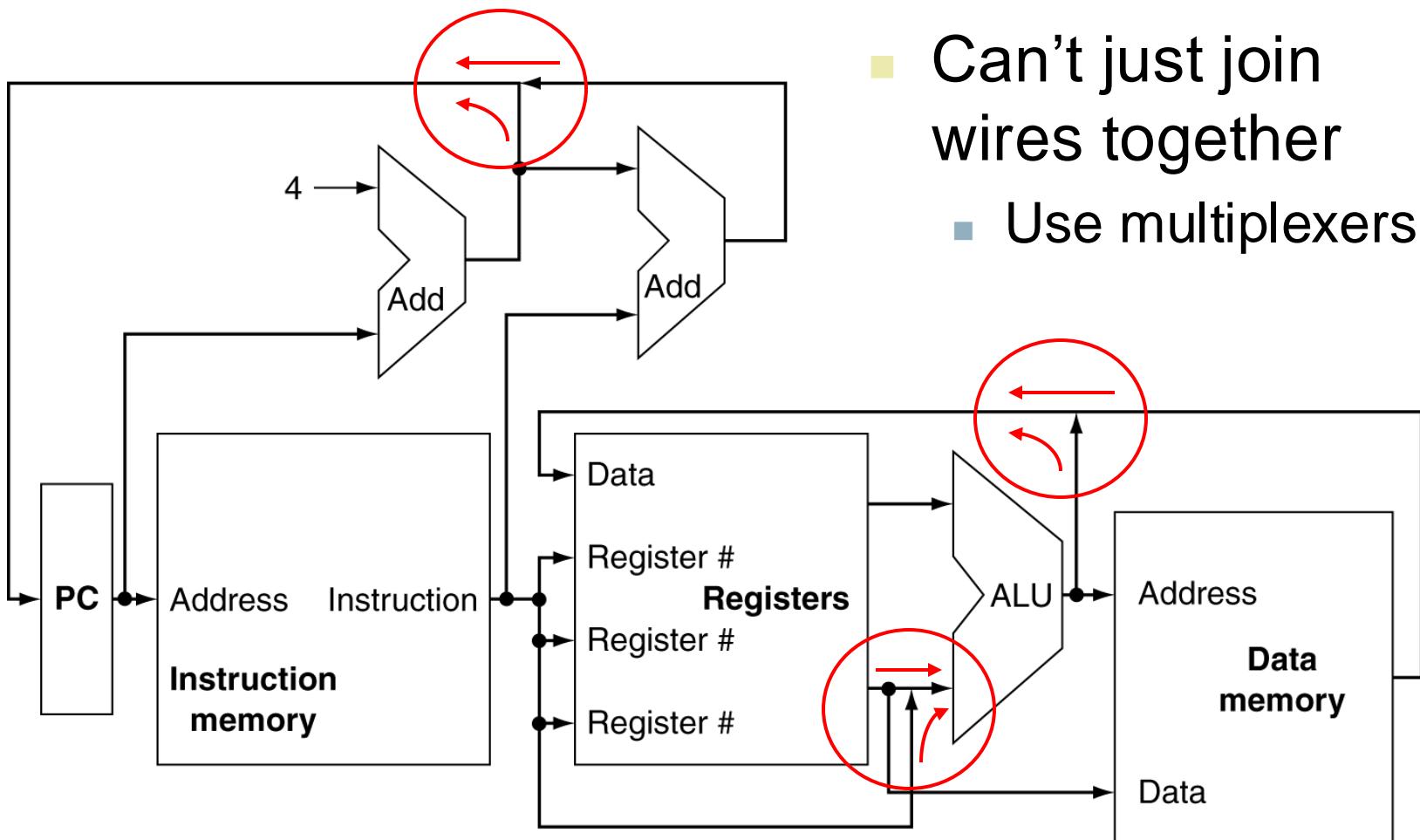
# Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

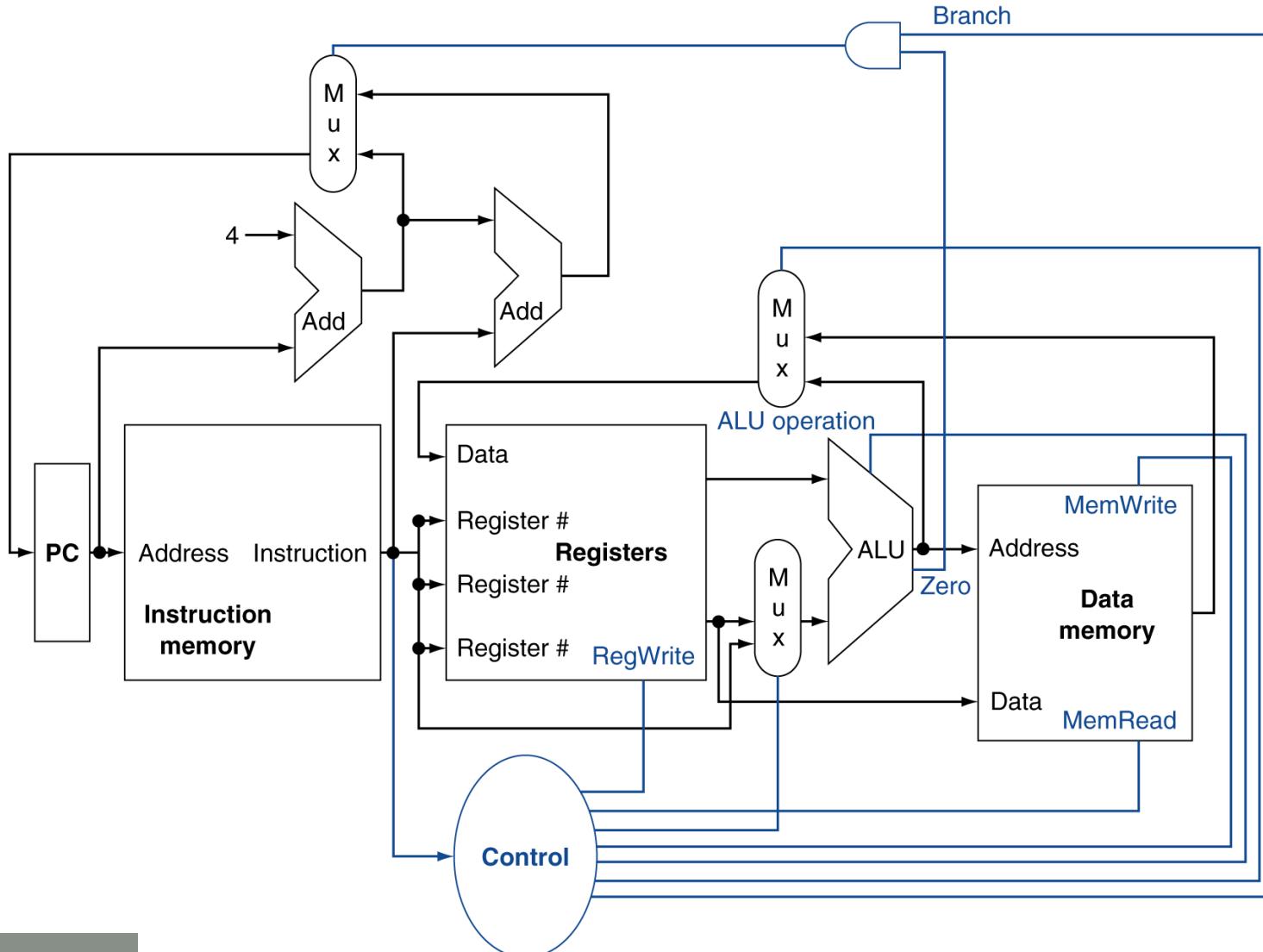
# CPU Overview



# Multiplexers



# Control

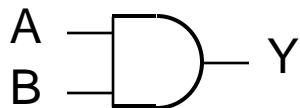


# Logic Design Basics

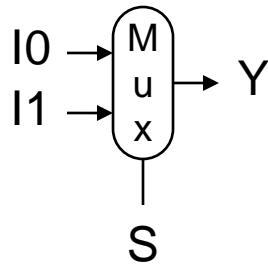
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

# Combinational Elements

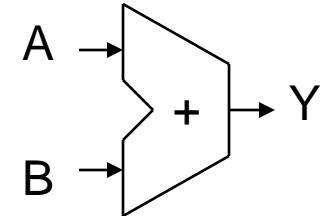
- AND-gate
  - $Y = A \& B$



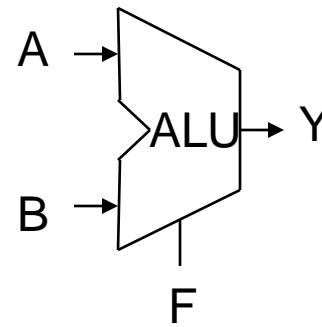
- Multiplexer
  - $Y = S ? I_1 : I_0$



- Adder
  - $Y = A + B$

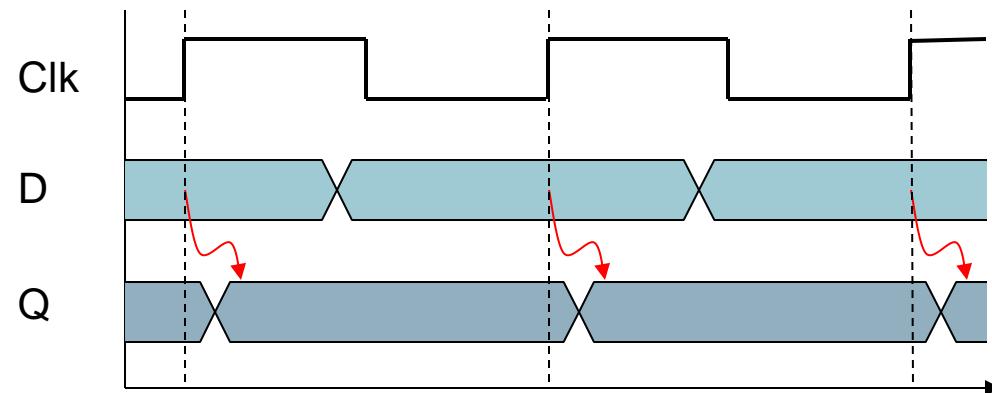
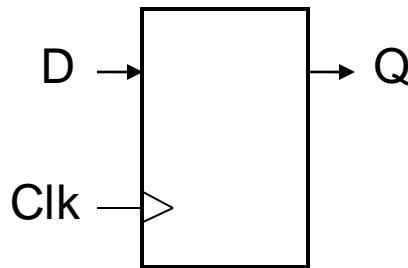


- Arithmetic/Logic Unit
  - $Y = F(A, B)$



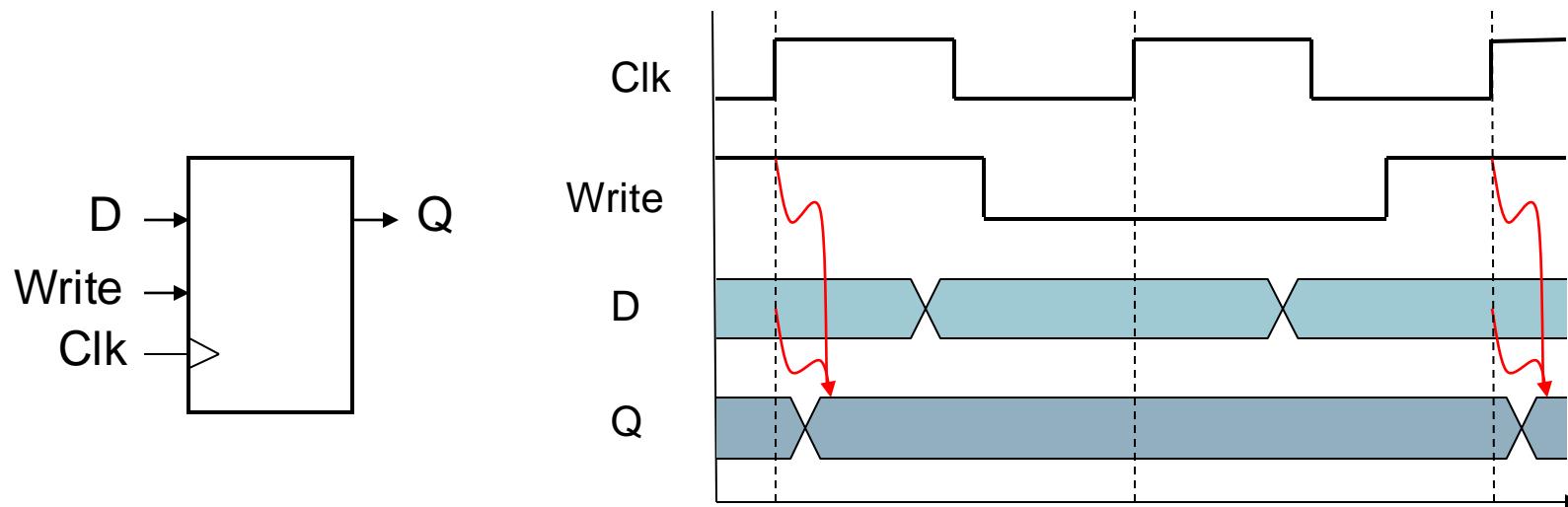
# Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1



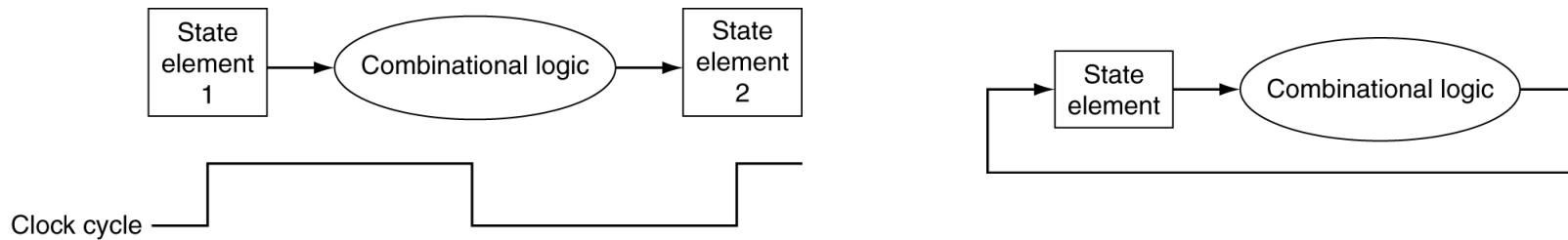
# Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



# Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

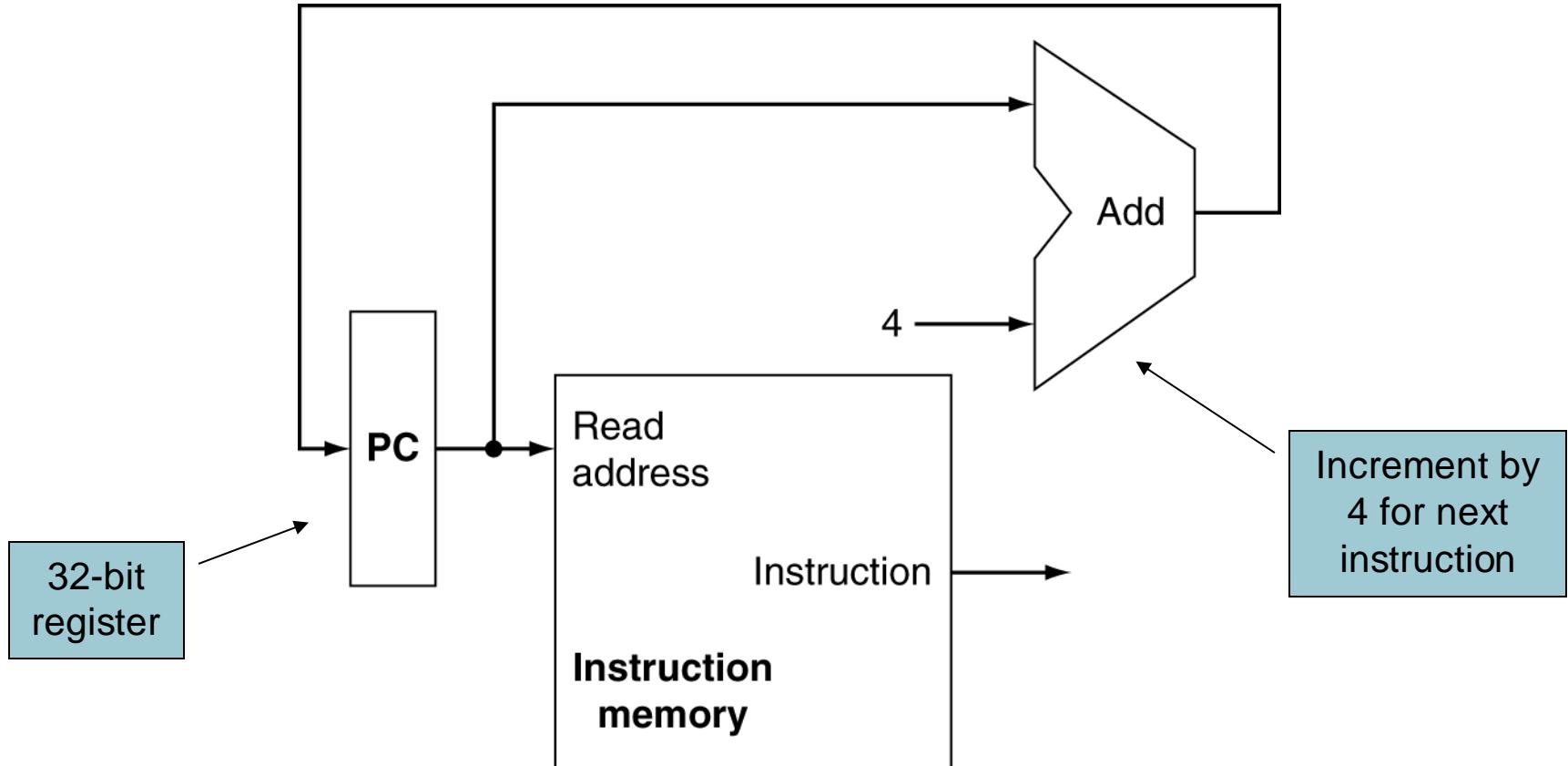


# Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
  - Refining the overview design

# Instruction Fetch

Common  
for all

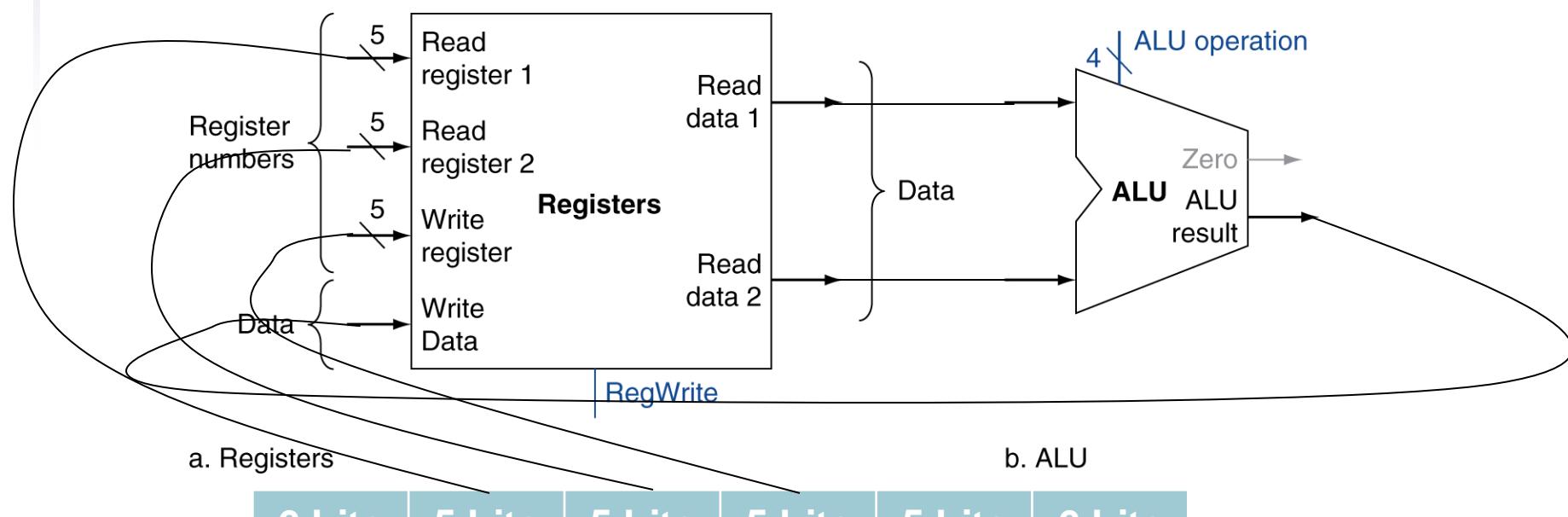


# Recall: Instruction Formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

# R-Format Instructions

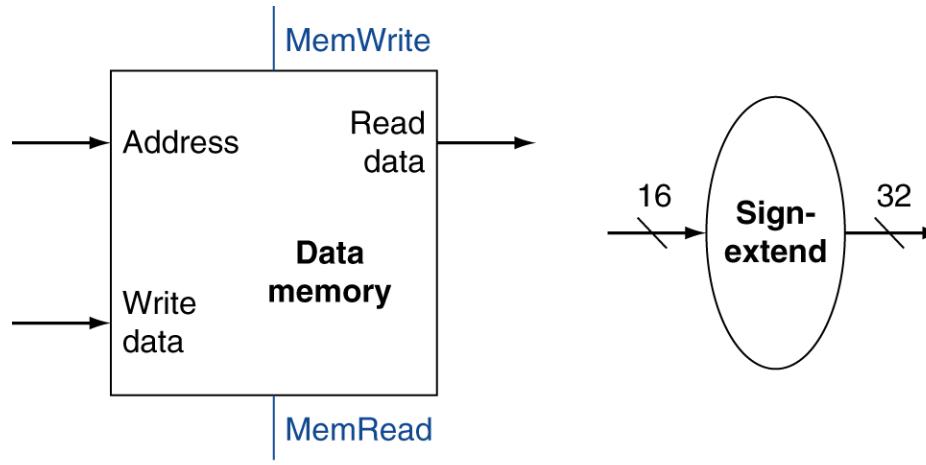
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

# Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

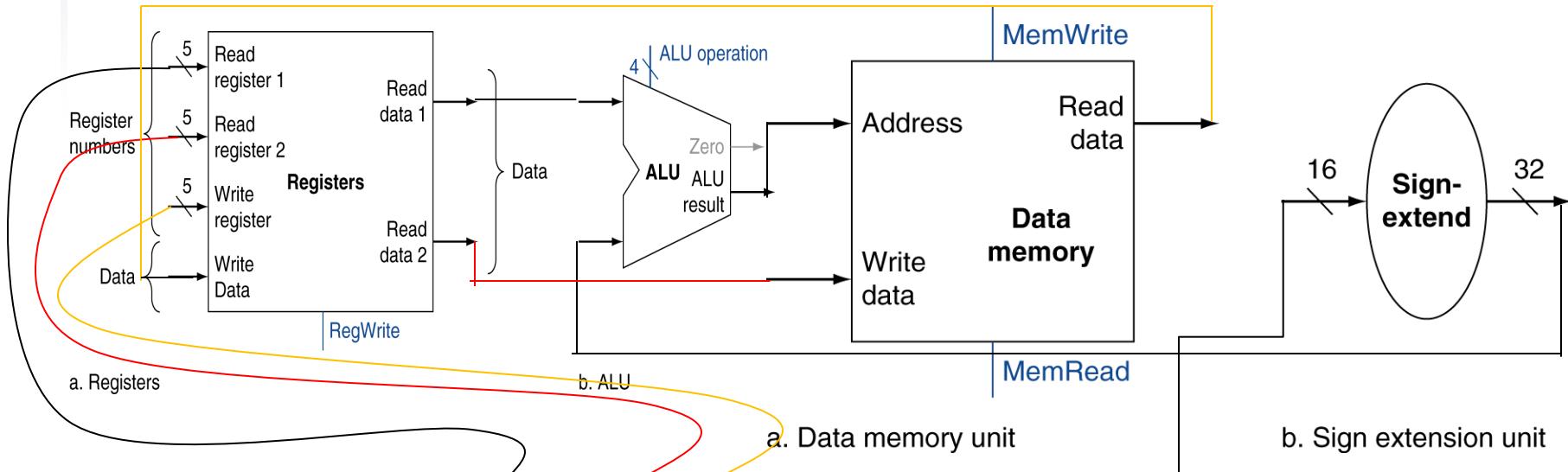
b. Sign extension unit

6 bits	5 bits	5 bits	5+5+6 = 16 bits
op	rs	rt	addr./imm.

# Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

— Store  
— Load

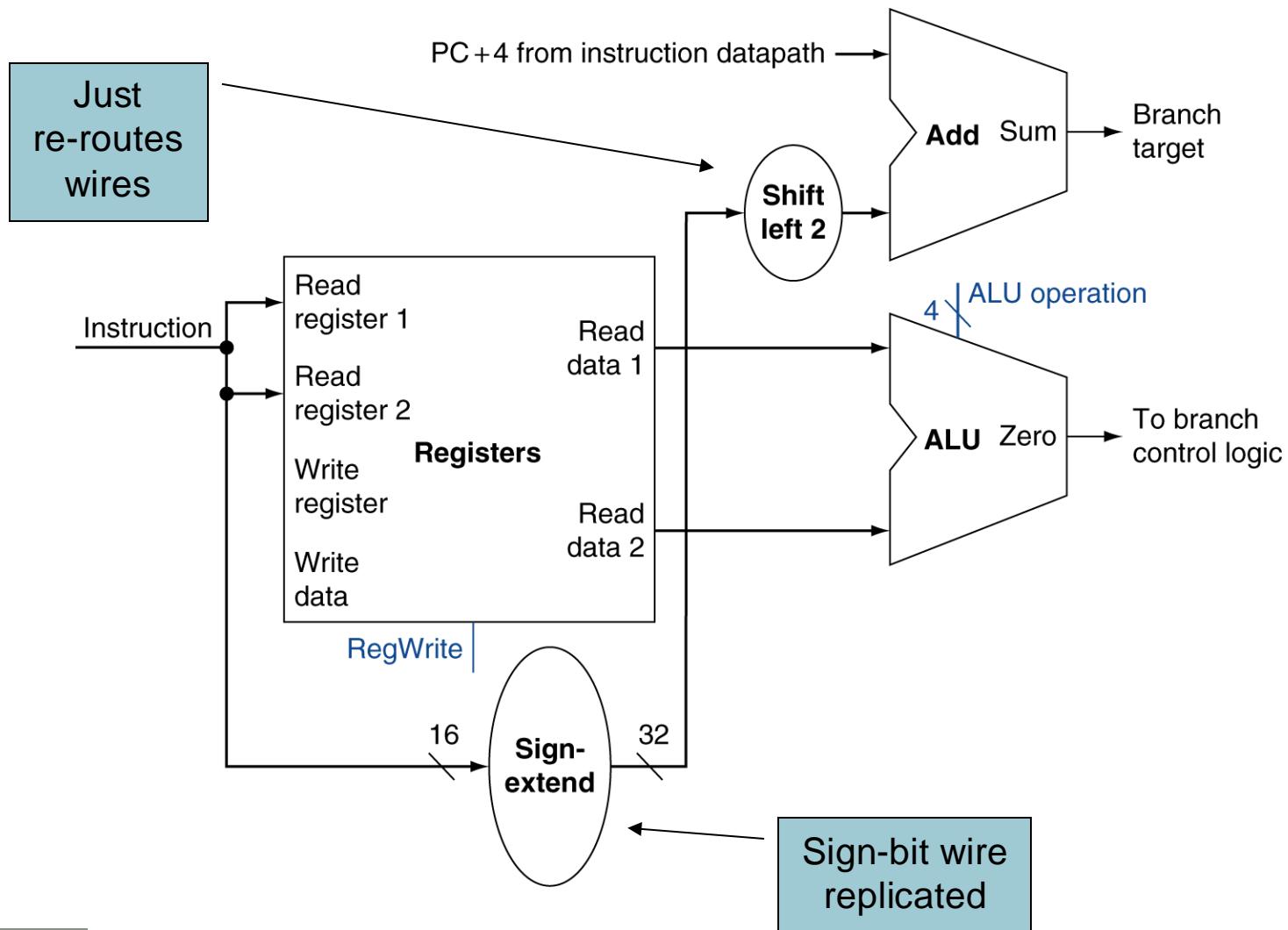


6 bits	5 bits	5 bits	5+5+6 = 16 bits
op	rs	rt	addr./imm.

# Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

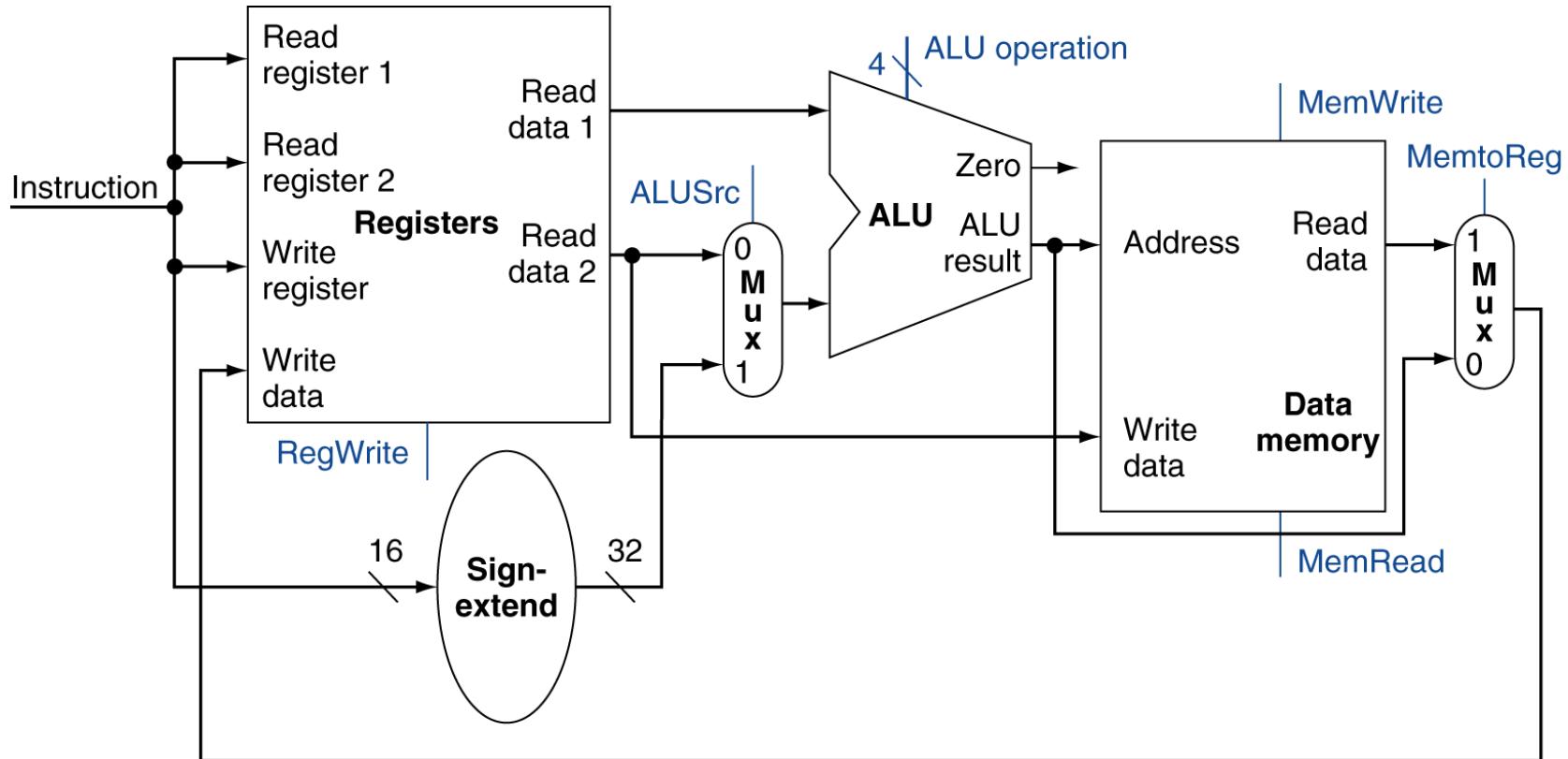
# Branch Instructions



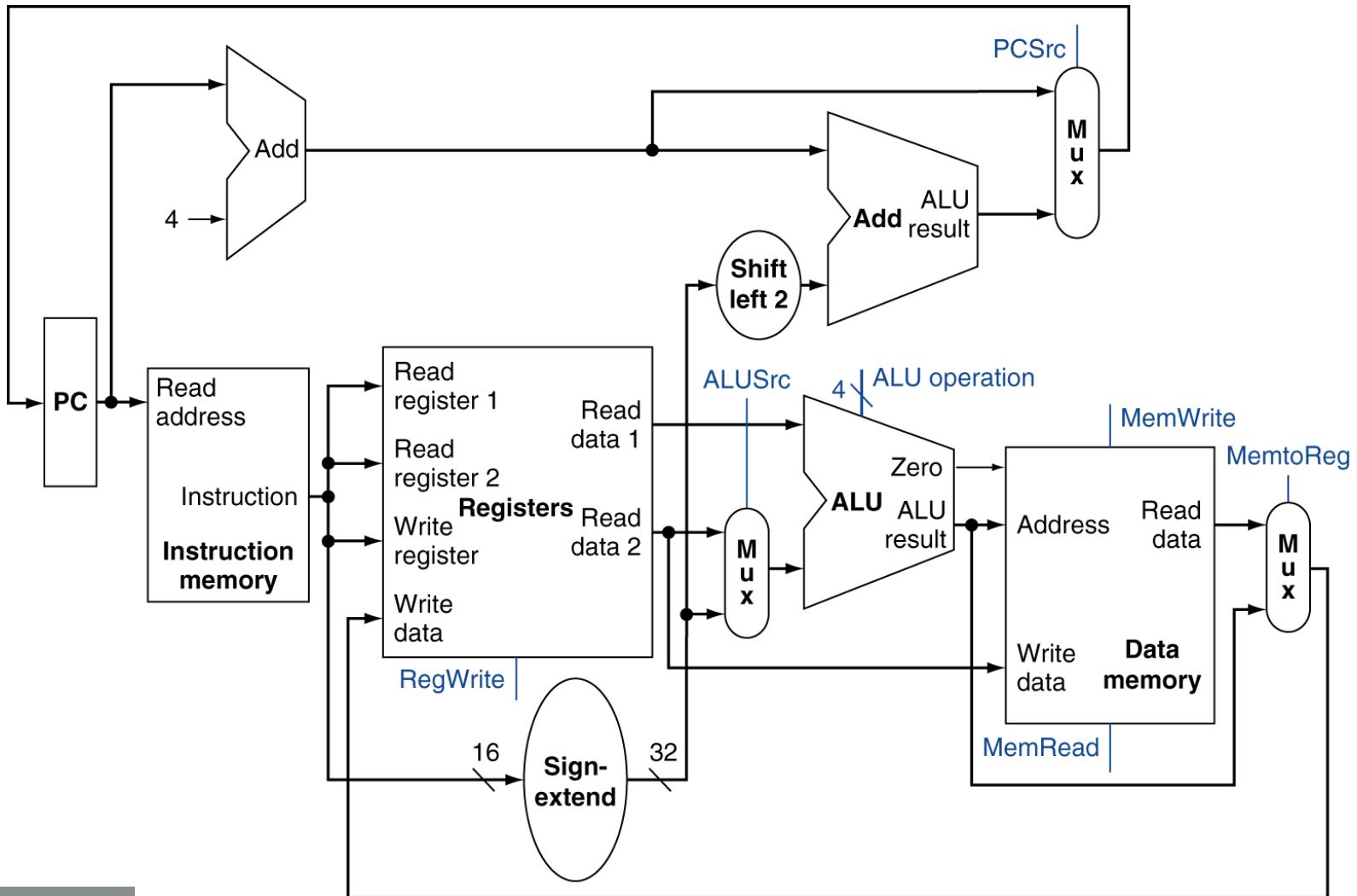
# Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

# R-Type/Load/Store Datapath



# Full Datapath



# ALU Control

- ALU used for
  - Load/Store:  $F = \text{add}$
  - Branch:  $F = \text{subtract}$
  - R-type:  $F$  depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

# ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

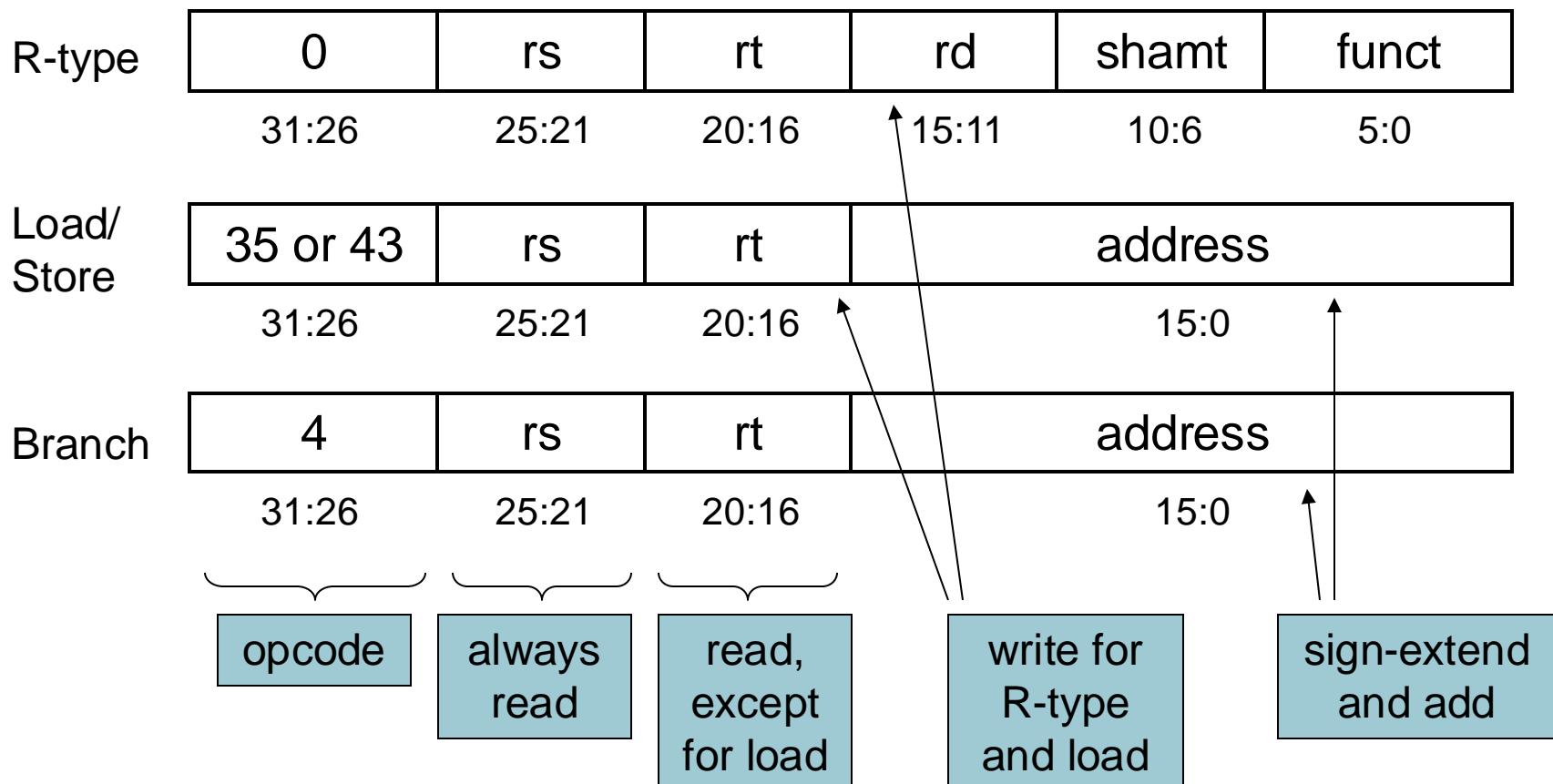
# ALU Control

## Truth Table For ALU Control Unit

ALUOp		Funct field							Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	0110	
1	X	X	X	0	0	0	0	0010	
1	X	X	X	0	0	1	0	0110	
1	X	X	X	0	1	0	0	0000	
1	X	X	X	0	1	0	1	0001	
1	X	X	X	1	0	1	0	0111	

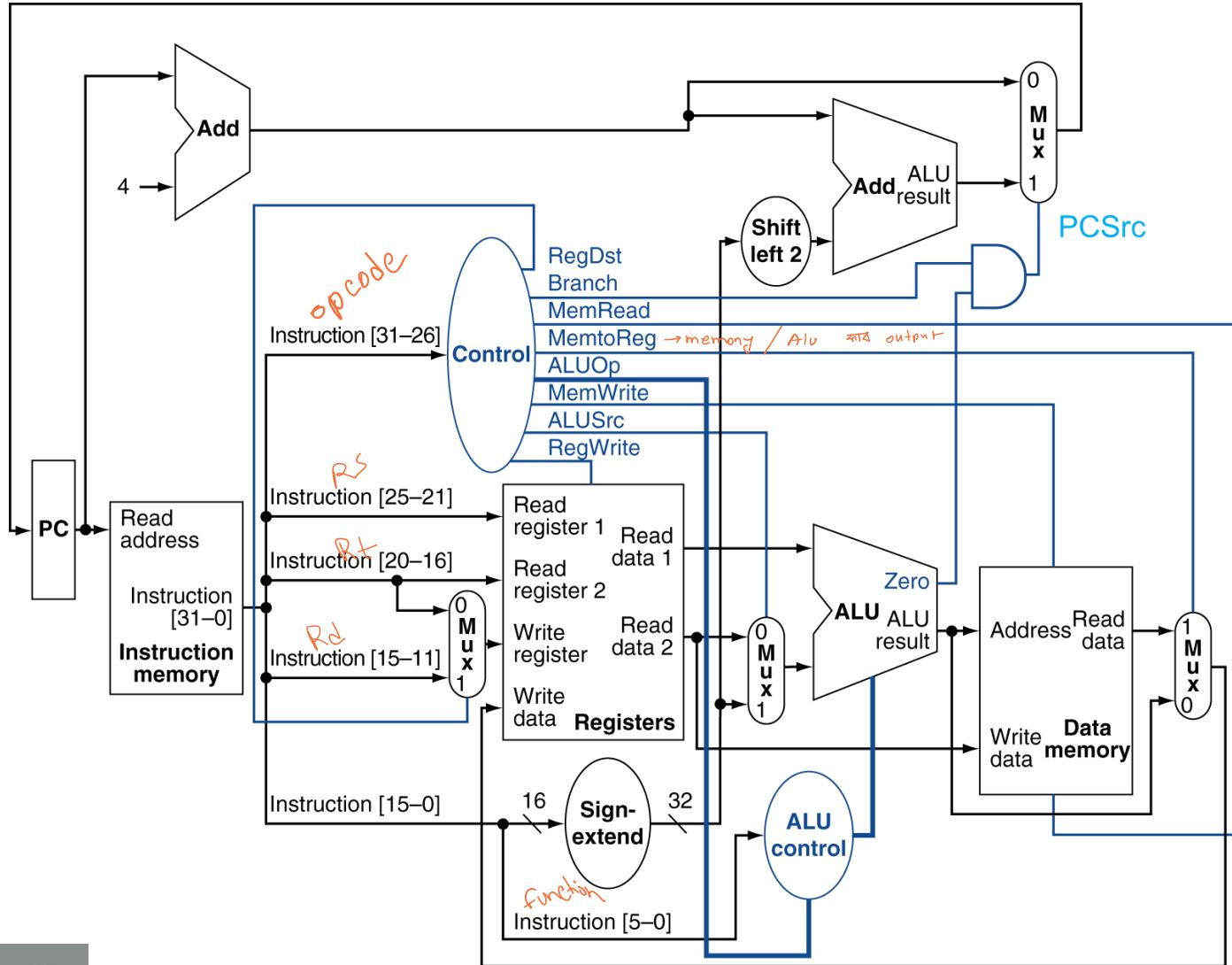
# The Main Control Unit

- Control signals derived from instruction



# Datapath With Control

Assertion - enable  
Deassertion → disable

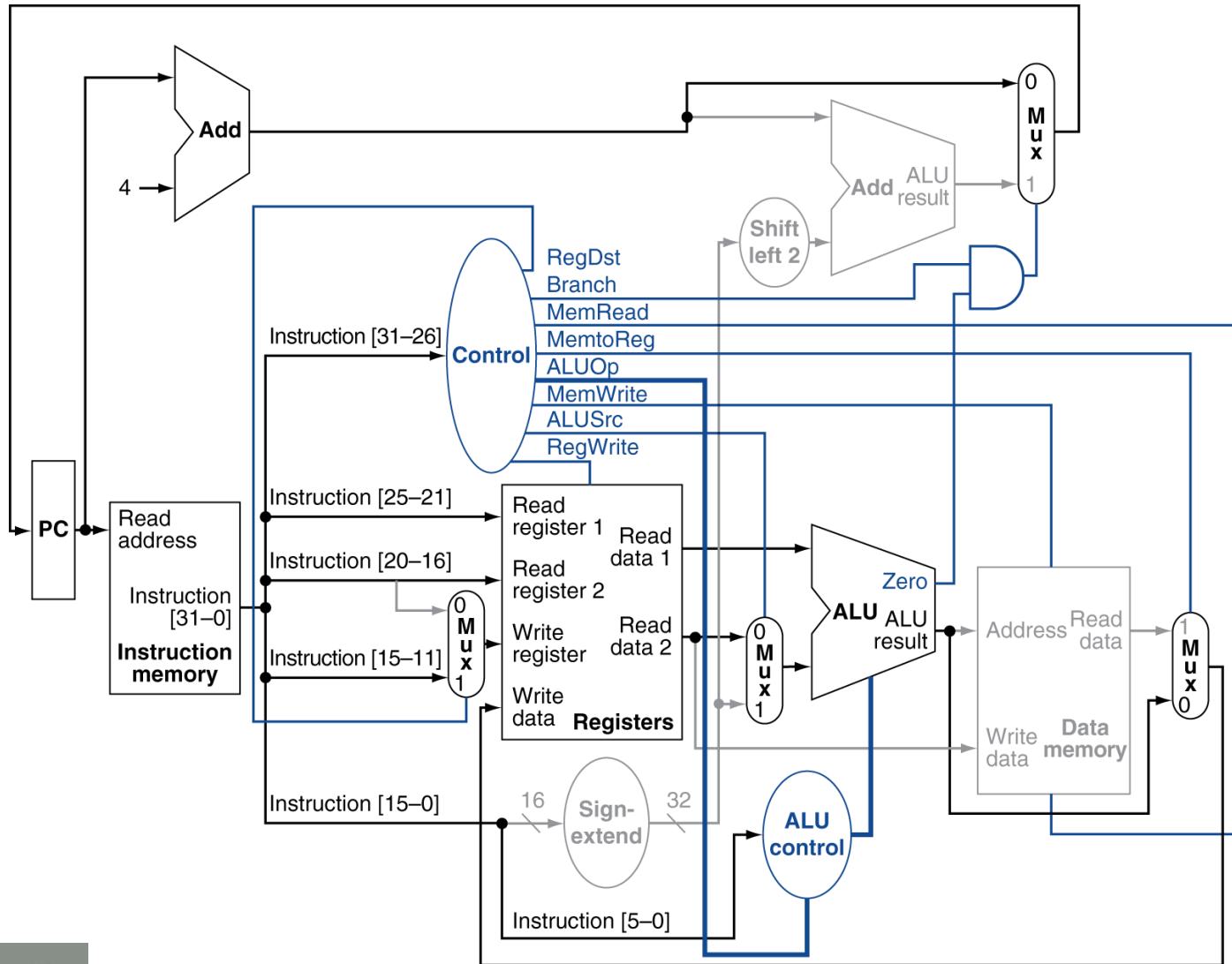


# All Control Signals

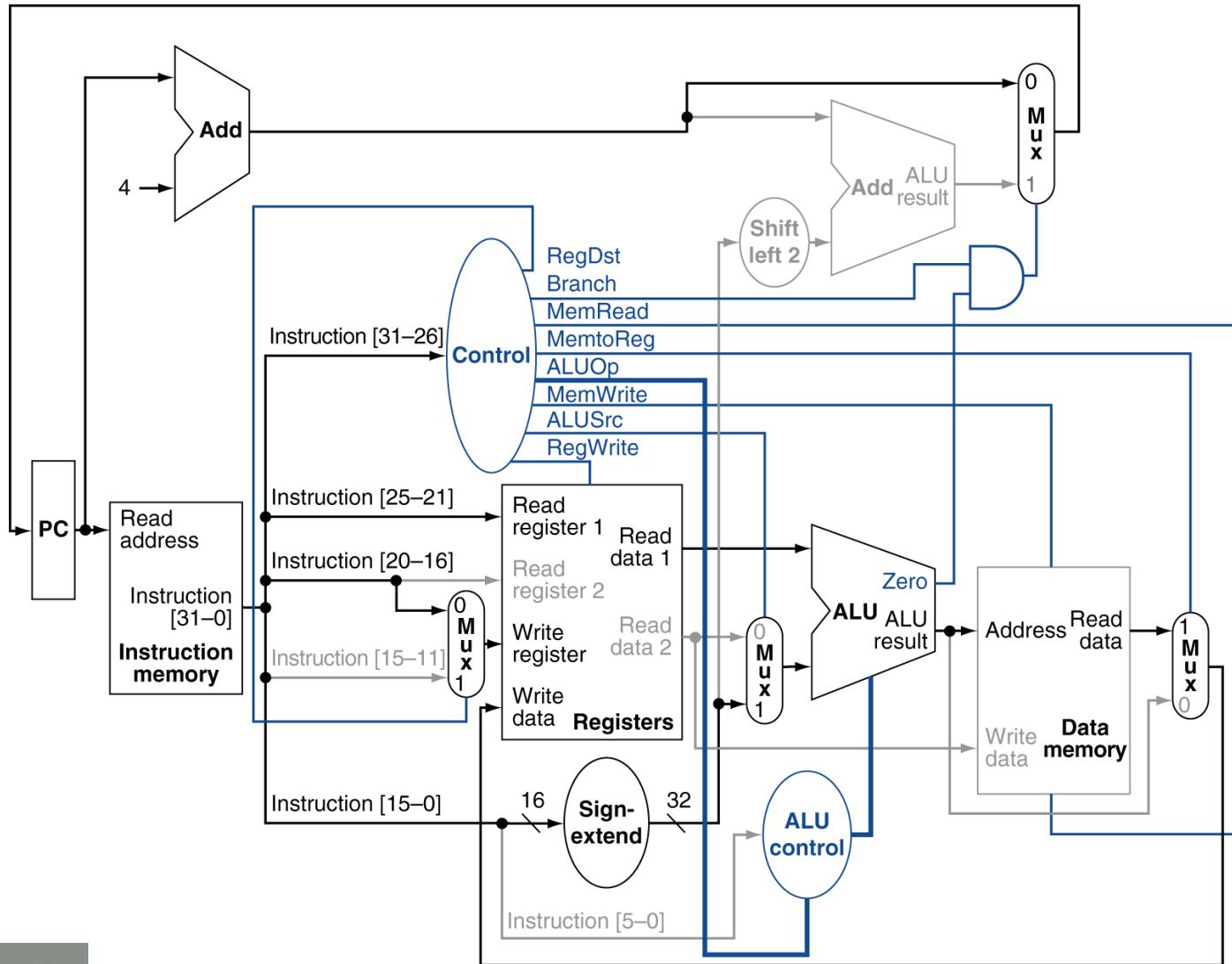
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

**FIGURE 4.16 The effect of each of the seven control signals.** When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See  Appendix B for further discussion of this problem.)

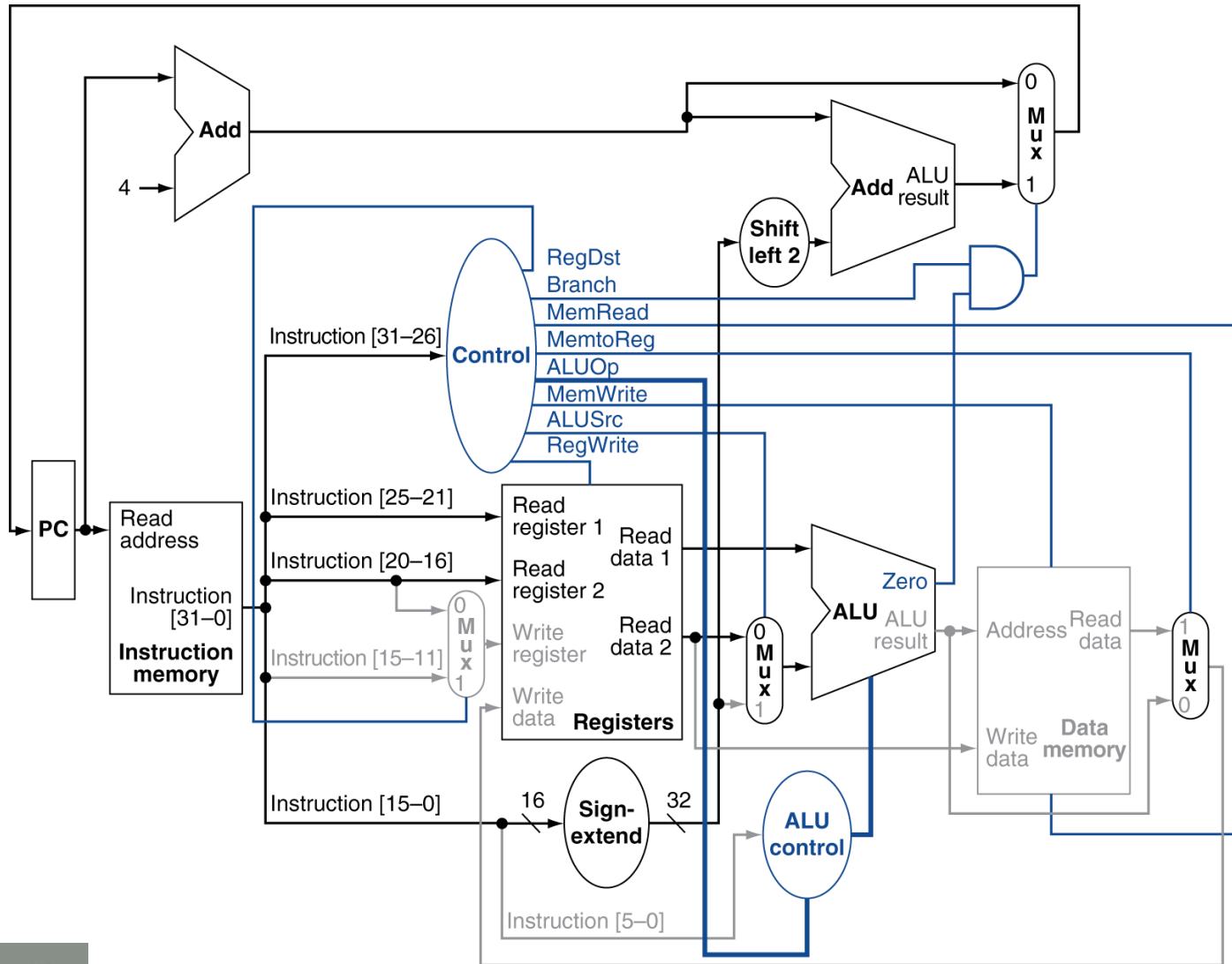
# R-Type Instruction



# Load Instruction



# Branch-on-Equal Instruction



# Control Values

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format									
lw		.		.		.			
sw									
beq									

# Control Values

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw		.		.		.			
sw									
beq									

# Control Values

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw									
beq									

# Control Values

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq									

# Control Values

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

load → critical path (height time  $\approx 7$ )

# Implementing Jumps

Jump

2

address

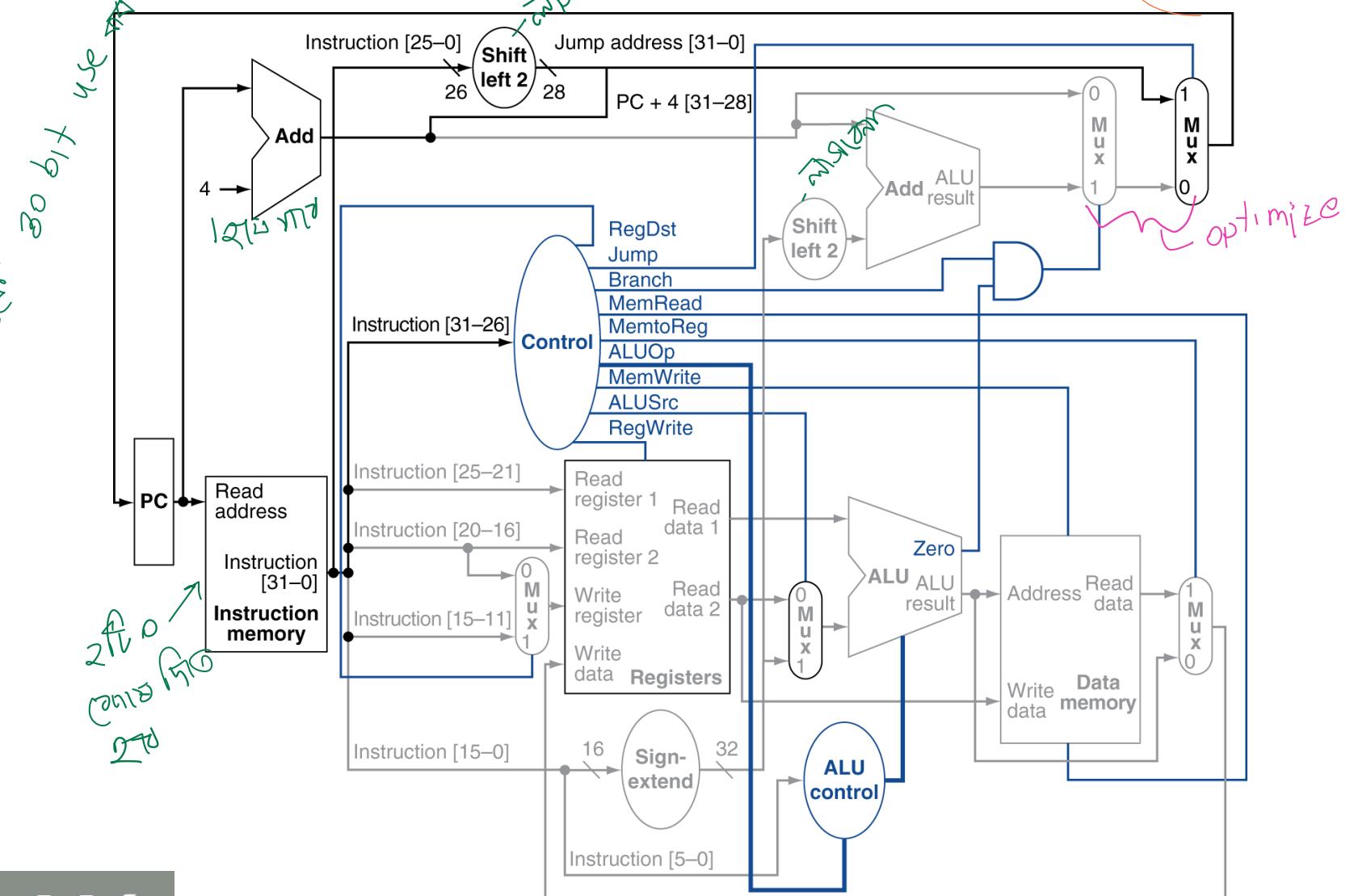
31:26

25:0

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

# Datapath With Jumps Added

(only black part)



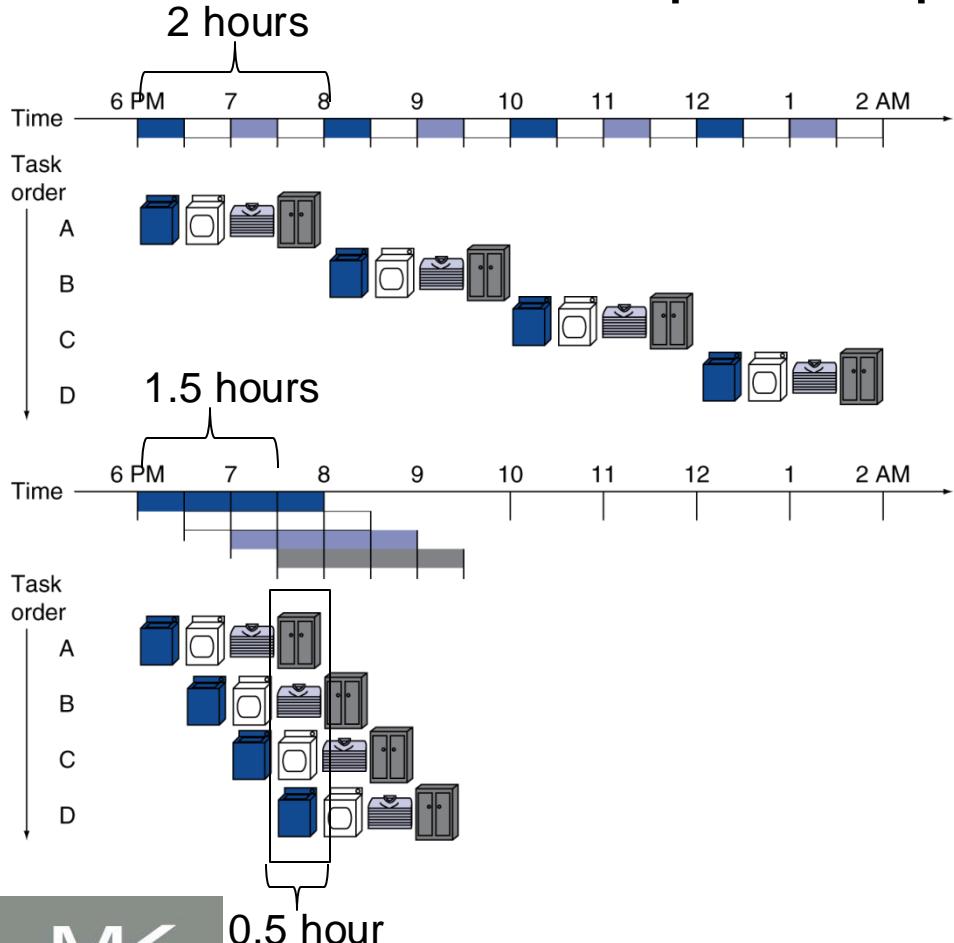
# Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

# Pipelining Analogy

throughput time optimize  
প্রক্ষেপণ

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup  
 $= 8/3.5 = 2.3$
- Non-stop:
  - Speedup  
 $= 2n/0.5n + 1.5 \approx 4$   
= number of stages

# MIPS Pipeline

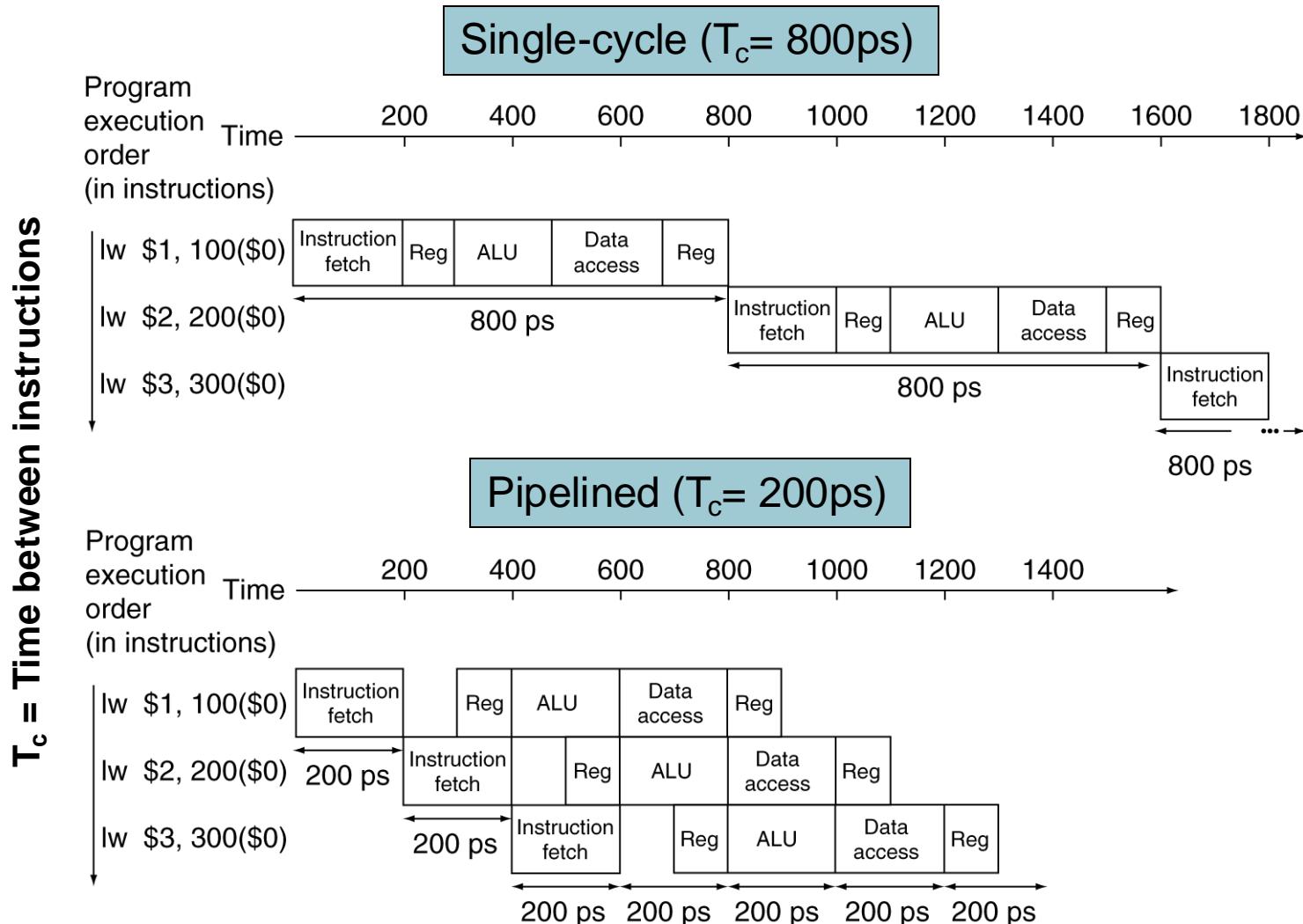
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

# Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

# Pipeline Performance



# Pipeline Speedup

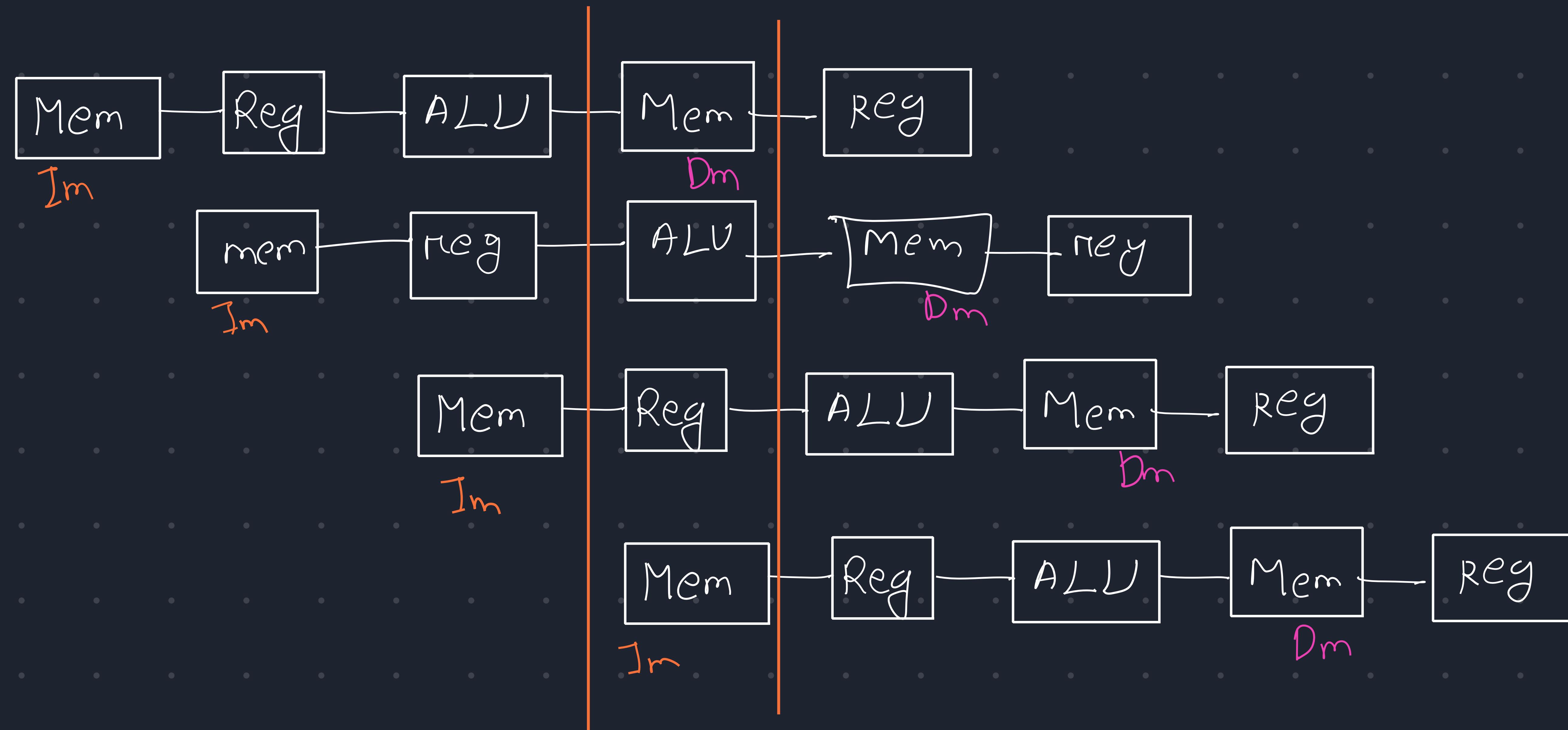
- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions<sub>pipelined</sub>  
= Time between instructions<sub>nonpipelined</sub>  
                                Number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

# Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to ~~15~~<sup>15</sup>-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3<sup>rd</sup> stage, access memory in 4<sup>th</sup> stage
  - Alignment of memory operands
    - Memory access takes only one cycle

# Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

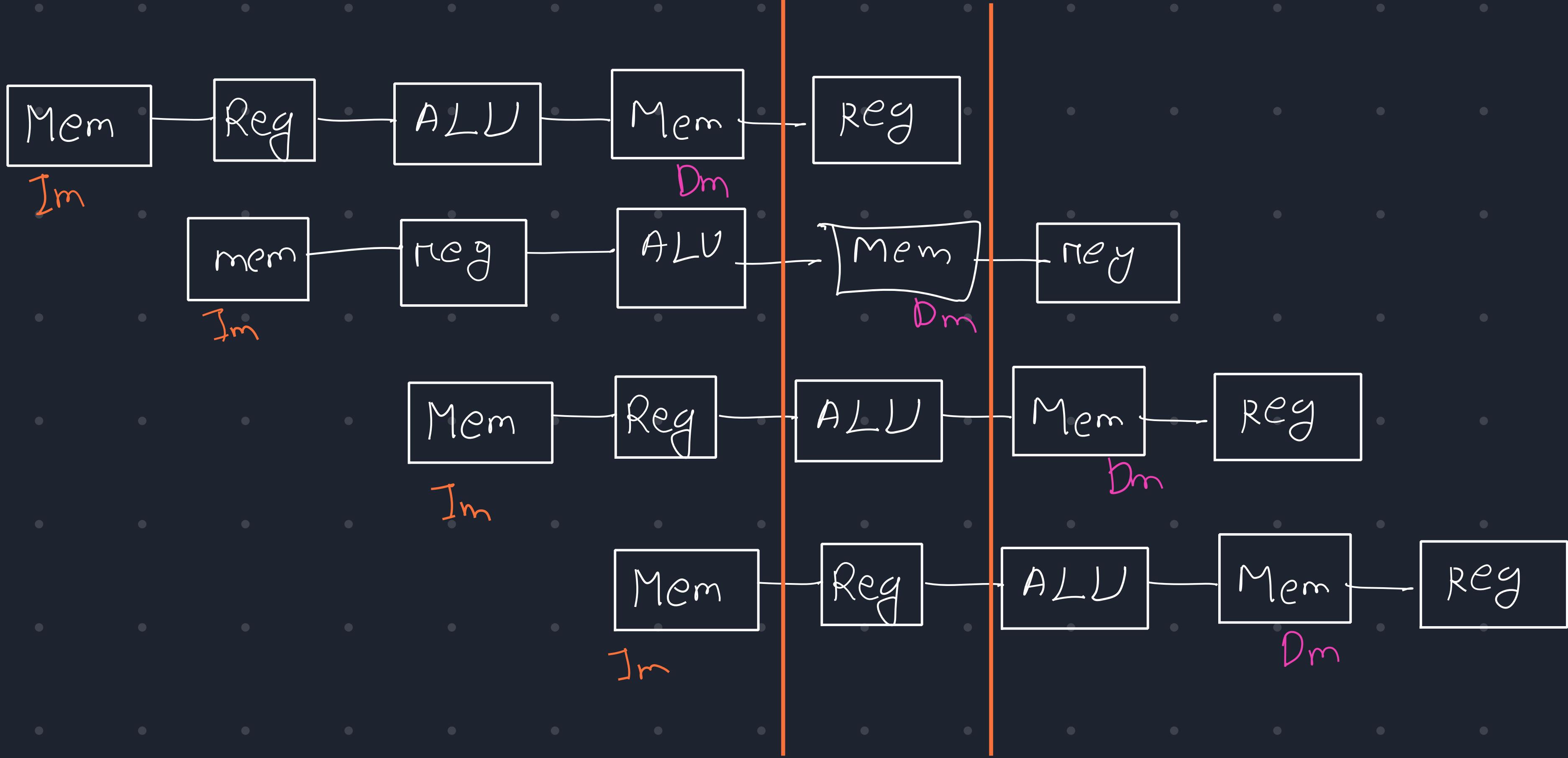


Im - Instruction memory

Dm - Data memory

add  $\$S_1, \$S_2, \$S_3$

$I_1$   
 $I_L$   
Sub  $\$S_3, \$S_1, \$S$



general write  
rising edge triggered  
triggered

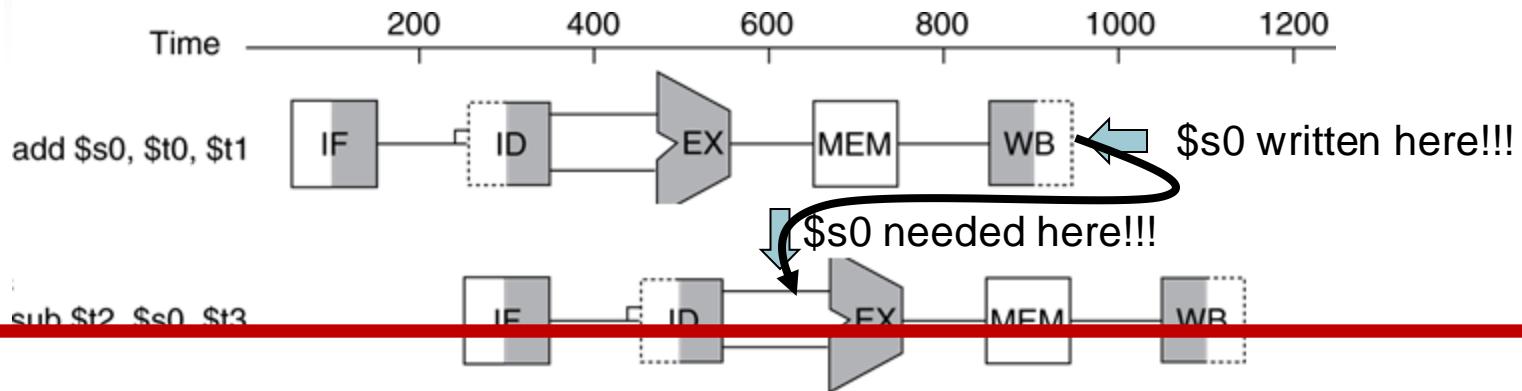
general write  
falling edge triggered

# Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

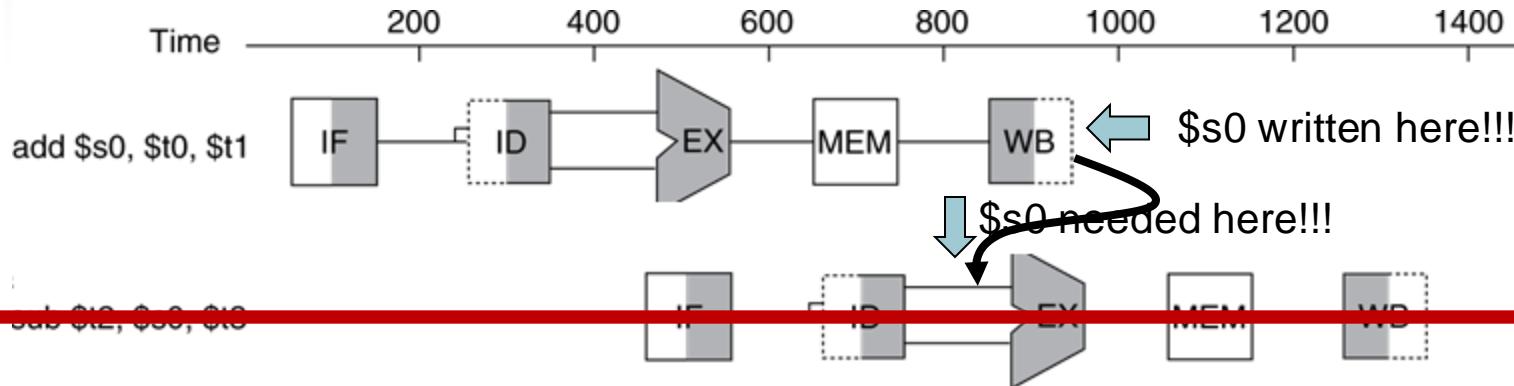
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add \$s0, \$t0, \$t1
  - sub \$t2, \$s0, \$t3



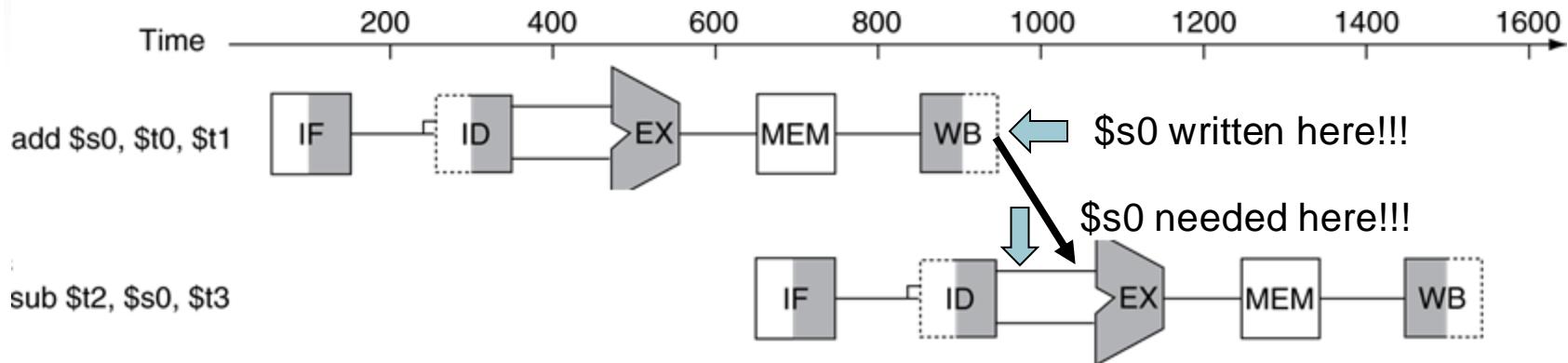
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add \$s0, \$t0, \$t1
  - sub \$t2, \$s0, \$t3



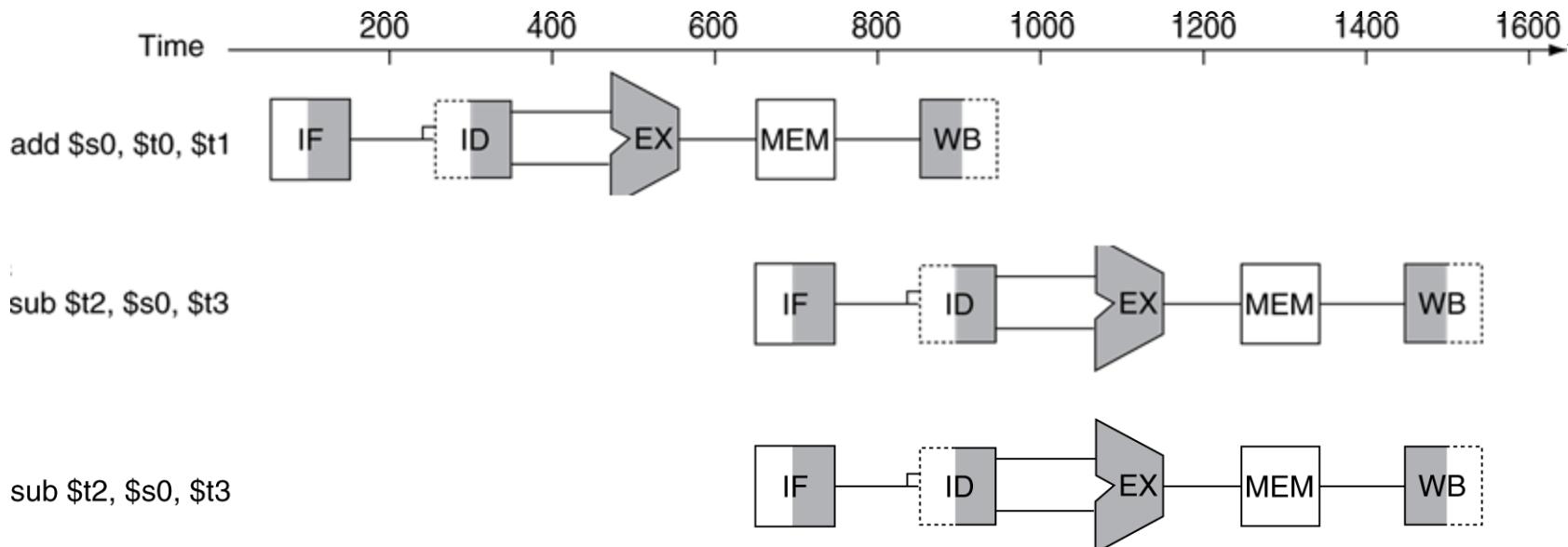
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add \$s0, \$t0, \$t1
  - sub \$t2, \$s0, \$t3



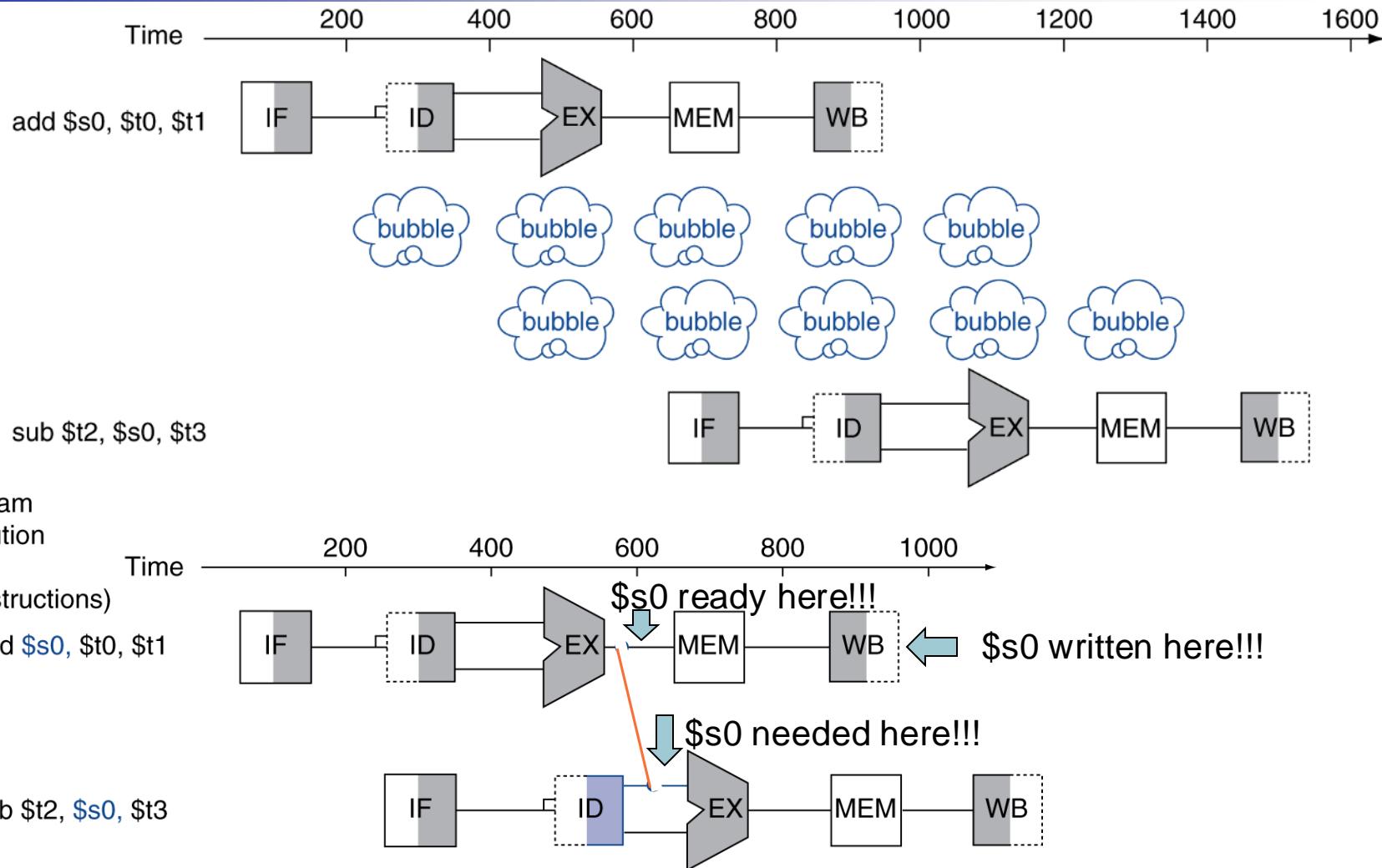
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add \$s0, \$t0, \$t1
  - sub \$t2, \$s0, \$t3



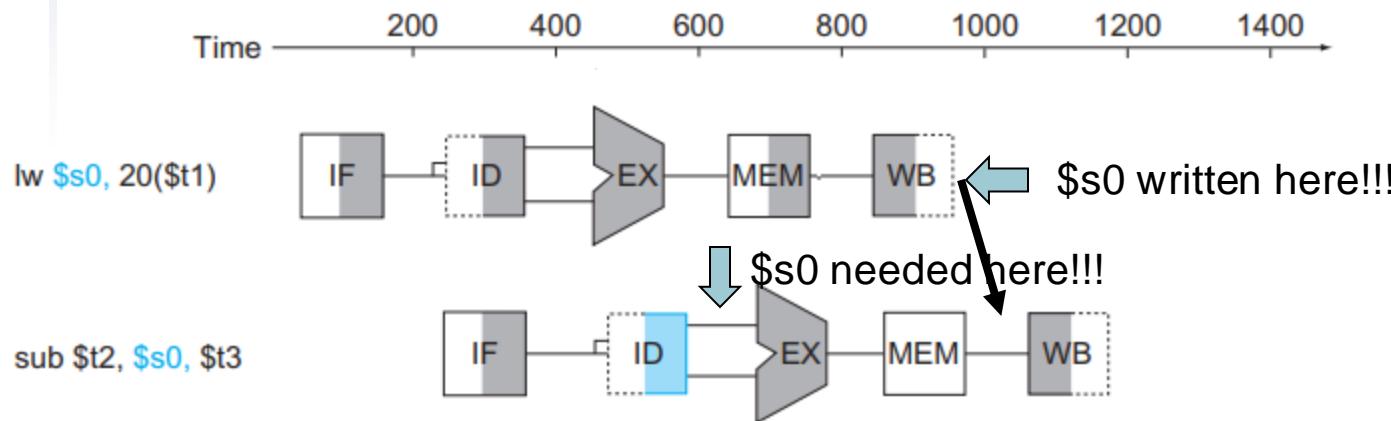
# Forwarding (aka Bypassing)

## COMPARE



# Load-Use Data Hazard

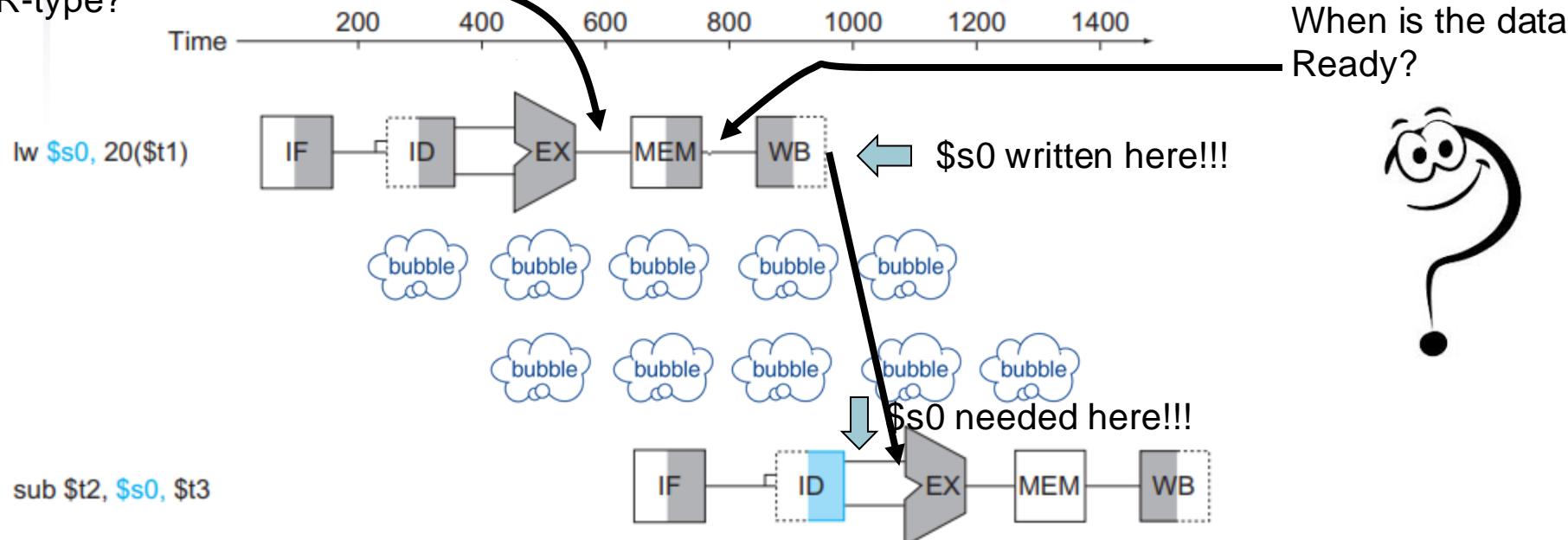
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



# Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!

When WAS the  
data Ready for  
R-type?



Can we solve by FORWARDING?

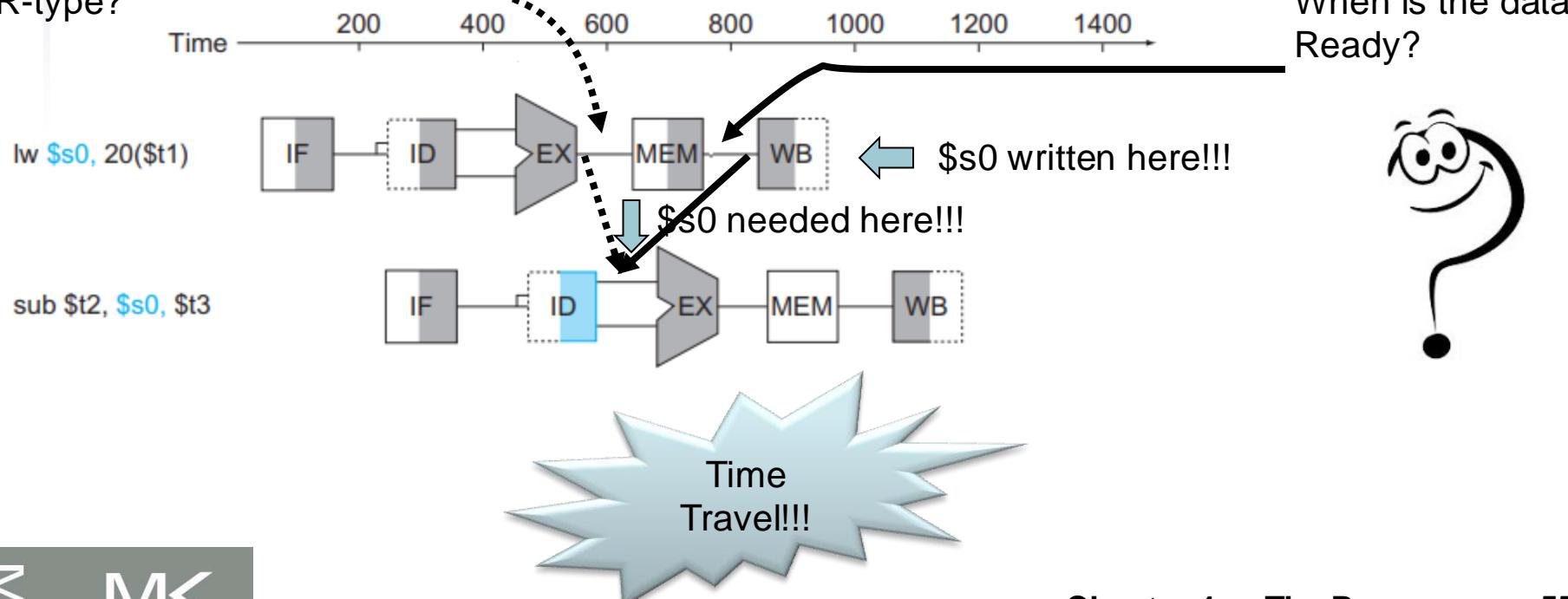
# Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!

When WAS the  
data Ready for  
R-type?

Can we solve by  
FORWARDING?

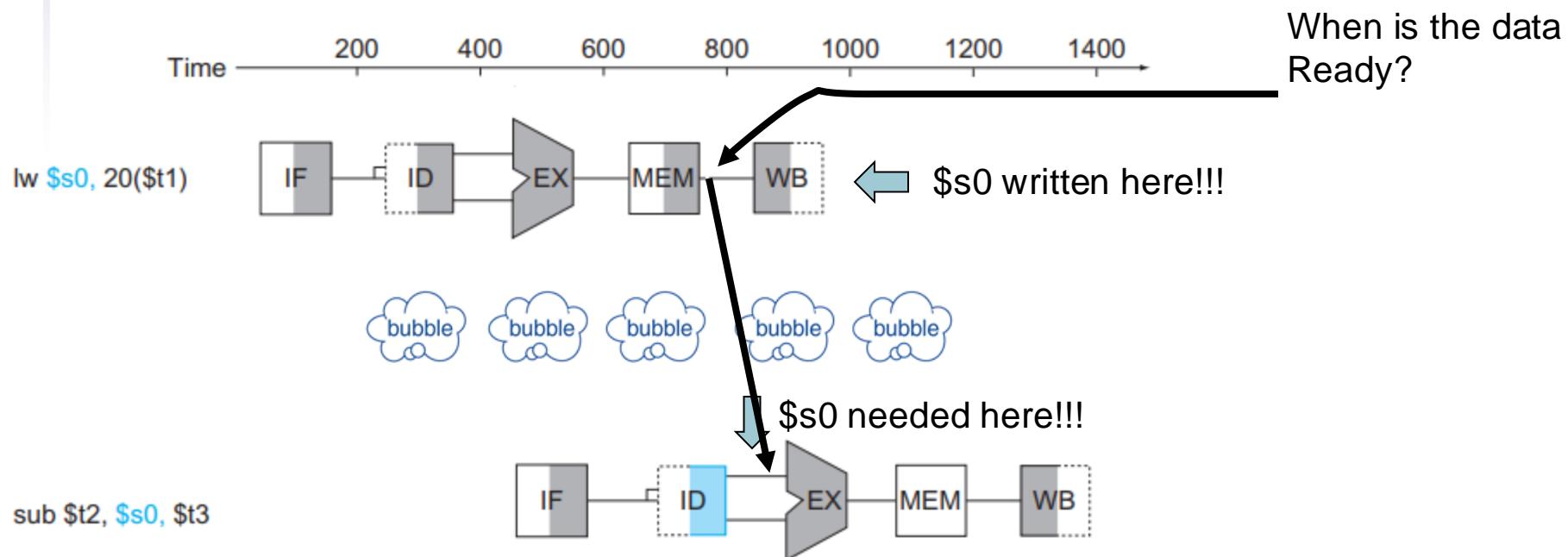
When is the data  
Ready?



# Load-Use Data Hazard

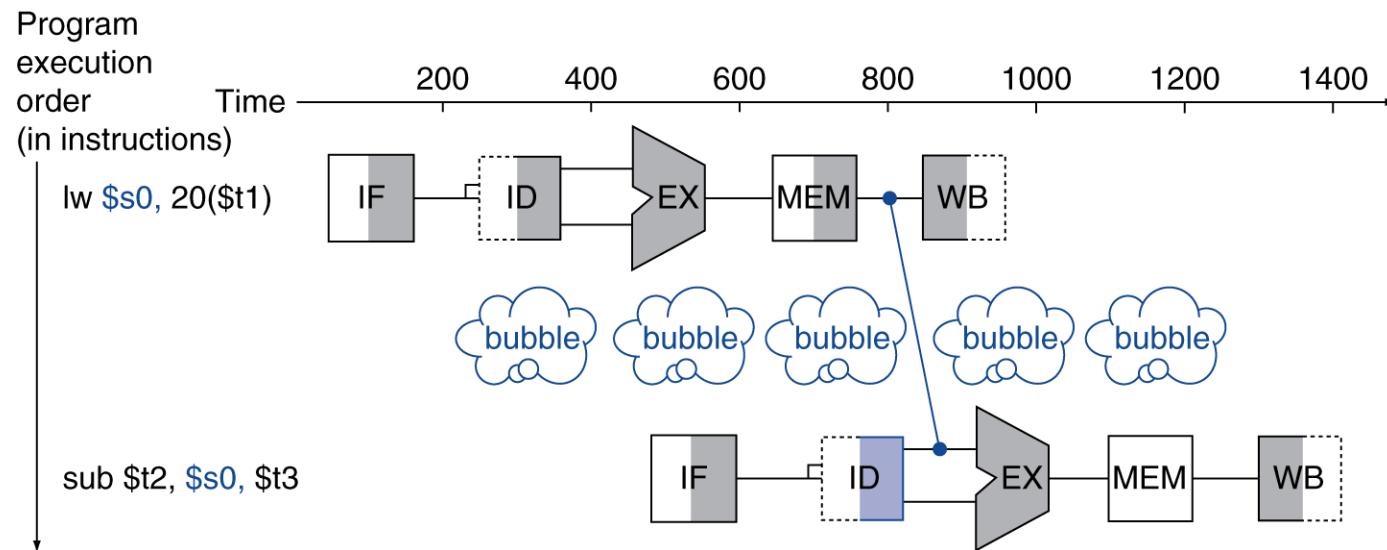
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!

Can we solve by FORWARDING?



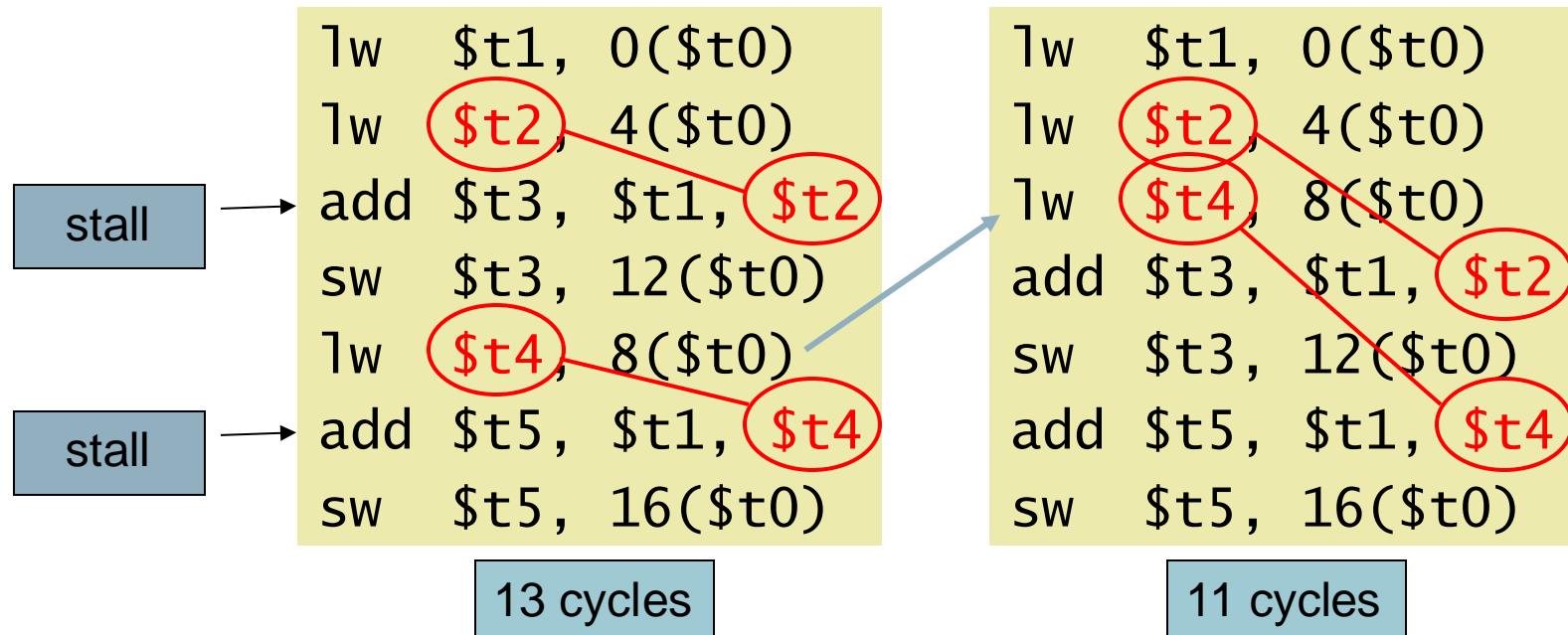
# Load-Use Data Hazard

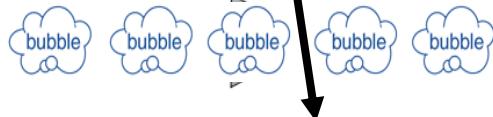
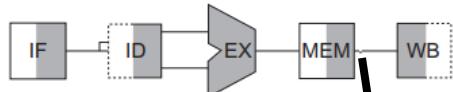
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



# Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for  $A = B + E; C = B + F;$





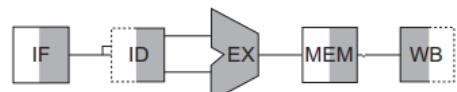
stall

```

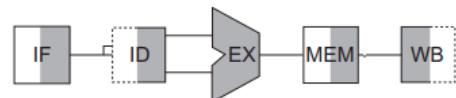
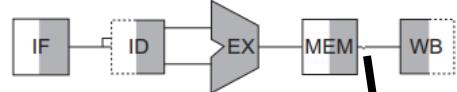
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add $t3, $t1, $t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add $t5, $t1, $t4
sw    $t5, 16($t0)

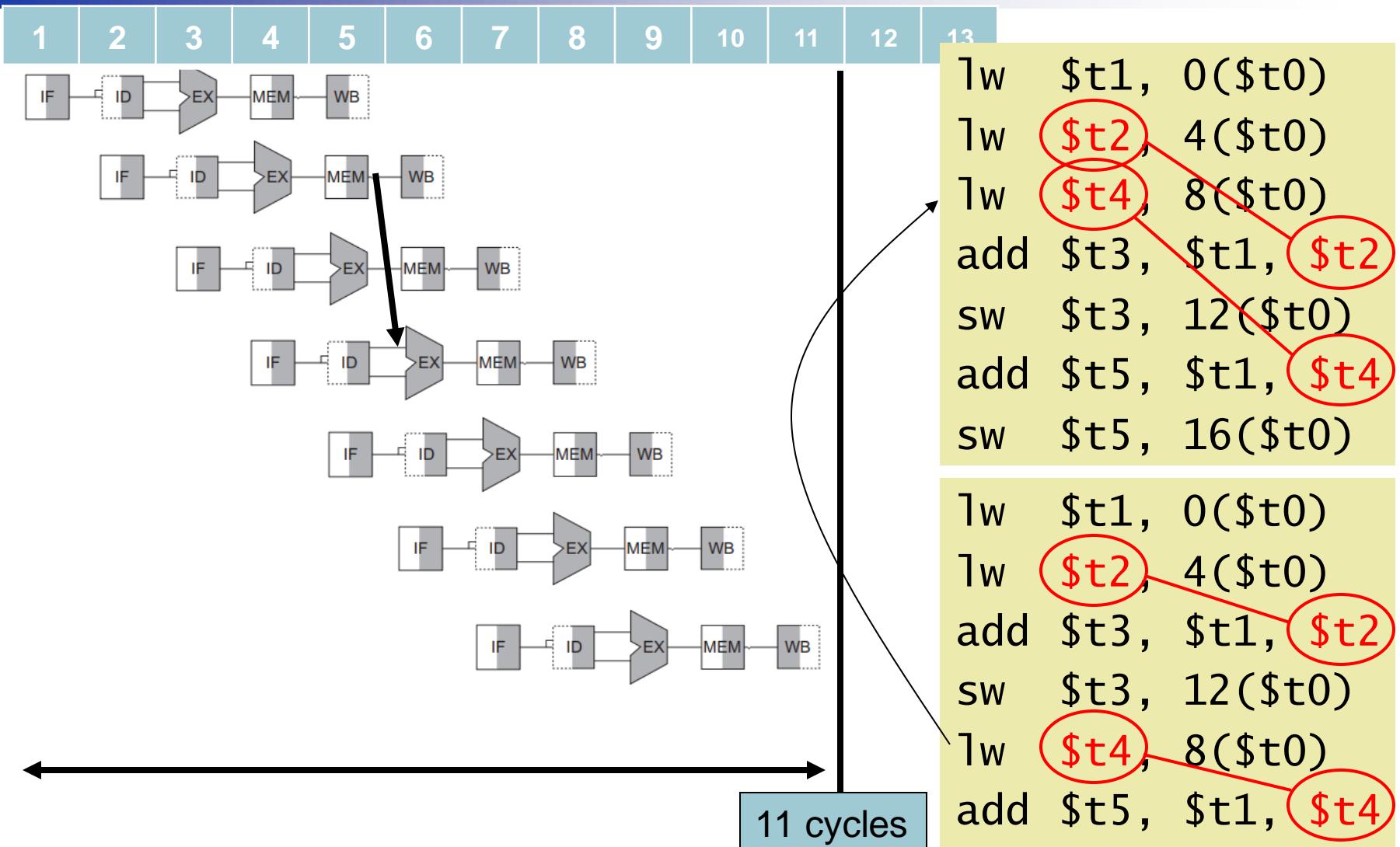
```

13 cycles



stall

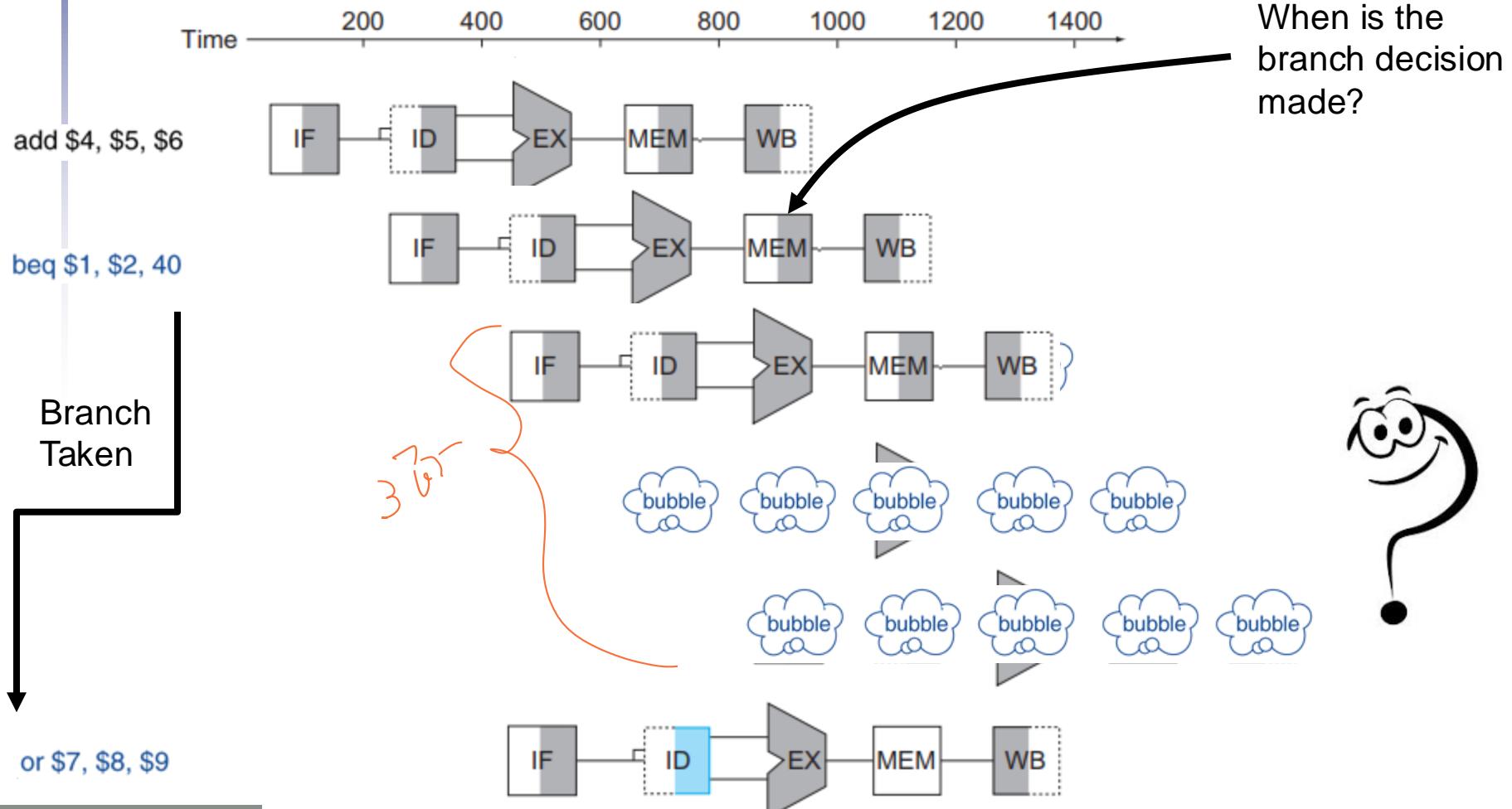




# Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

# Branch Scenario

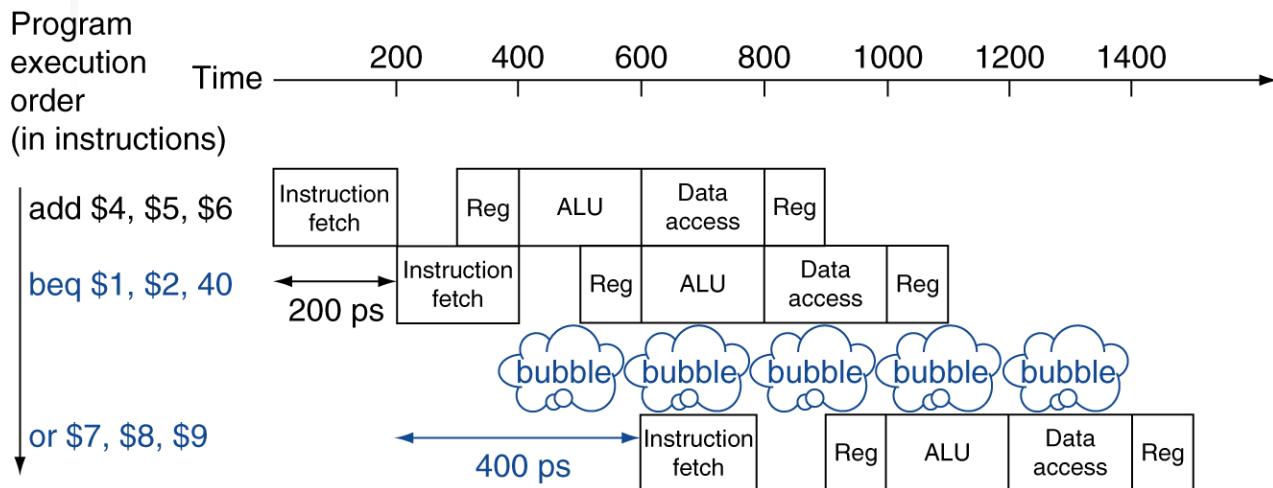


Top tested loop → TLSI check, then inside loop

bottom " → TLSI loop ↗, then check

# Stall on Branch (Improved)

- Wait until branch outcome determined before fetching next instruction
- 17% branch
- How much is the slowdown?



83% => 1 cycle

17% => 2 cycles

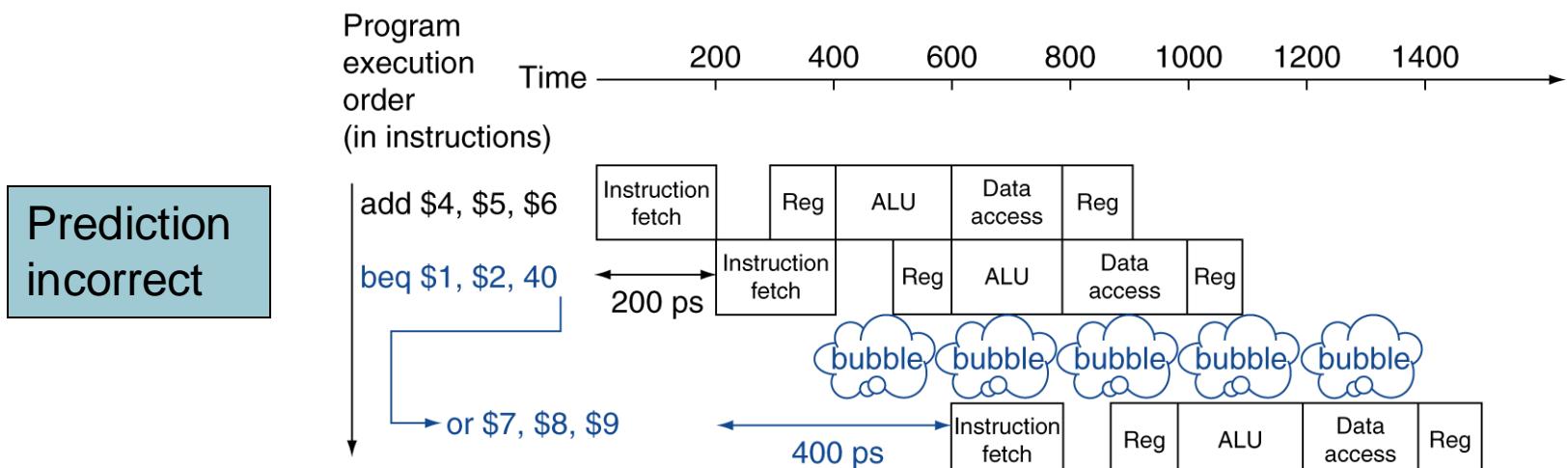
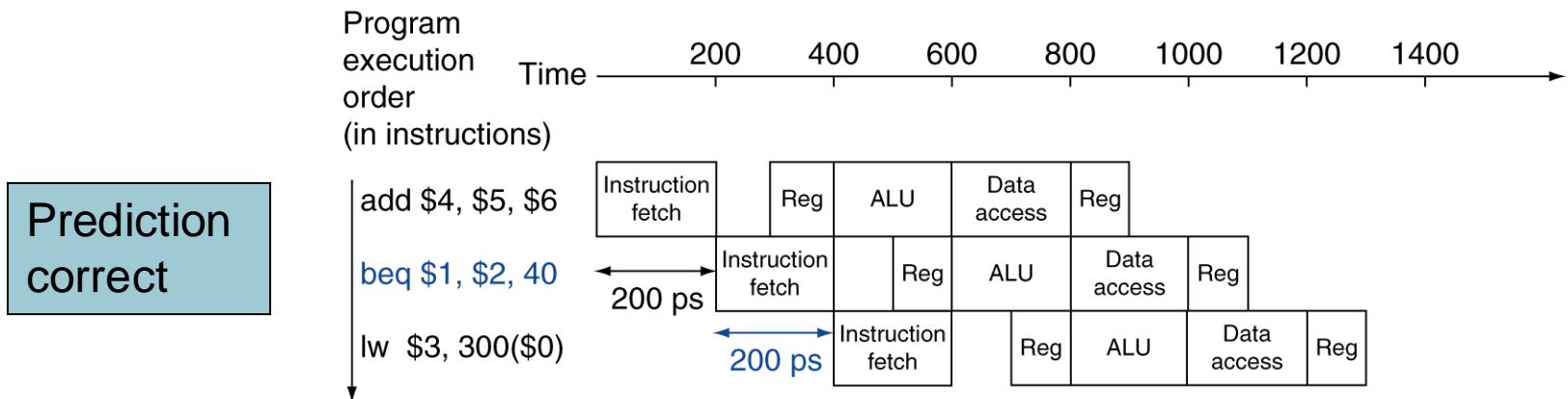
So, effective CPI

$$\begin{aligned} &= \frac{1 \times 83 + 2 \times 1.7}{100} \\ &= \frac{83 + 34}{100} \\ &= 1.17 \end{aligned}$$

# Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# MIPS with Predict Not Taken

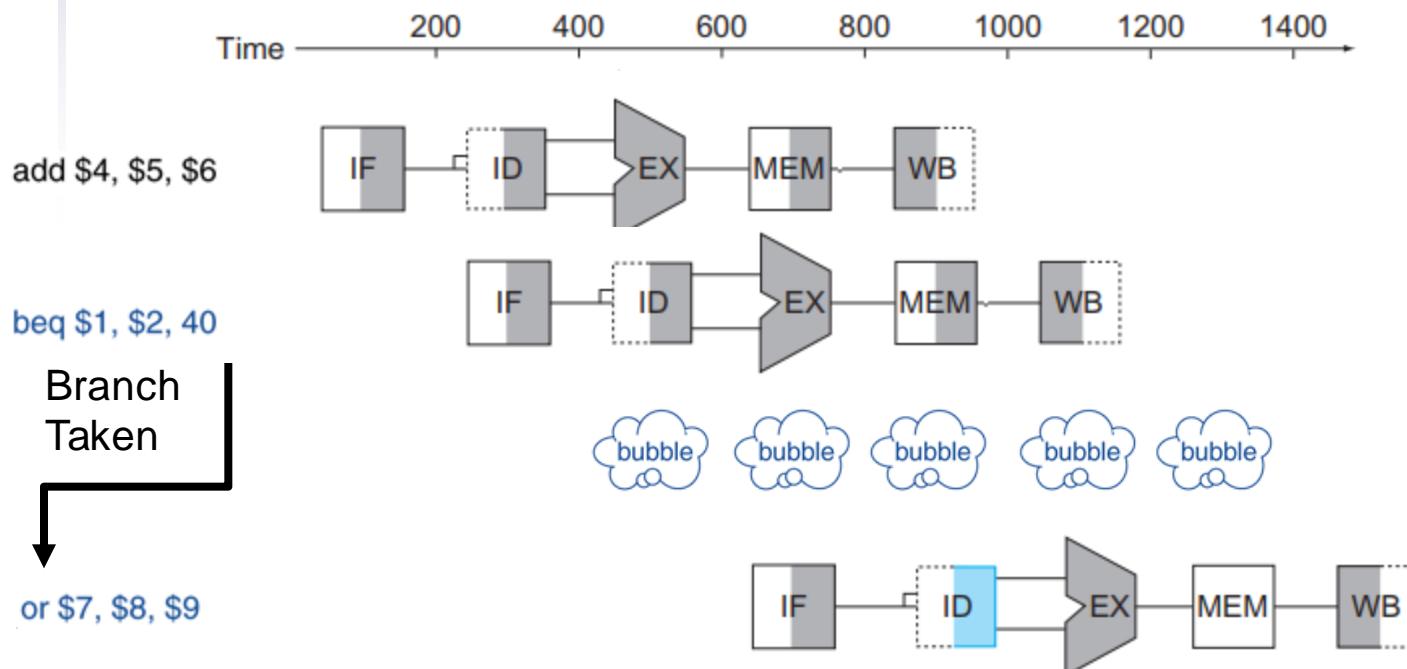


# More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

# Delayed Branch

- Always execute the next sequential instruction
  - i.e., taken branch is always delayed
  - The delay is hidden by rearrangement



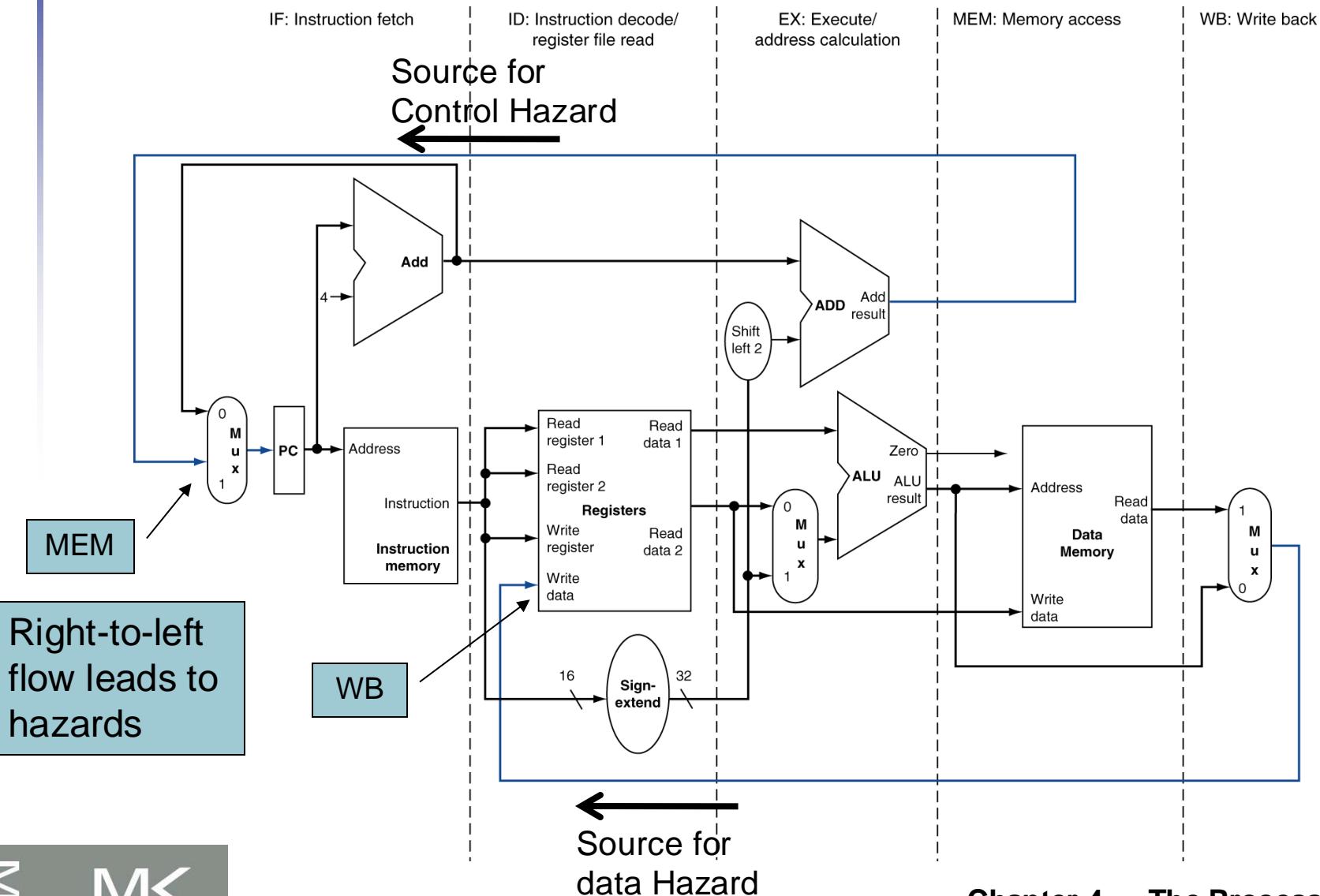
MIPS Solution to Control Hazard

# Pipeline Summary

## The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

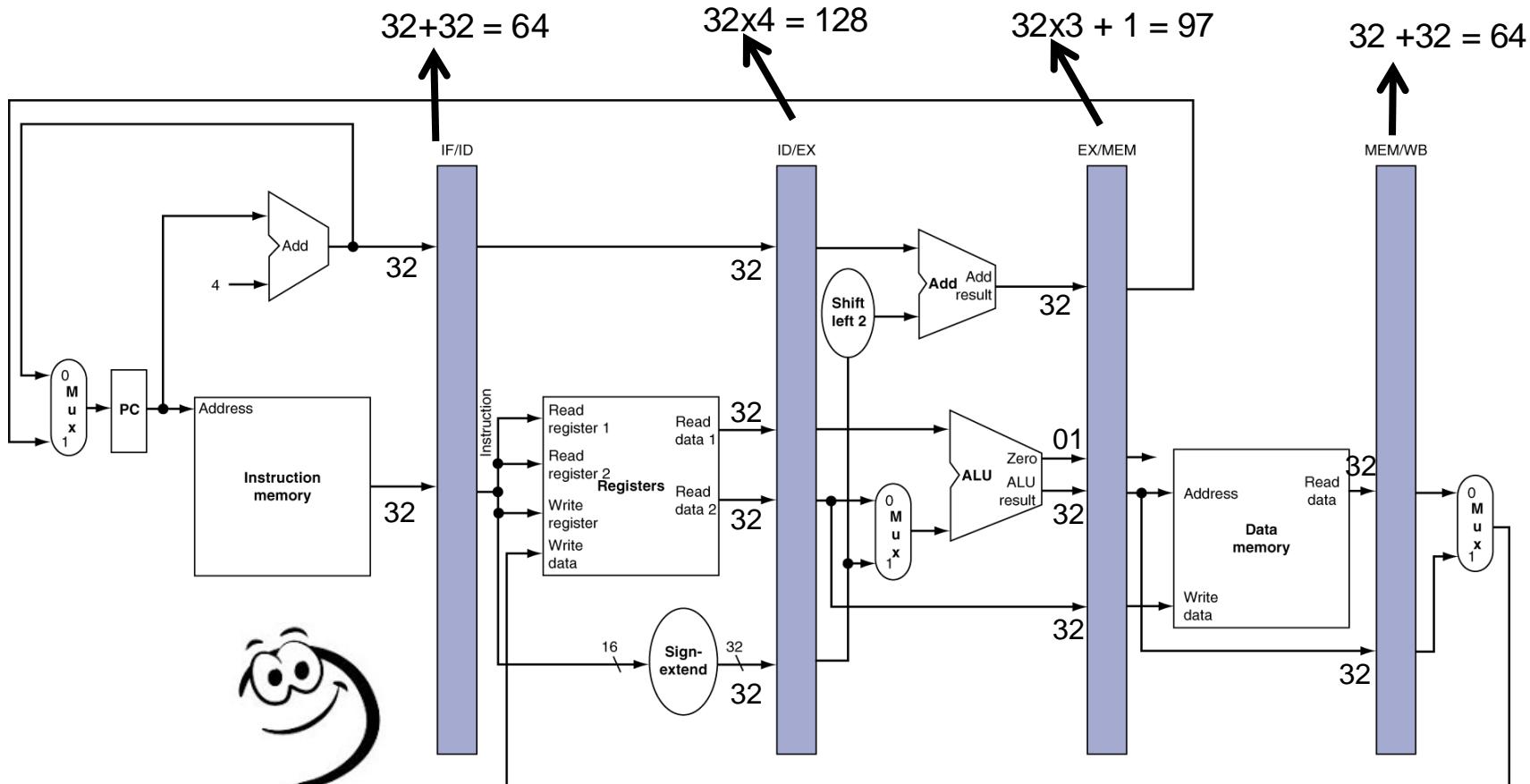
# MIPS Pipelined Datapath



# Pipeline registers

We may need to update the sizes as we proceed!!!

- Need registers between stages
  - To hold information produced in previous cycle



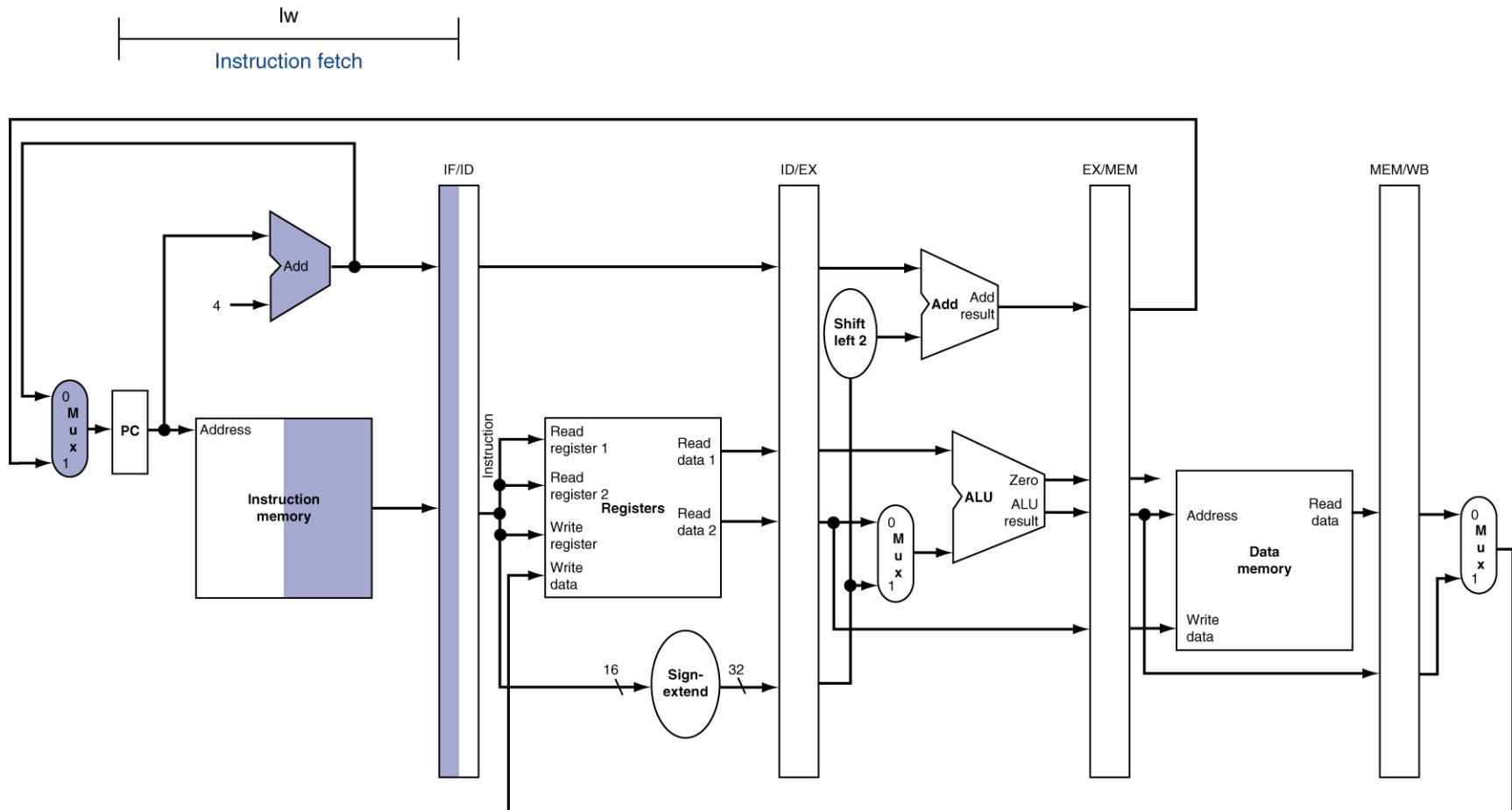
What is the size of the pipeline registers?

# Pipeline Operation

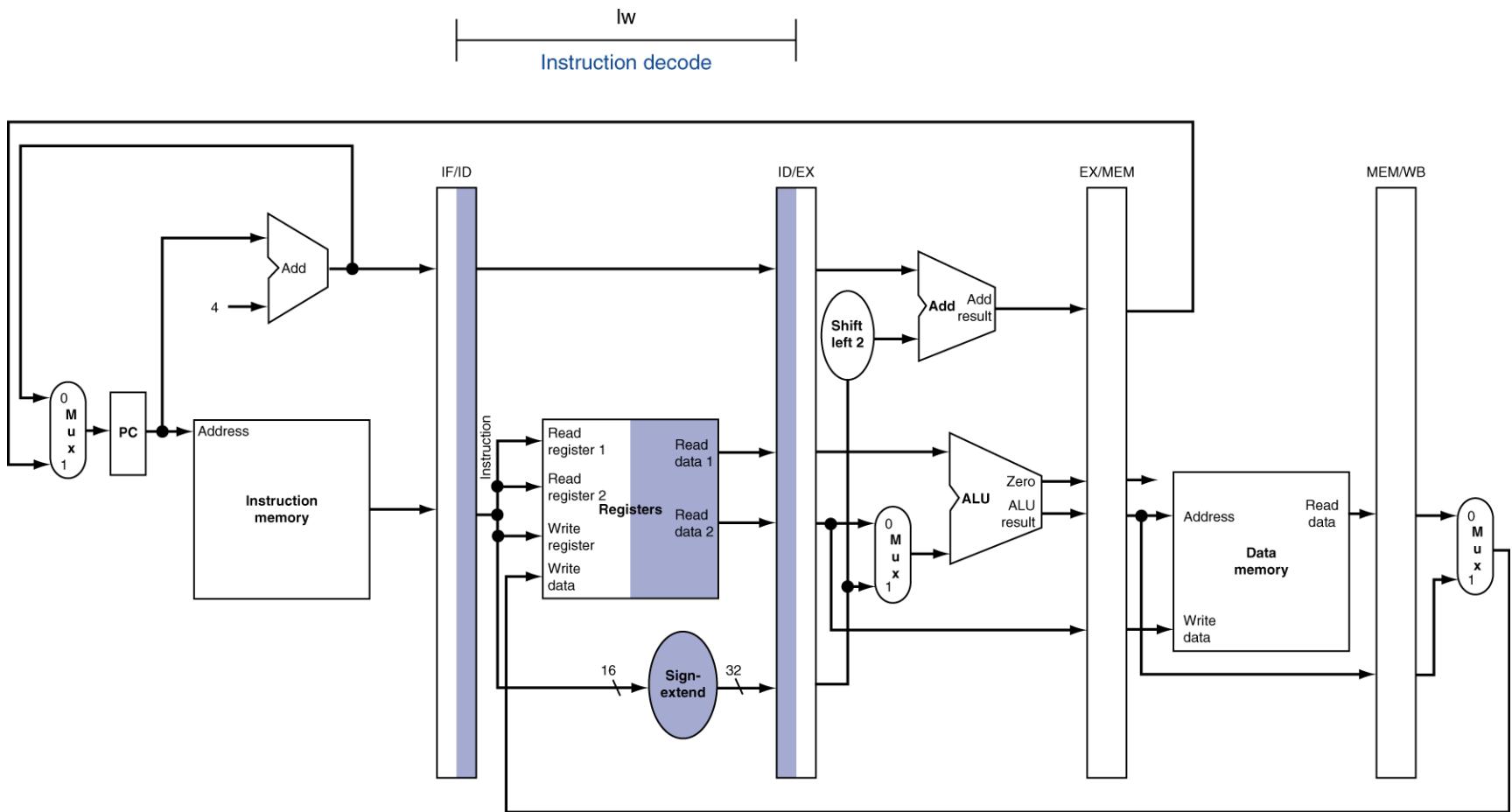
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We'll look at “single-clock-cycle” diagrams for load & store

# IF for Load

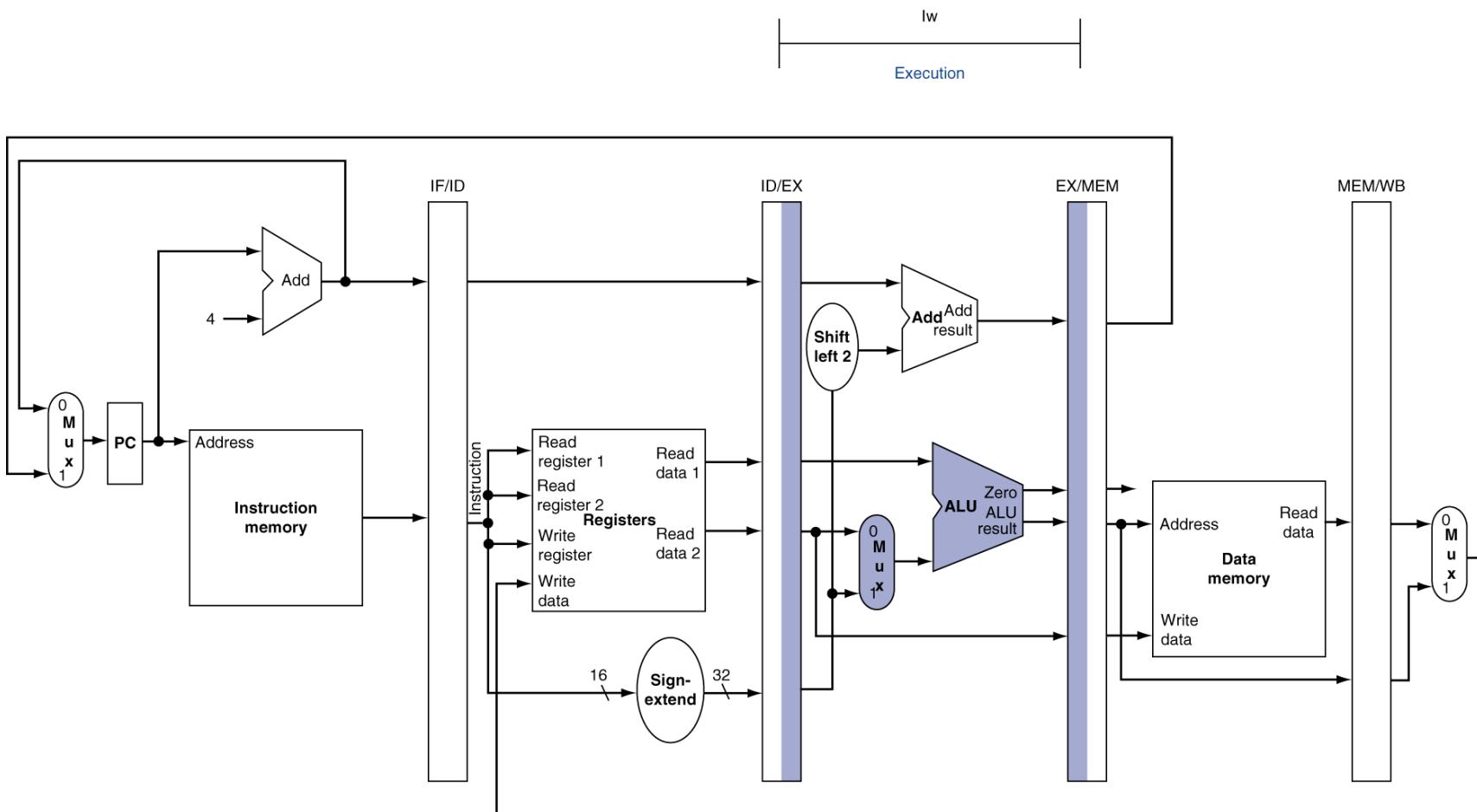
left half  $\hookrightarrow$  shedding - write  
right : -  $\rightarrow$  read



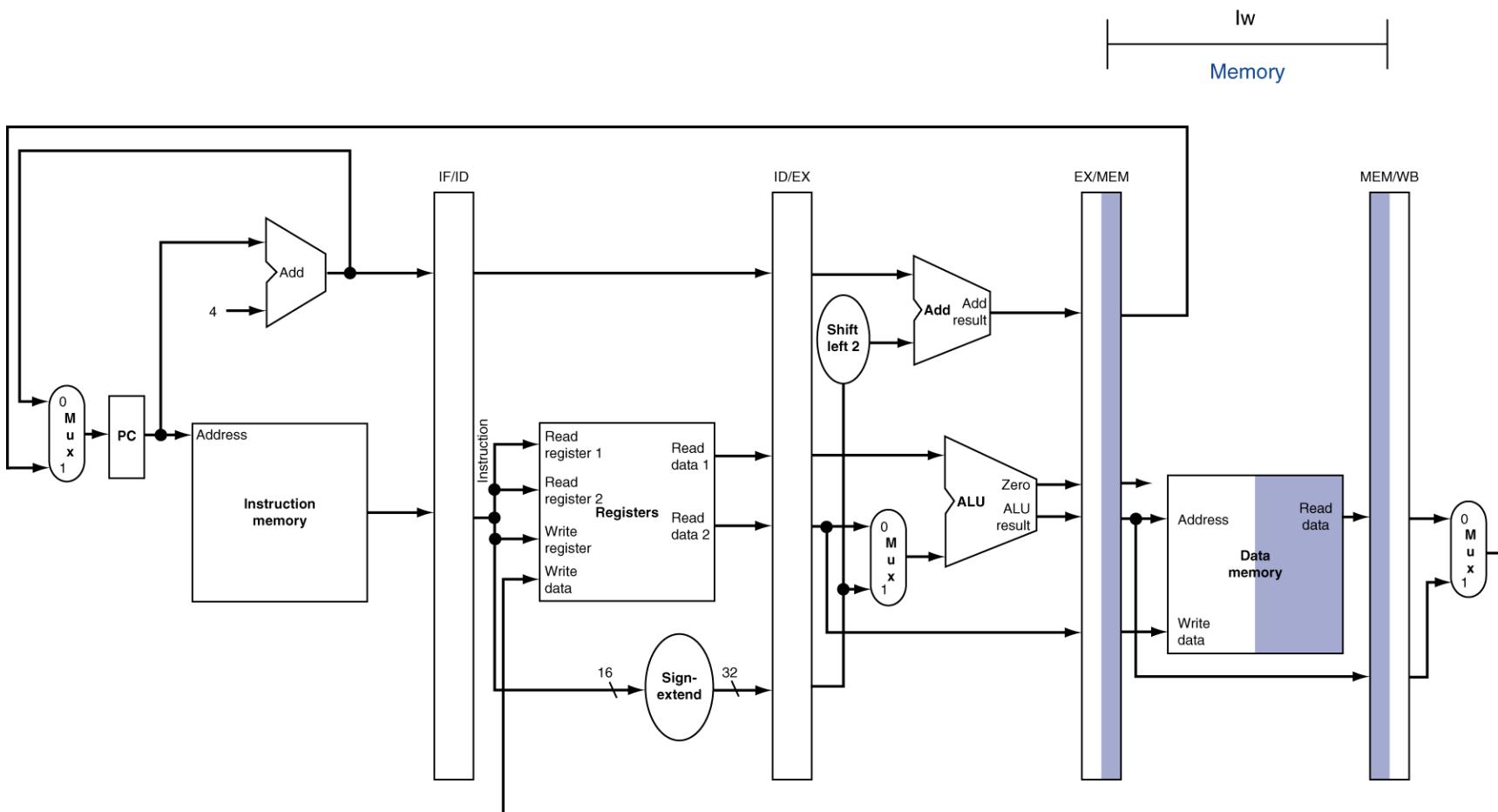
# ID for Load



# EX for Load

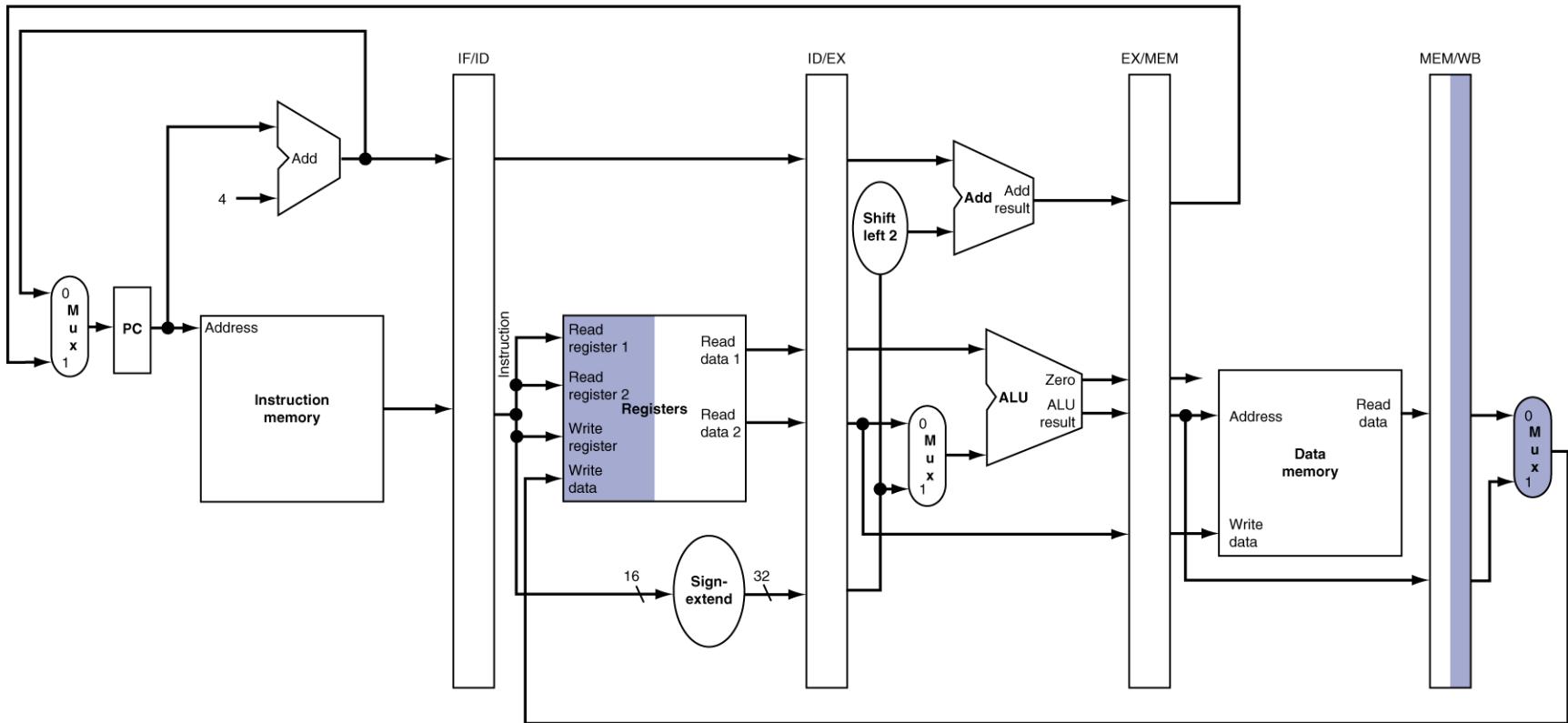


# MEM for Load

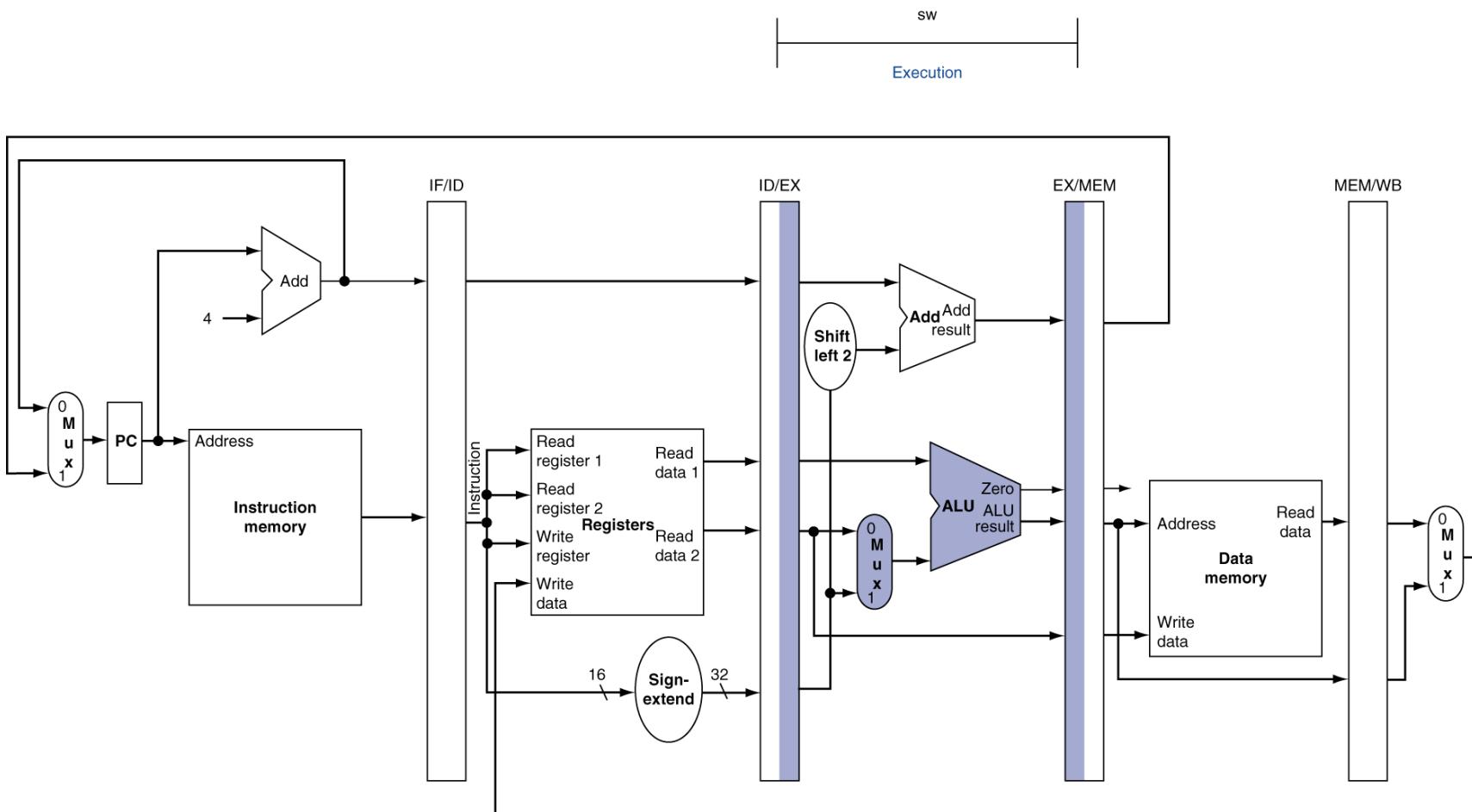


# WB for Load

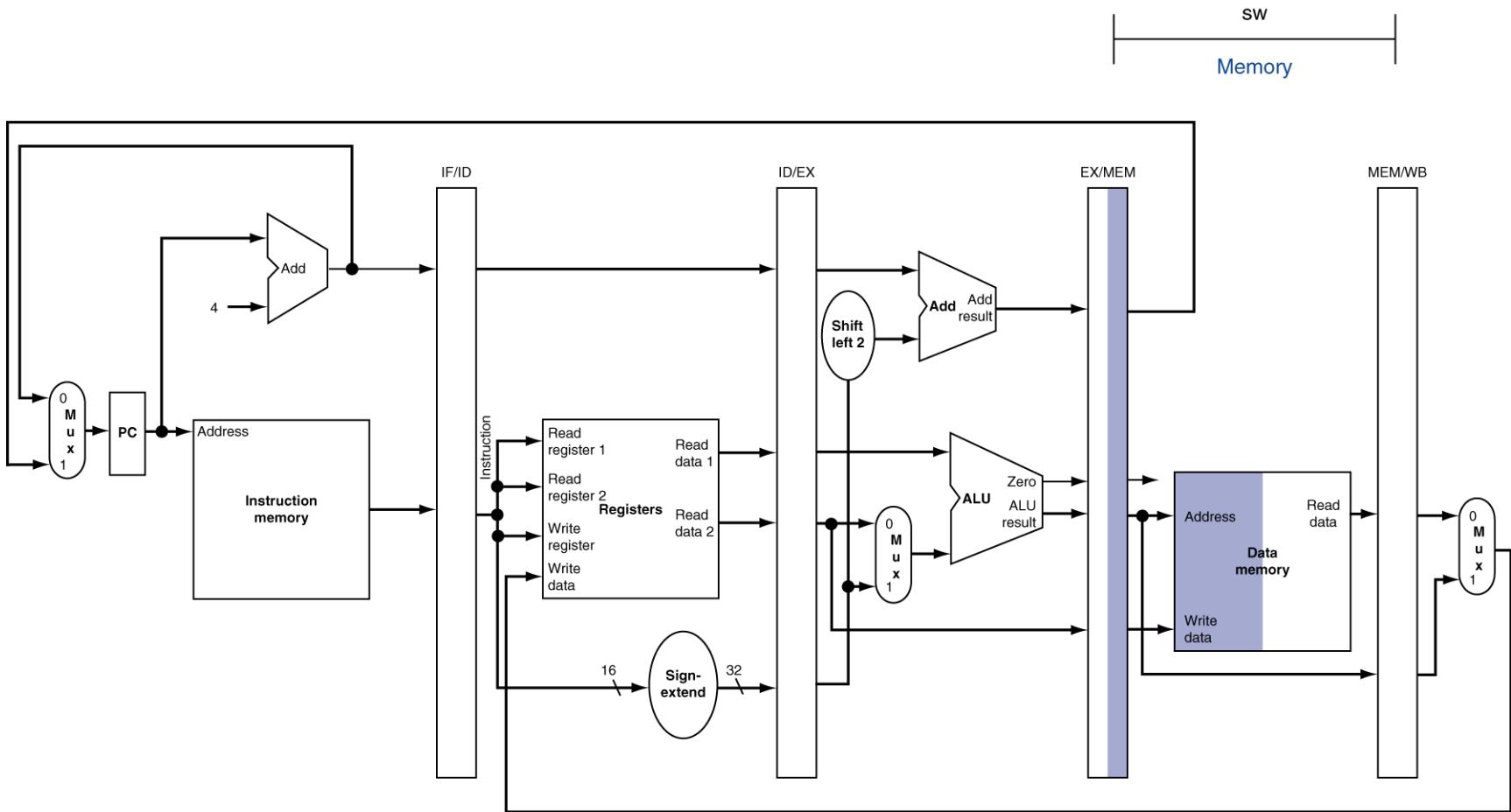
lw  
Write back



# EX for Store

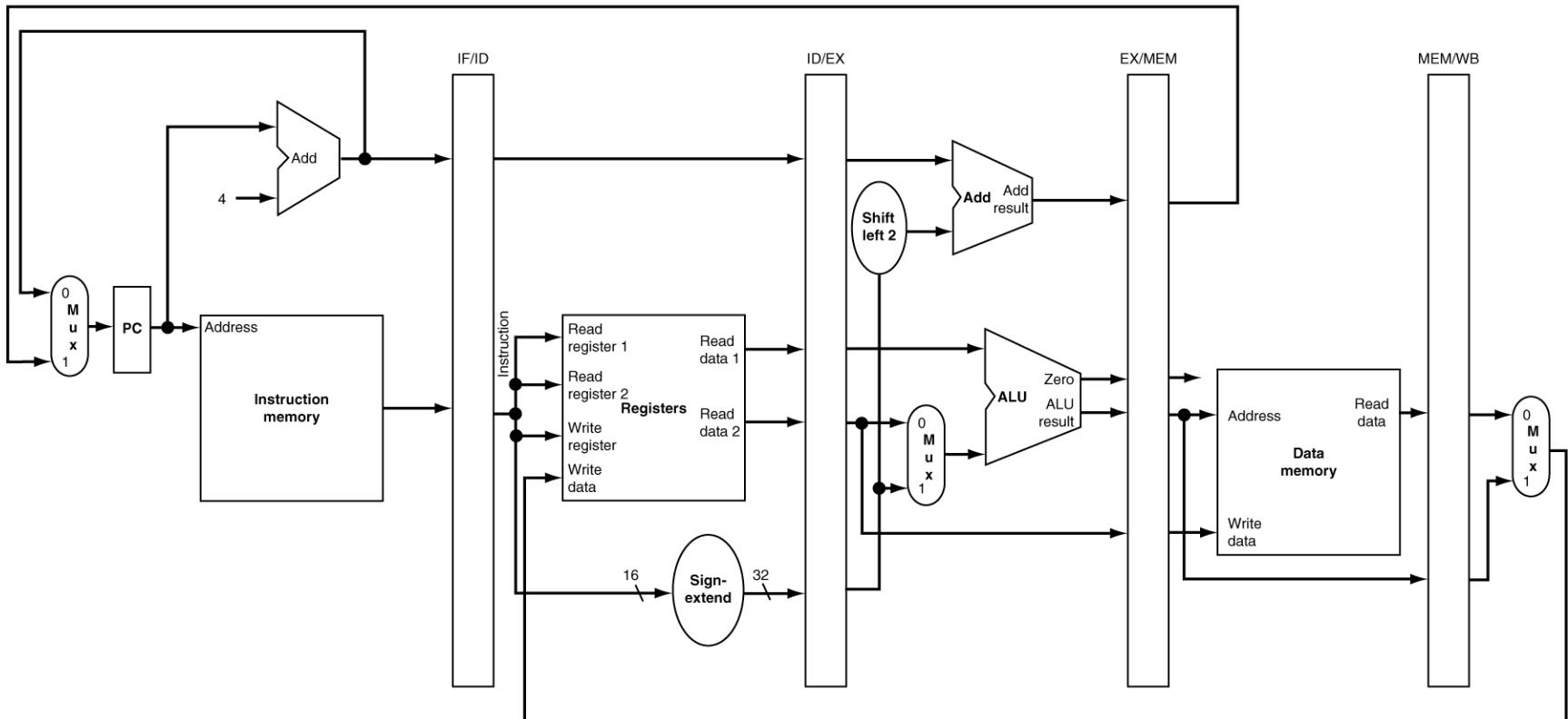


# MEM for Store



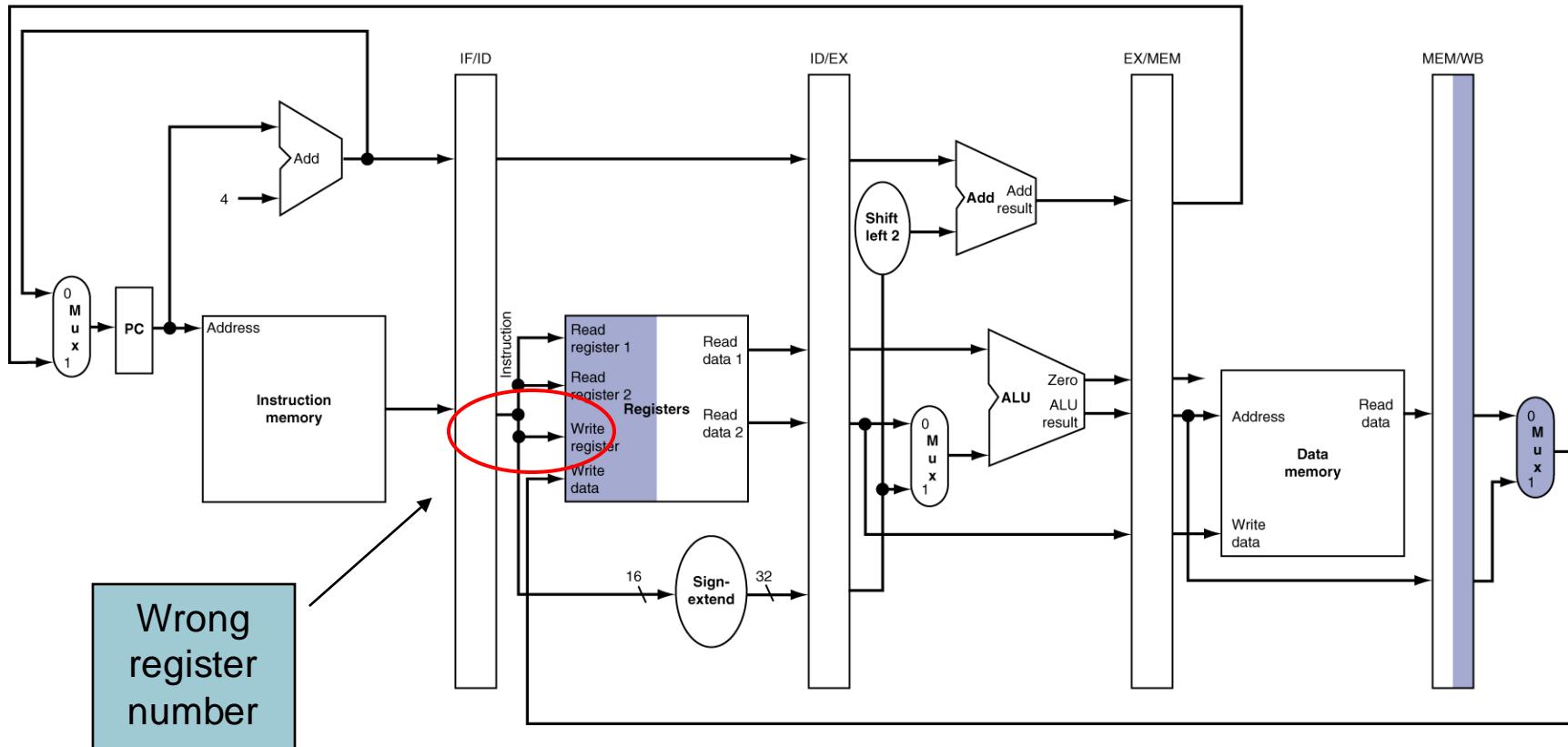
# WB for Store

SW  
Write-back



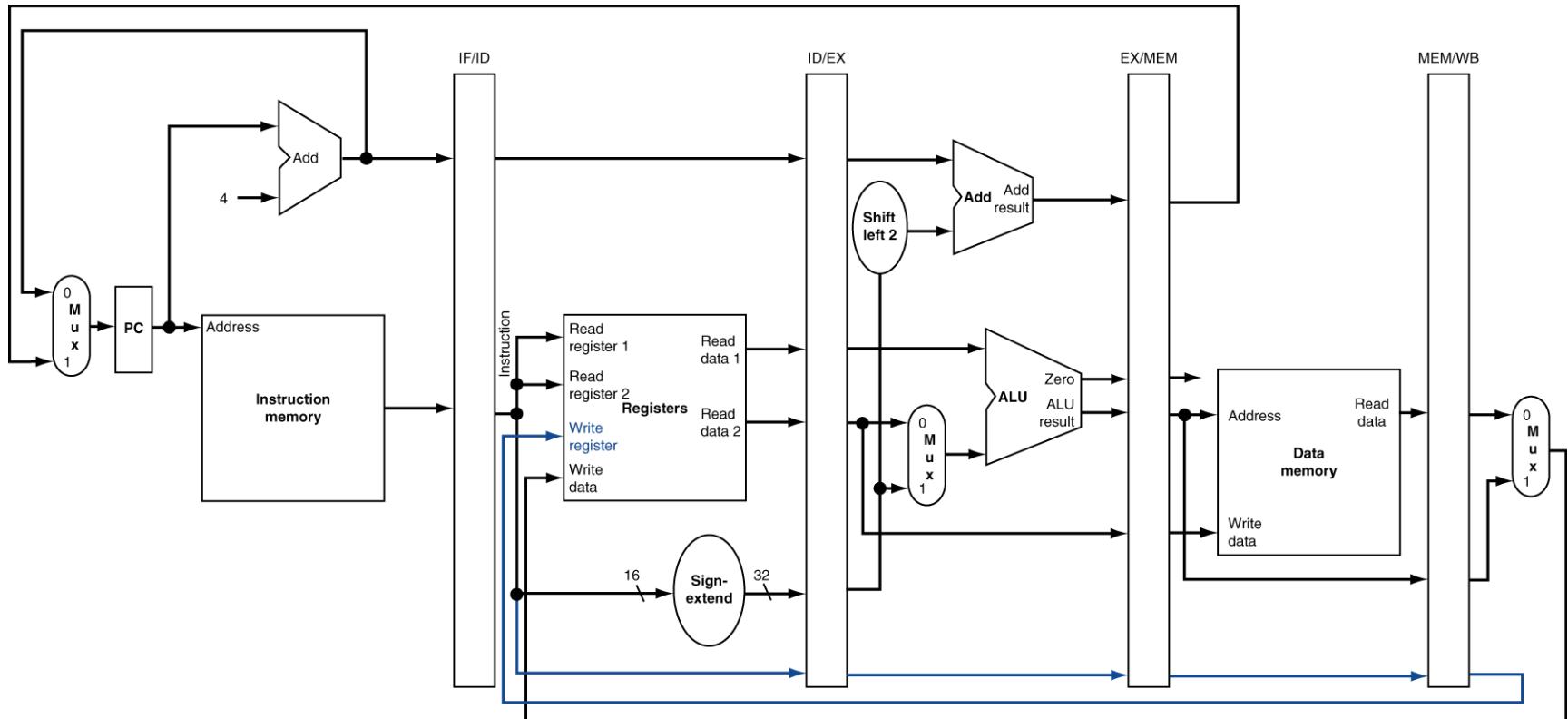
# WB for Load Revisited

lw  
Write back



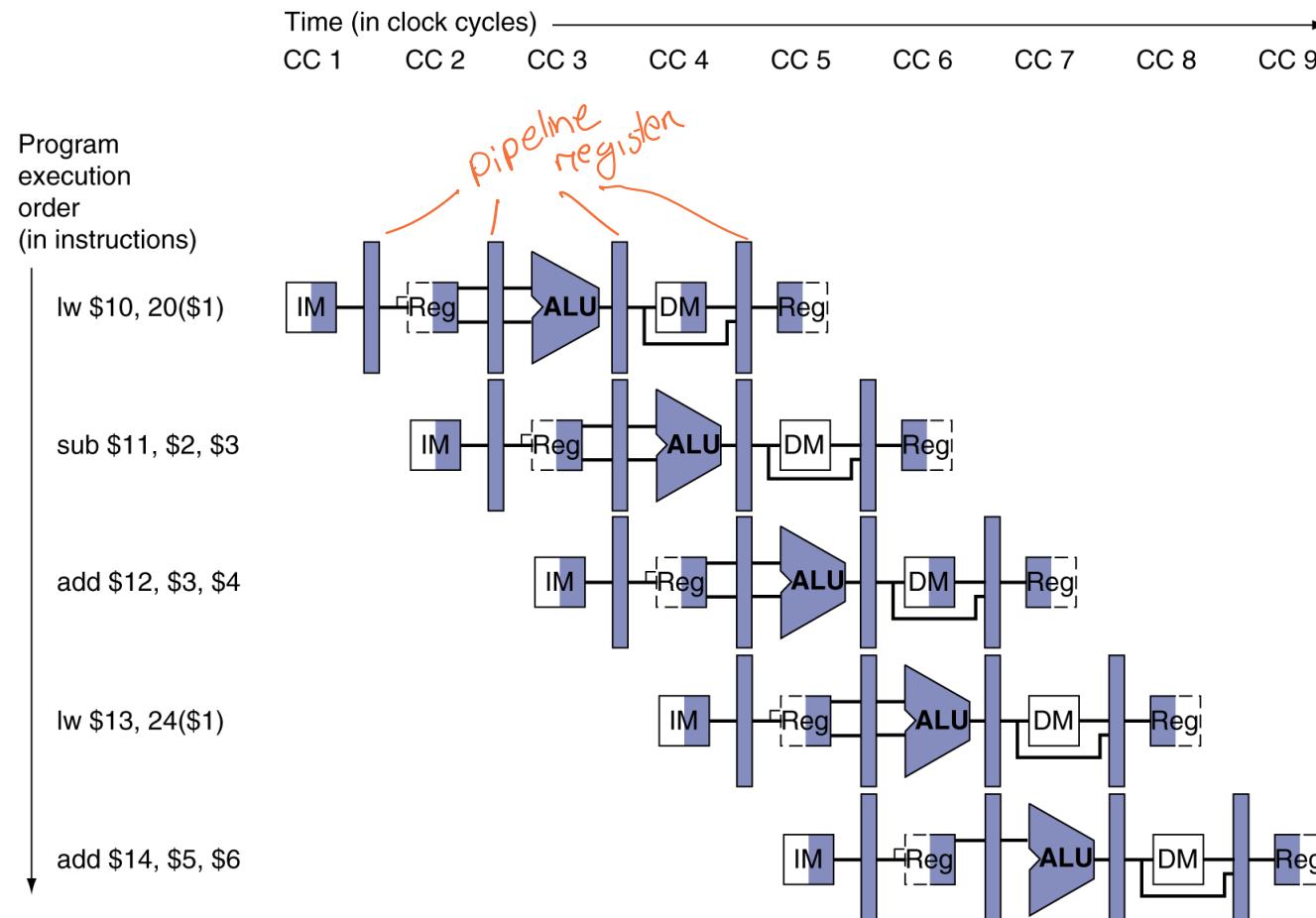
Wrong  
register  
number

# Corrected Datapath for Load



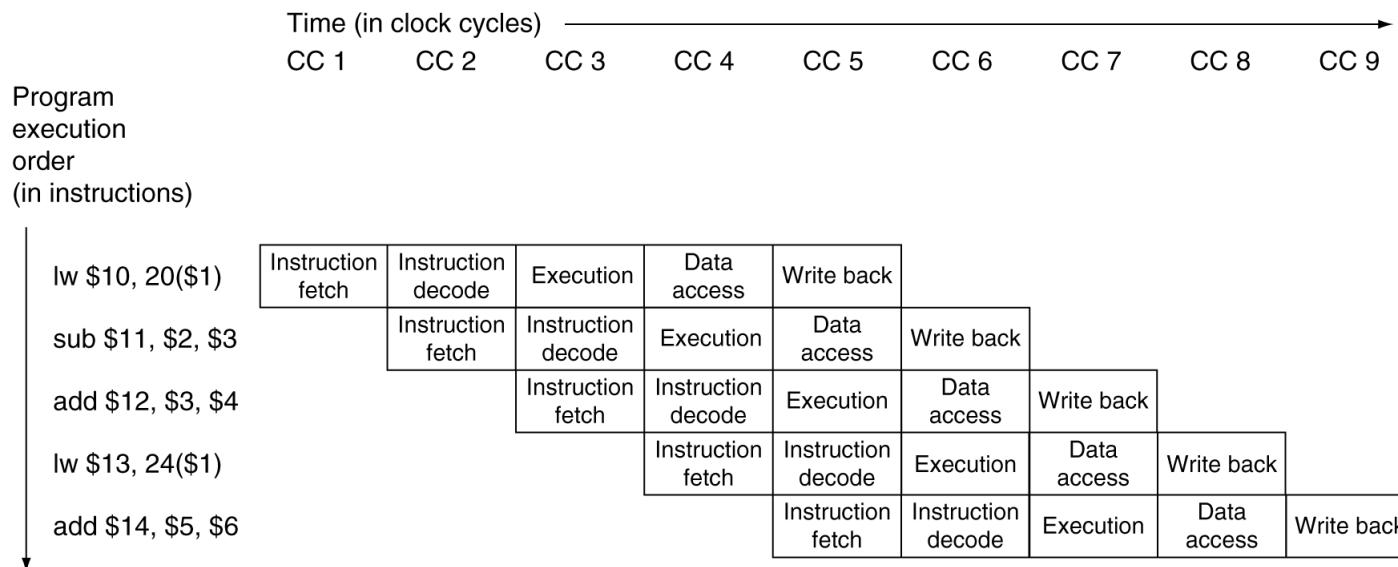
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



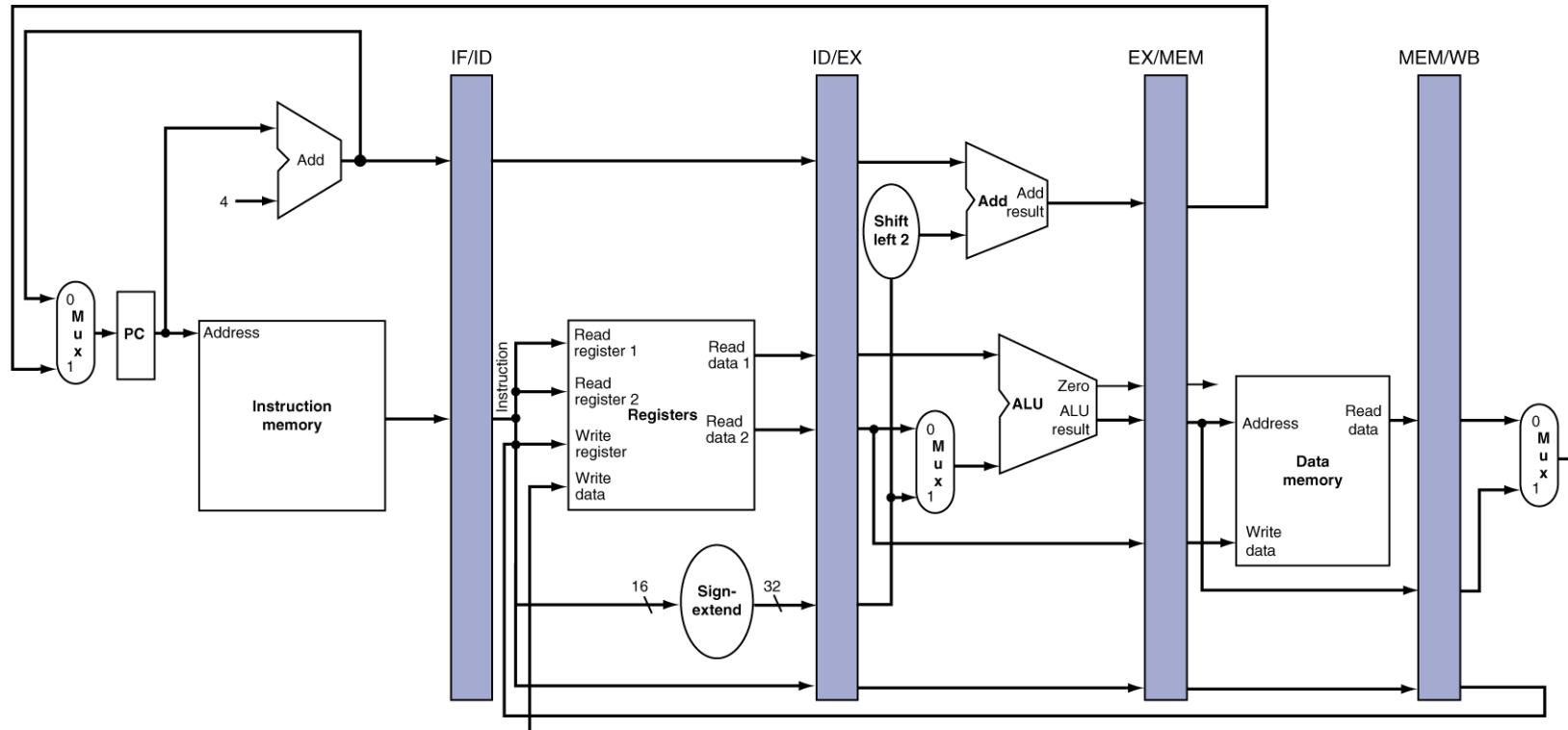
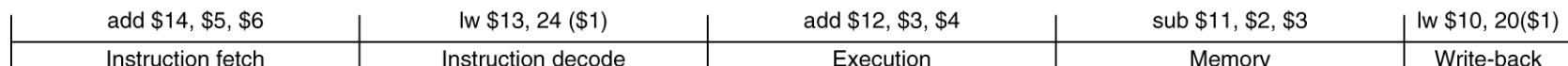
# Multi-Cycle Pipeline Diagram

## Traditional form



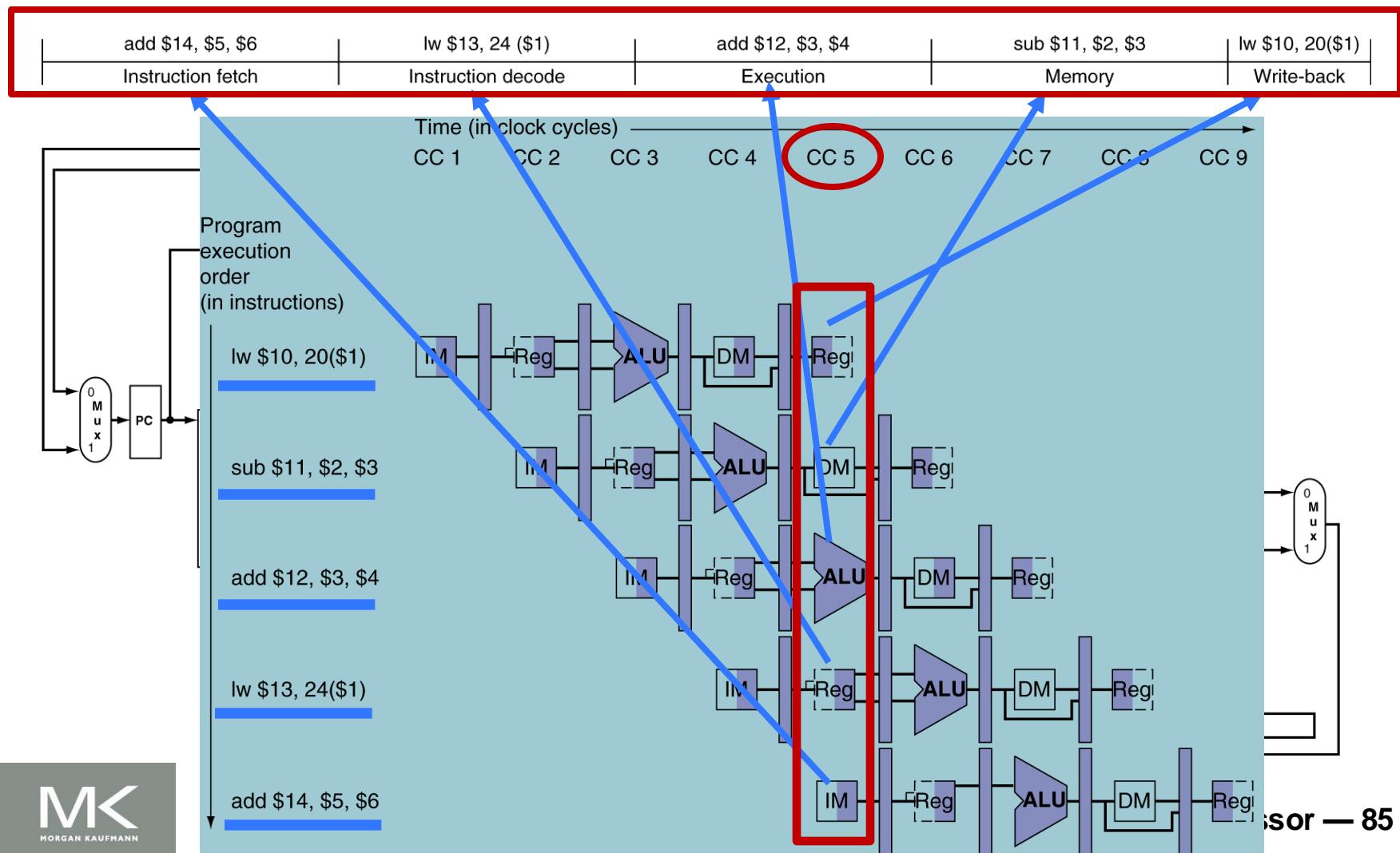
# Single-Cycle Pipeline Diagram

## State of pipeline in a given cycle

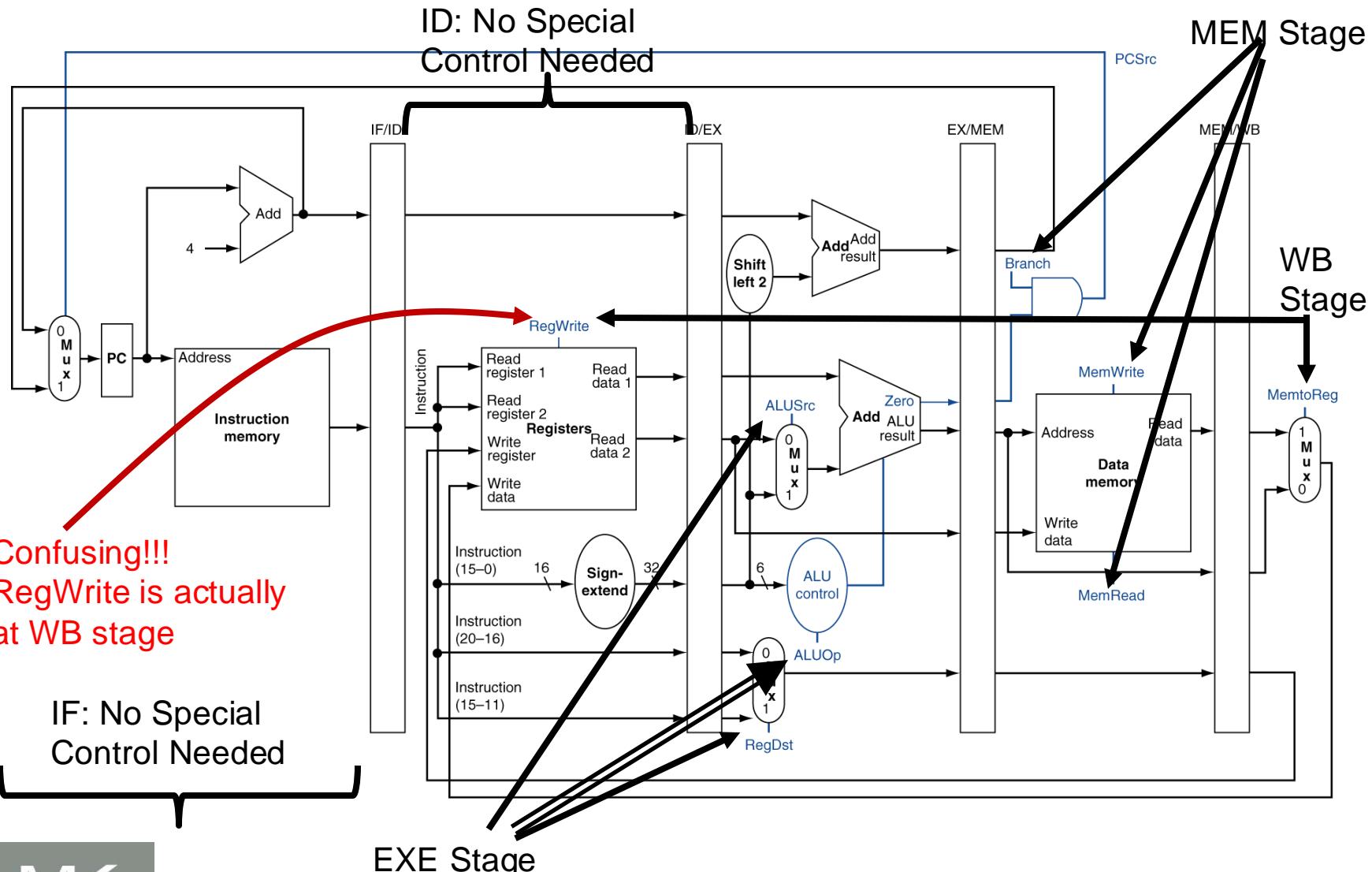


# Single-Cycle Pipeline Diagram

## State of pipeline in a given cycle

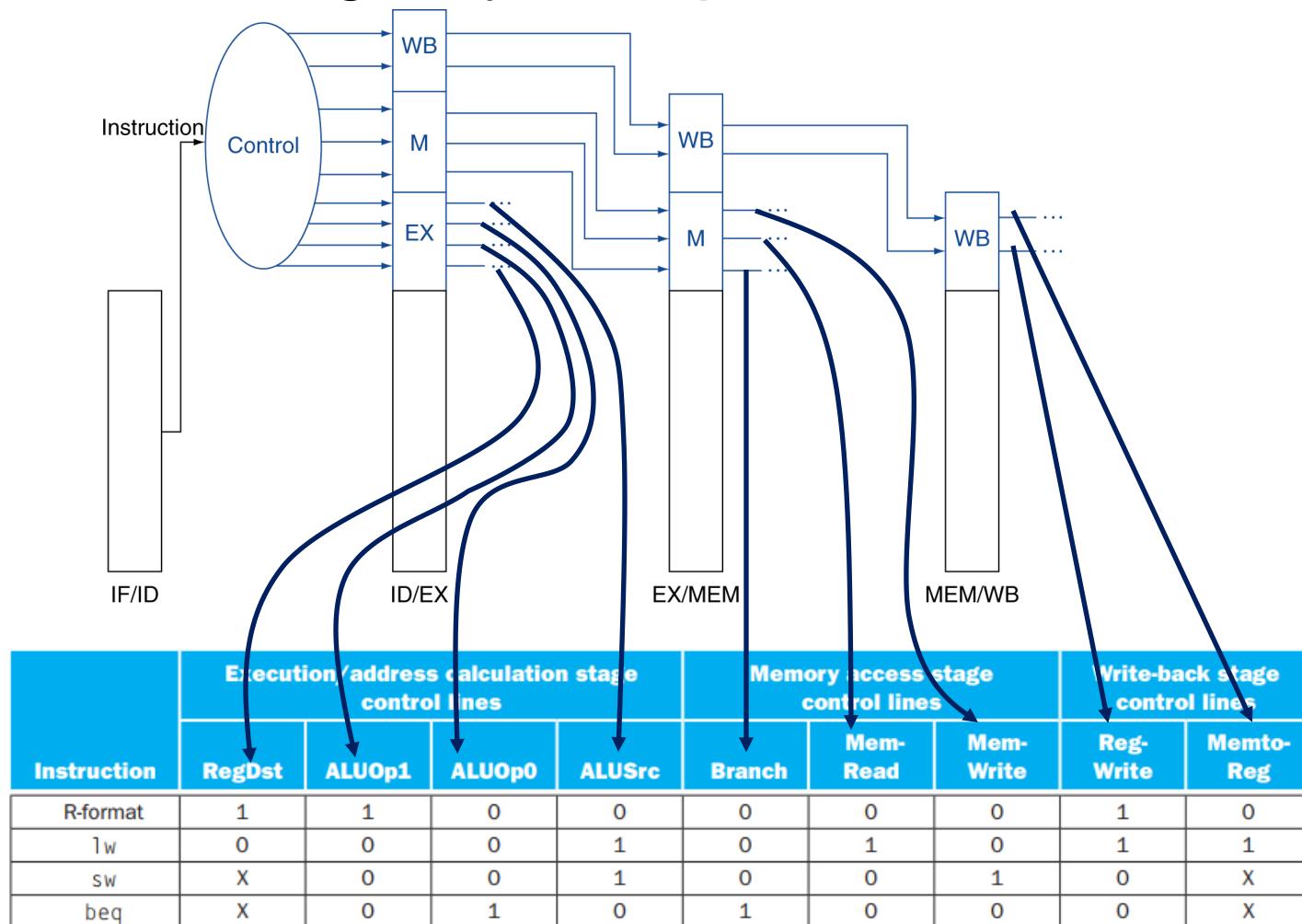


# Pipelined Control (Simplified)

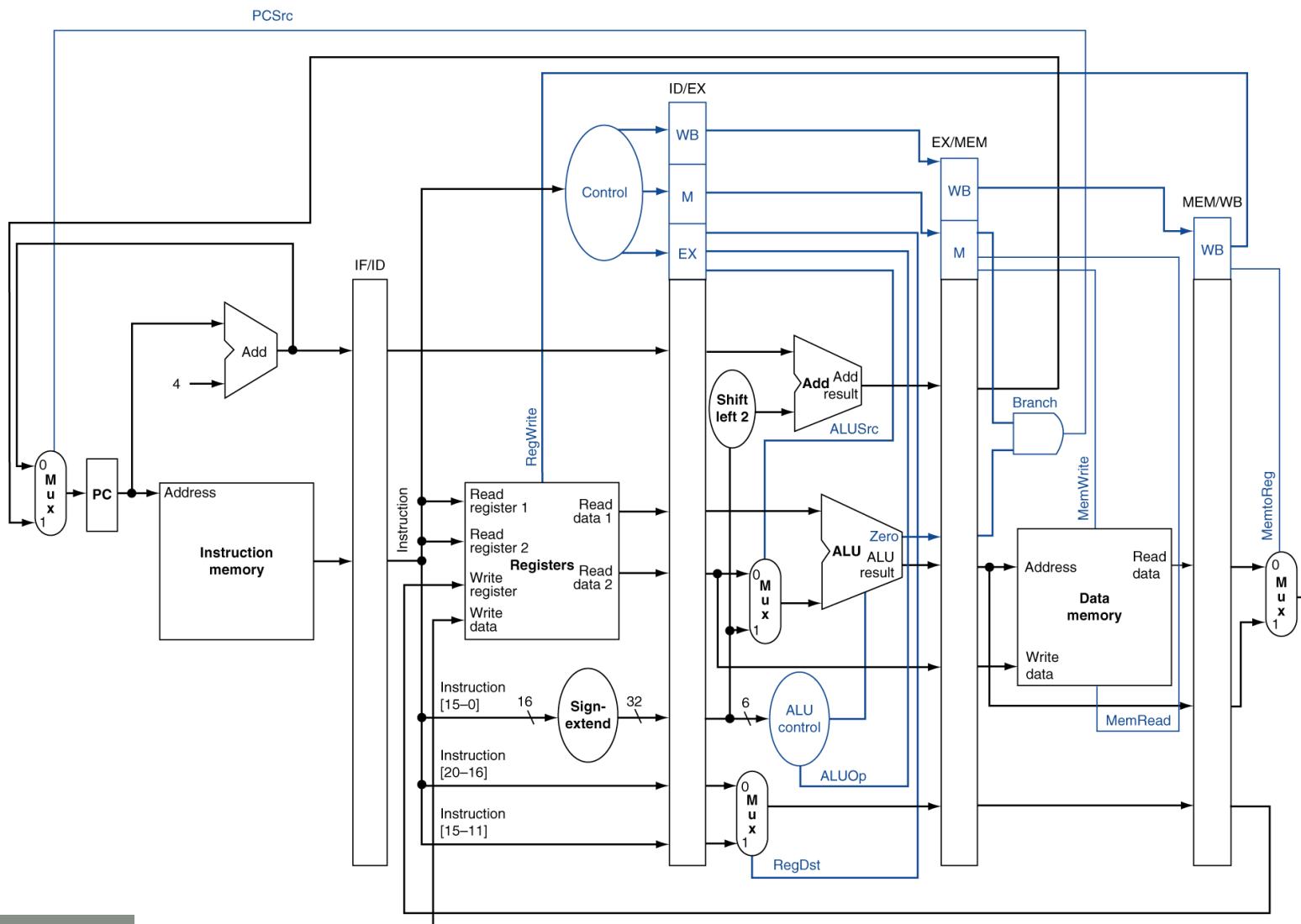


# Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



# Pipelined Control



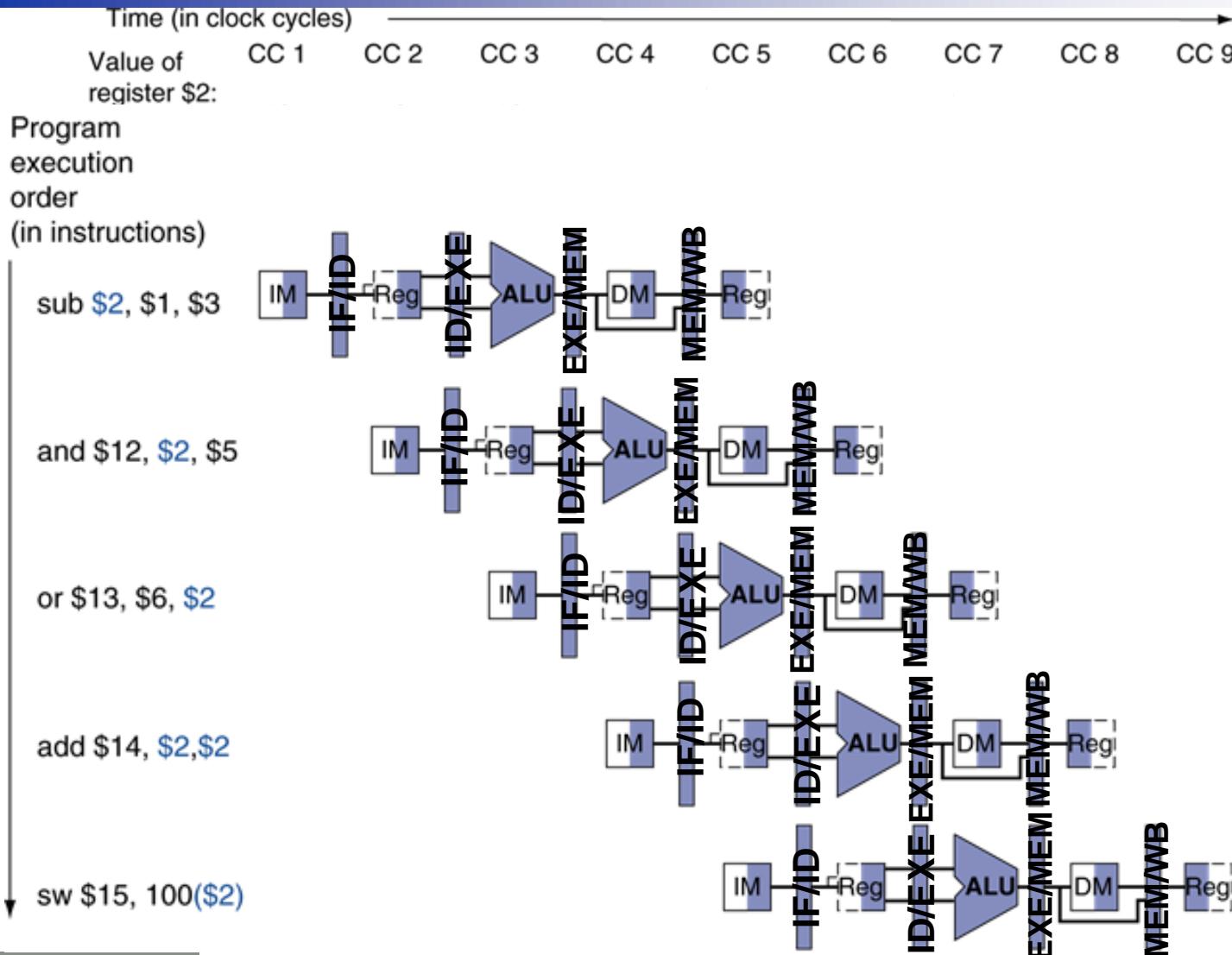
# Data Hazards in ALU Instructions

- Consider this sequence:

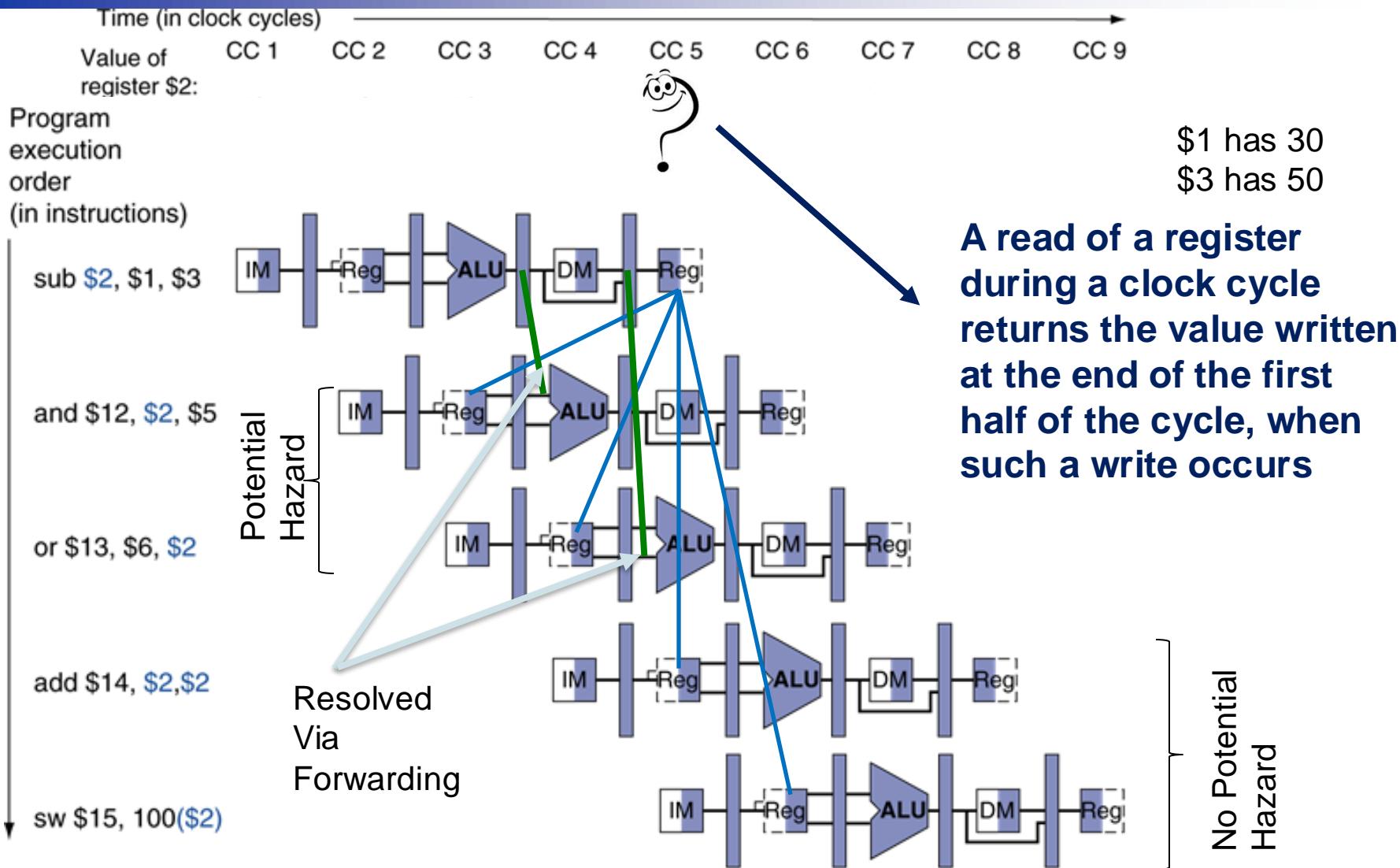
```
sub $2, $1, $3  
and $12, $2, $5  
or $13, $6, $2  
add $14, $2, $2  
sw $15, 100($2)
```

- We can resolve hazards with forwarding
  - How do we detect when to forward?

# Dependencies & Forwarding



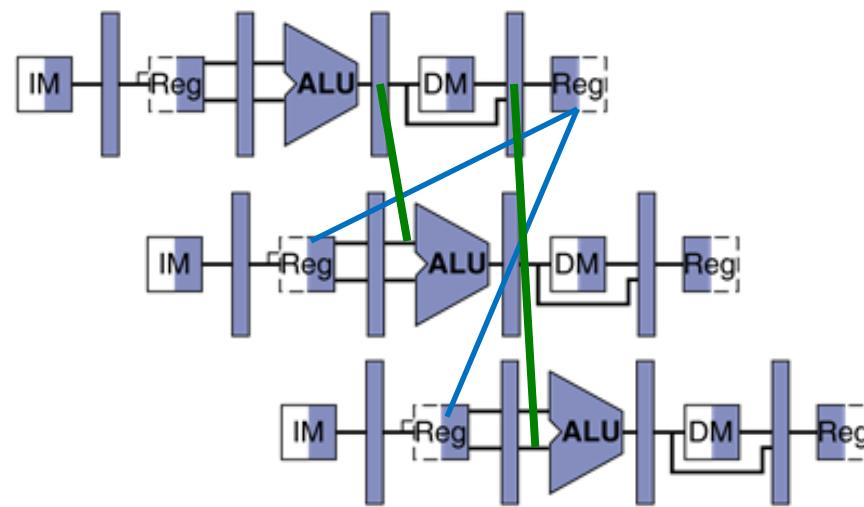
# Dependencies & Forwarding



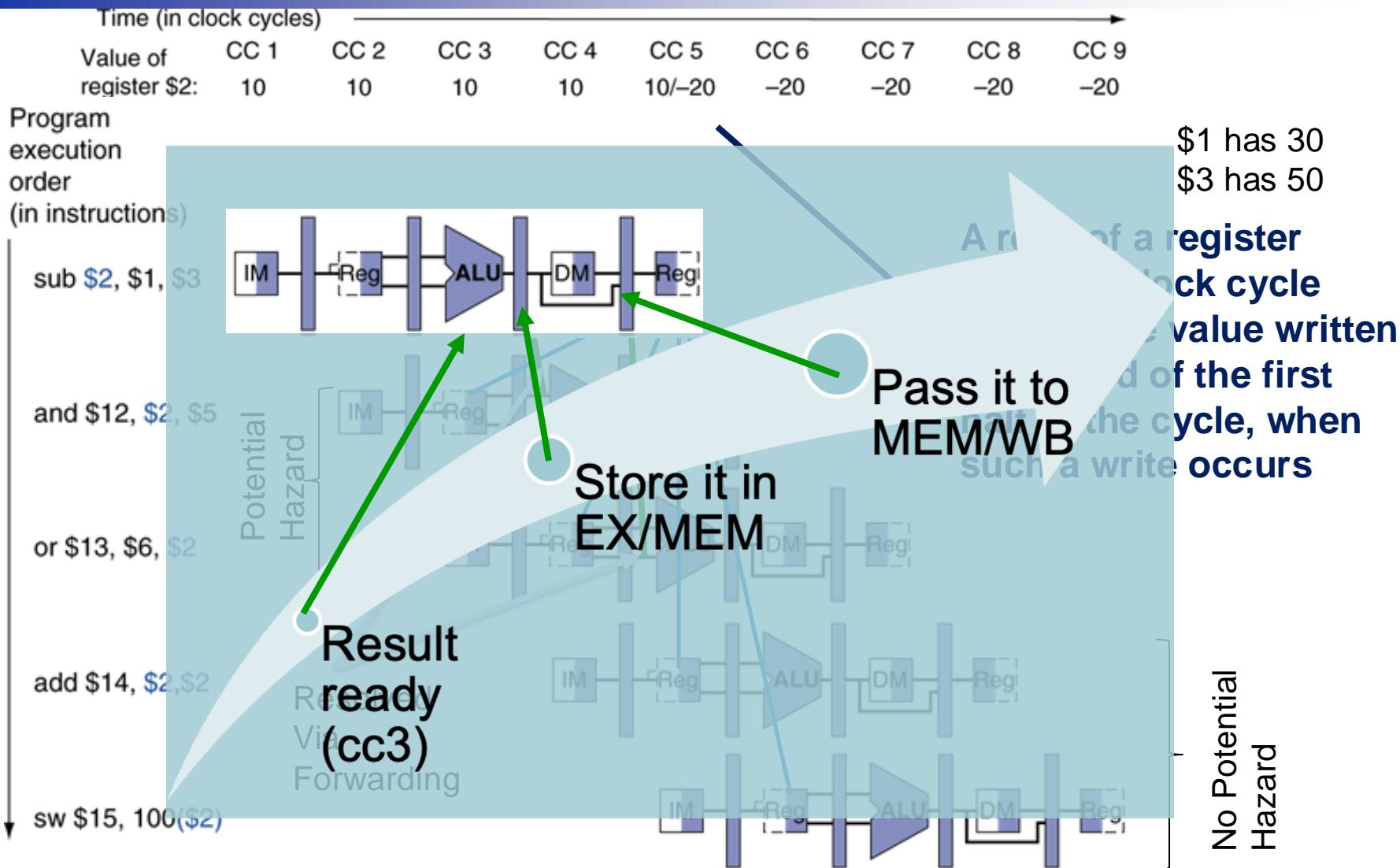
sub \$2, \$1, \$3

and \$12, \$2, \$5

or \$13, \$6, \$2



# Dependencies & Forwarding



# Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

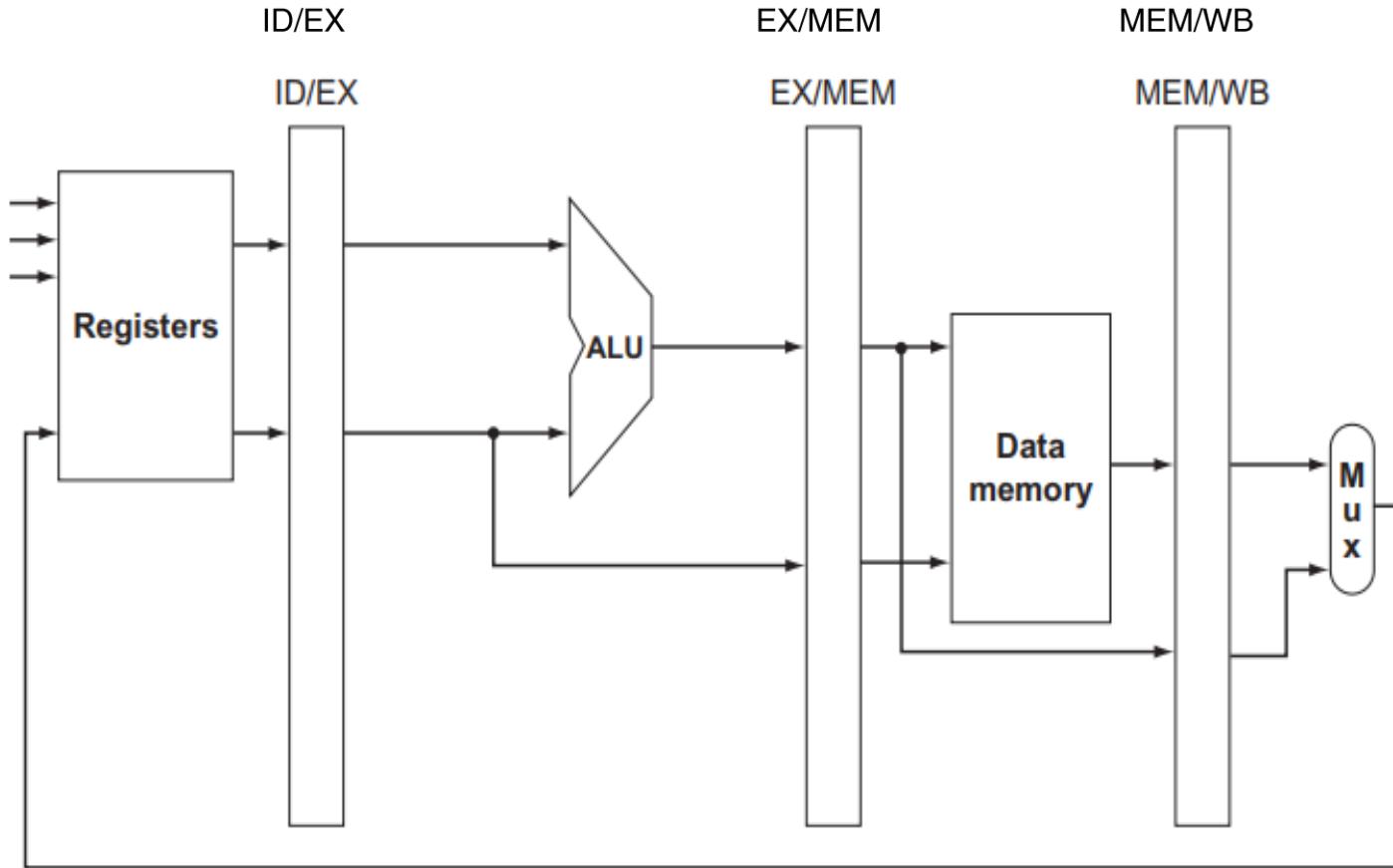
Fwd from  
EX/MEM  
pipeline reg

Fwd from  
MEM/WB  
pipeline reg

# Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not \$zero
  - EX/MEM.RegisterRd  $\neq$  0,  
MEM/WB.RegisterRd  $\neq$  0

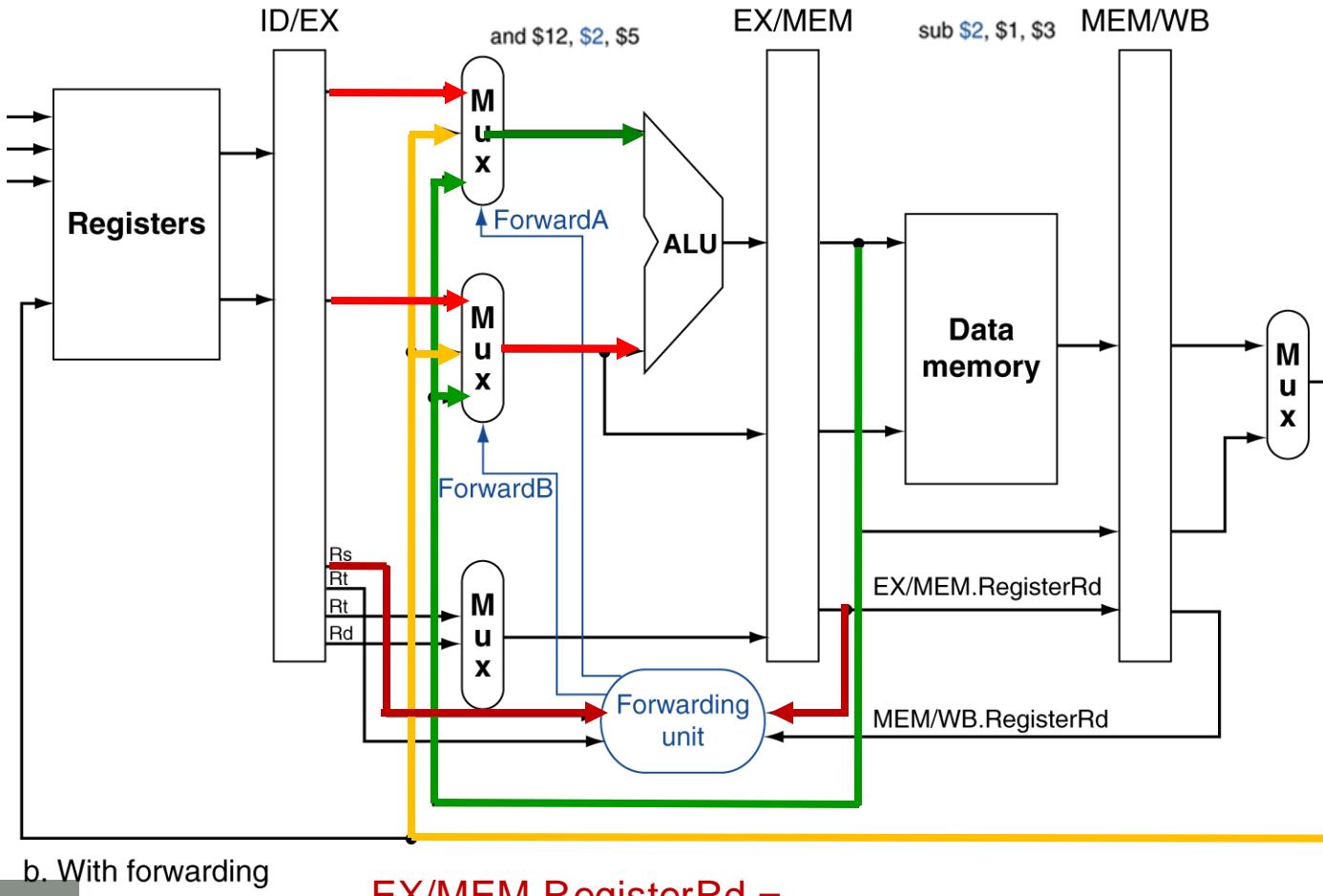
# Forwarding Paths



a. No forwarding

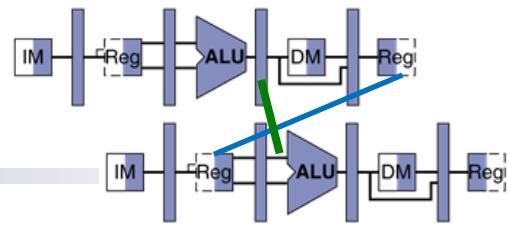
# Forwarding Paths

We have not shown regWrite and \$zero checks here.



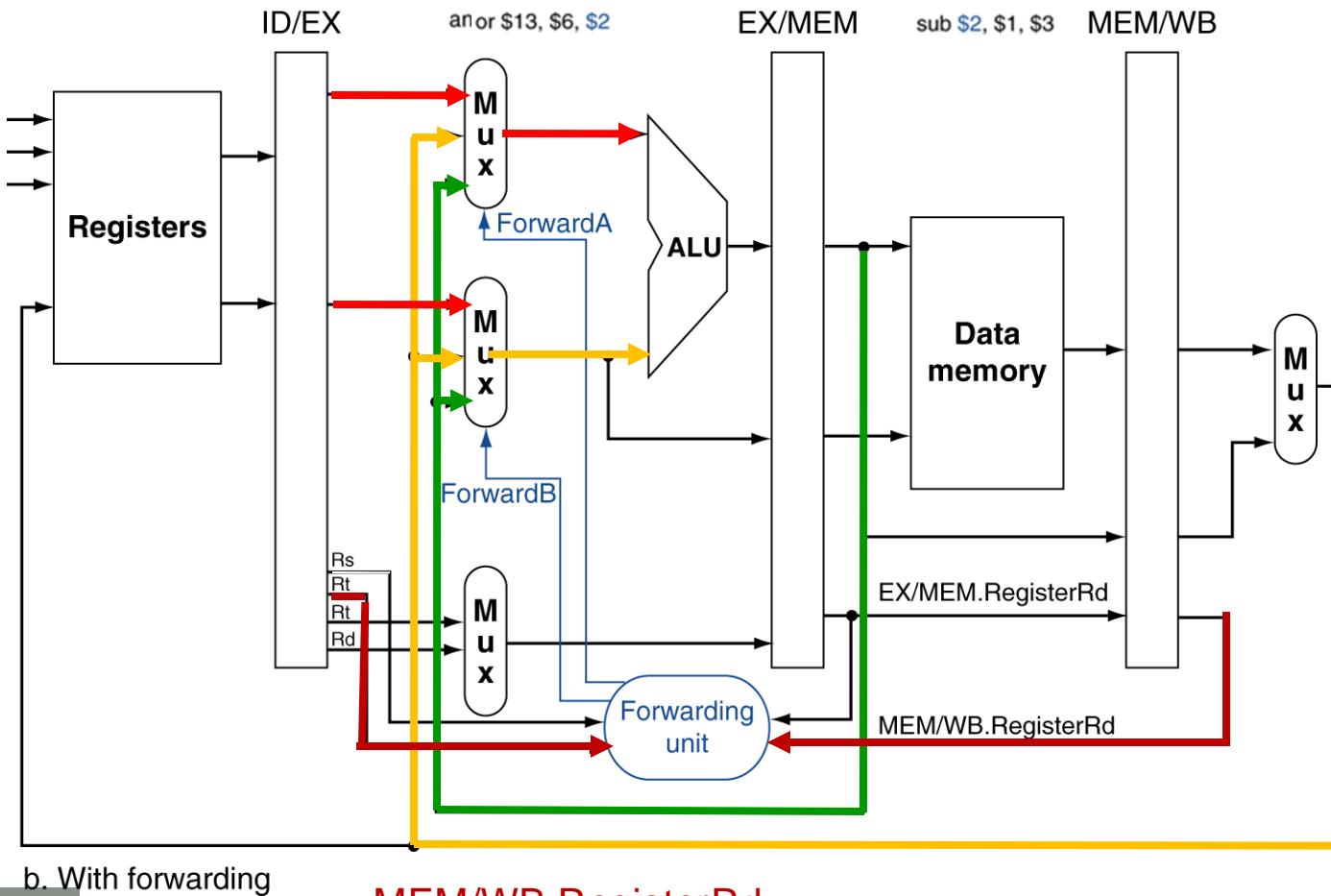
b. With forwarding

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}$$



# Forwarding Paths

We have not shown regWrite and \$zero checks here.

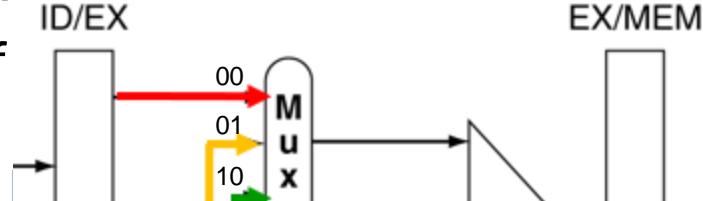


$$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}$$

# Forwarding Conditions

## EX Forward

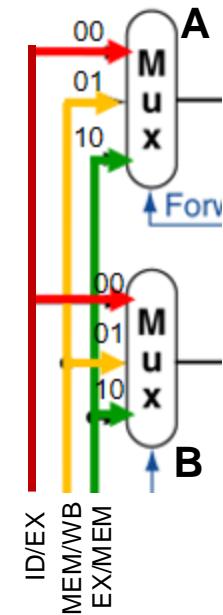
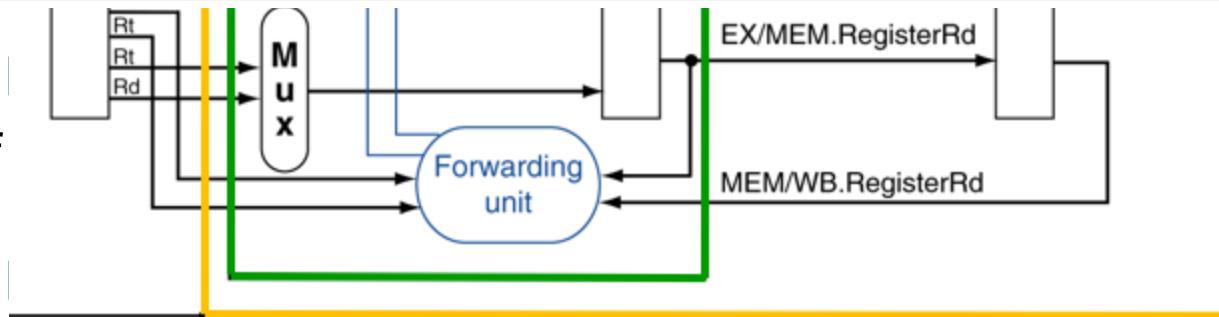
- if



### Explanation

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

- if



# Double Data Hazard

- Consider the sequence:

add \$1, \$1, \$2

add \$1, \$1, \$3

add \$1, \$1, \$4

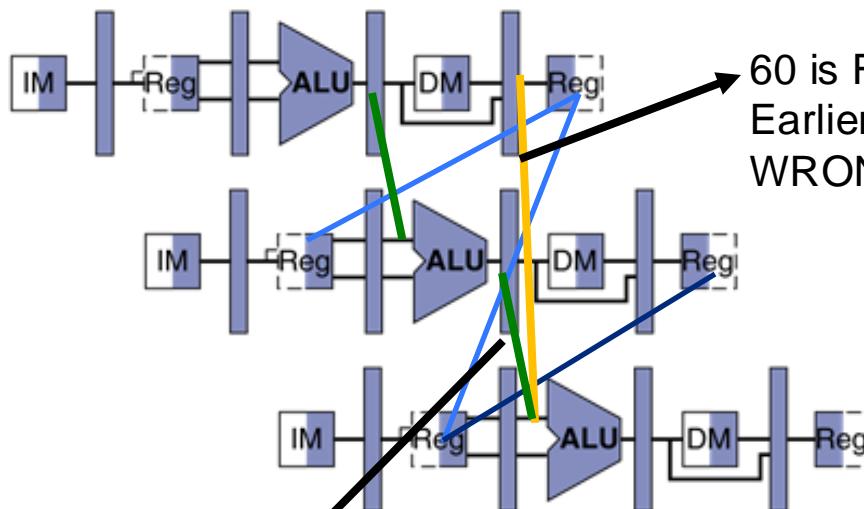
- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

# Double Data Hazard

CC:	cc1	cc2	cc3	cc4	cc5	cc6	cc7
\$2:	30	30	30	30	30	30	30
\$3:	20	20	20	20	20	20	20
\$4:	4	4	4	4	4	4	4
\$1:	10	10	10	10	10/40		

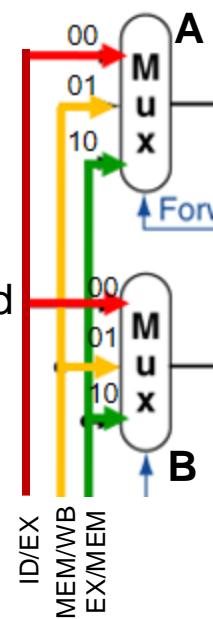
What is expected?

add \$1,\$1,\$2



add \$1,\$1,\$3

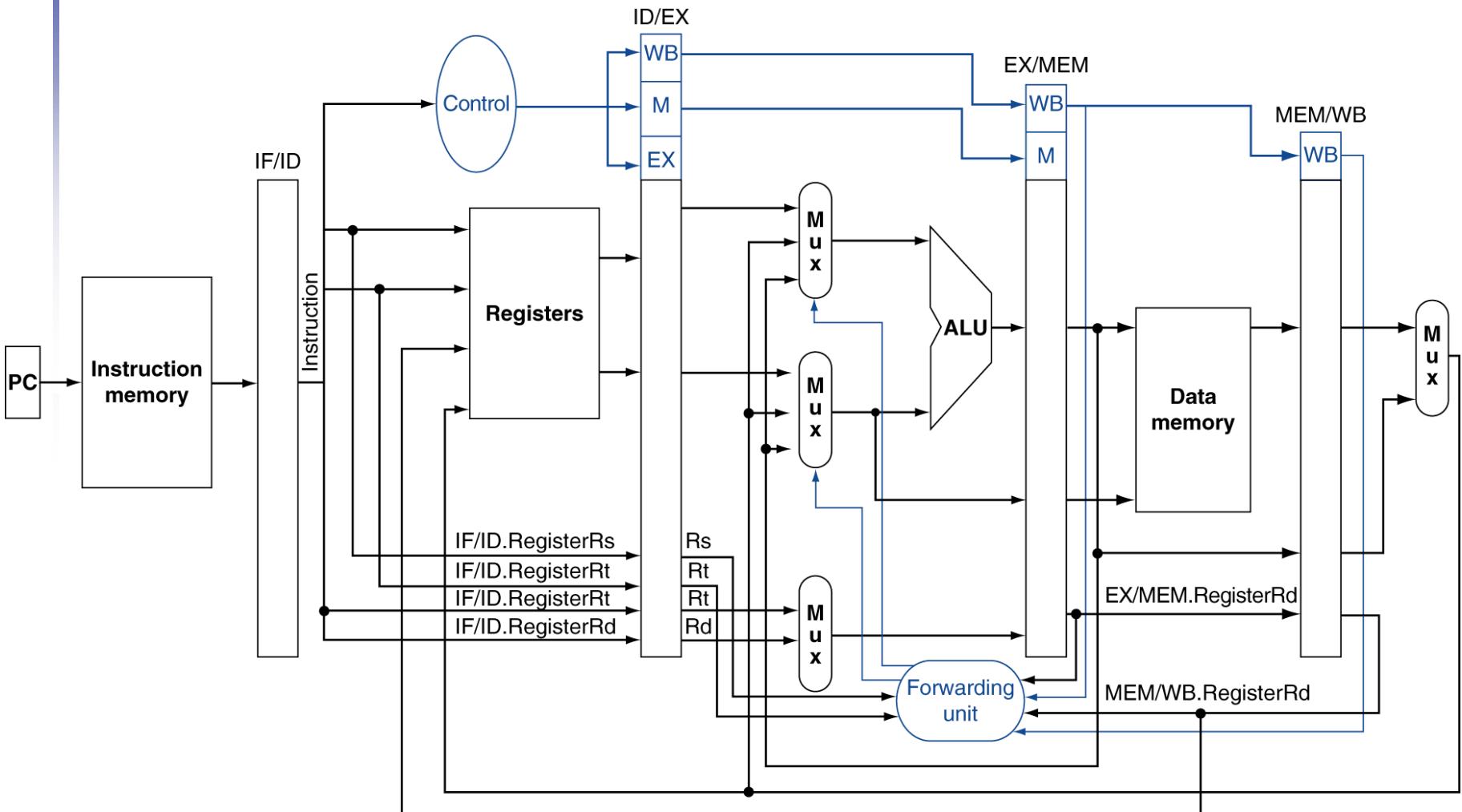
add \$1,\$1,\$4



# Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
ForwardB = 01

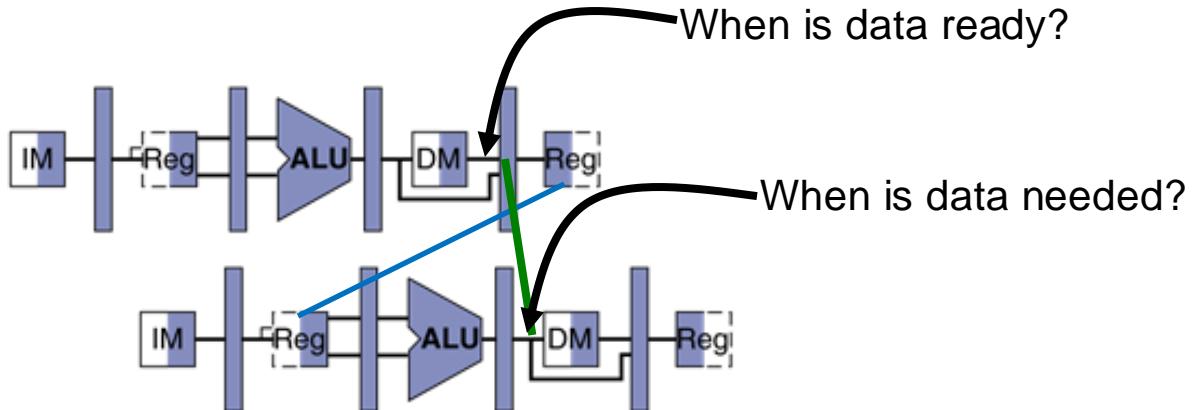
# Datapath with Forwarding



# Load -> Store (Mem to Mem Copy)

register & memory connection  $\Rightarrow$  ALU (2nd  $\Rightarrow$  1st)

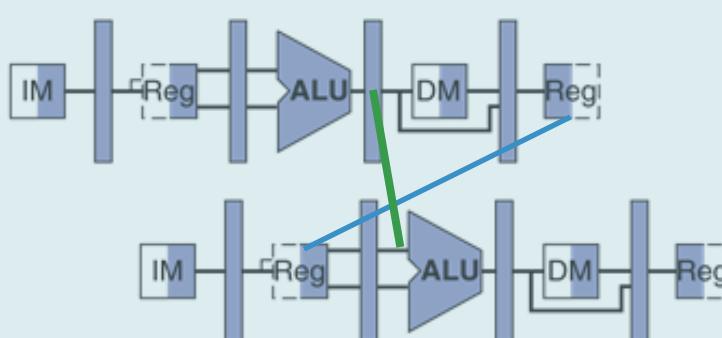
lw \$2, 20(\$1)



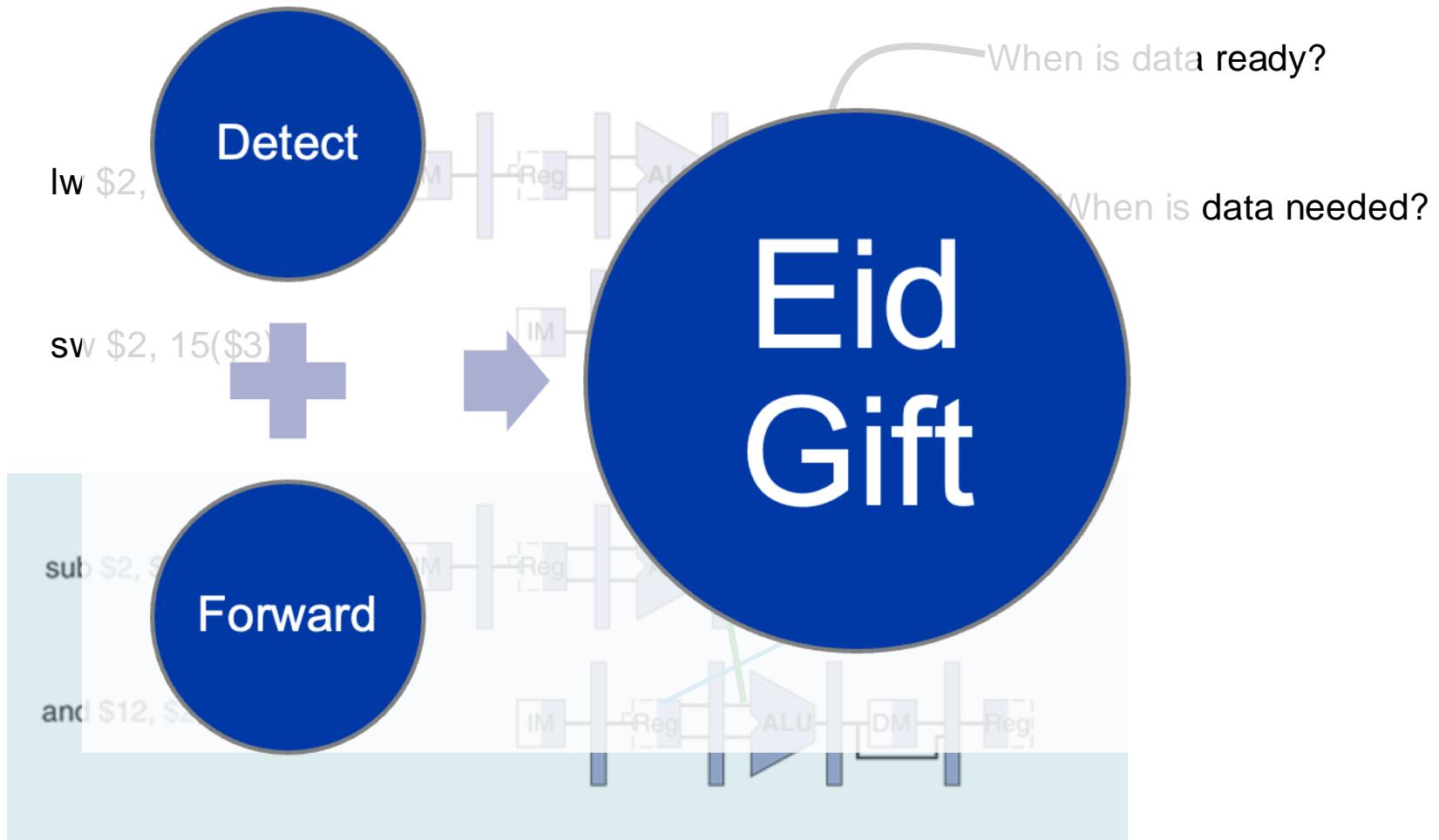
sw \$2, 15(\$3)

sub \$2, \$1, \$3

and \$12, \$2, \$5



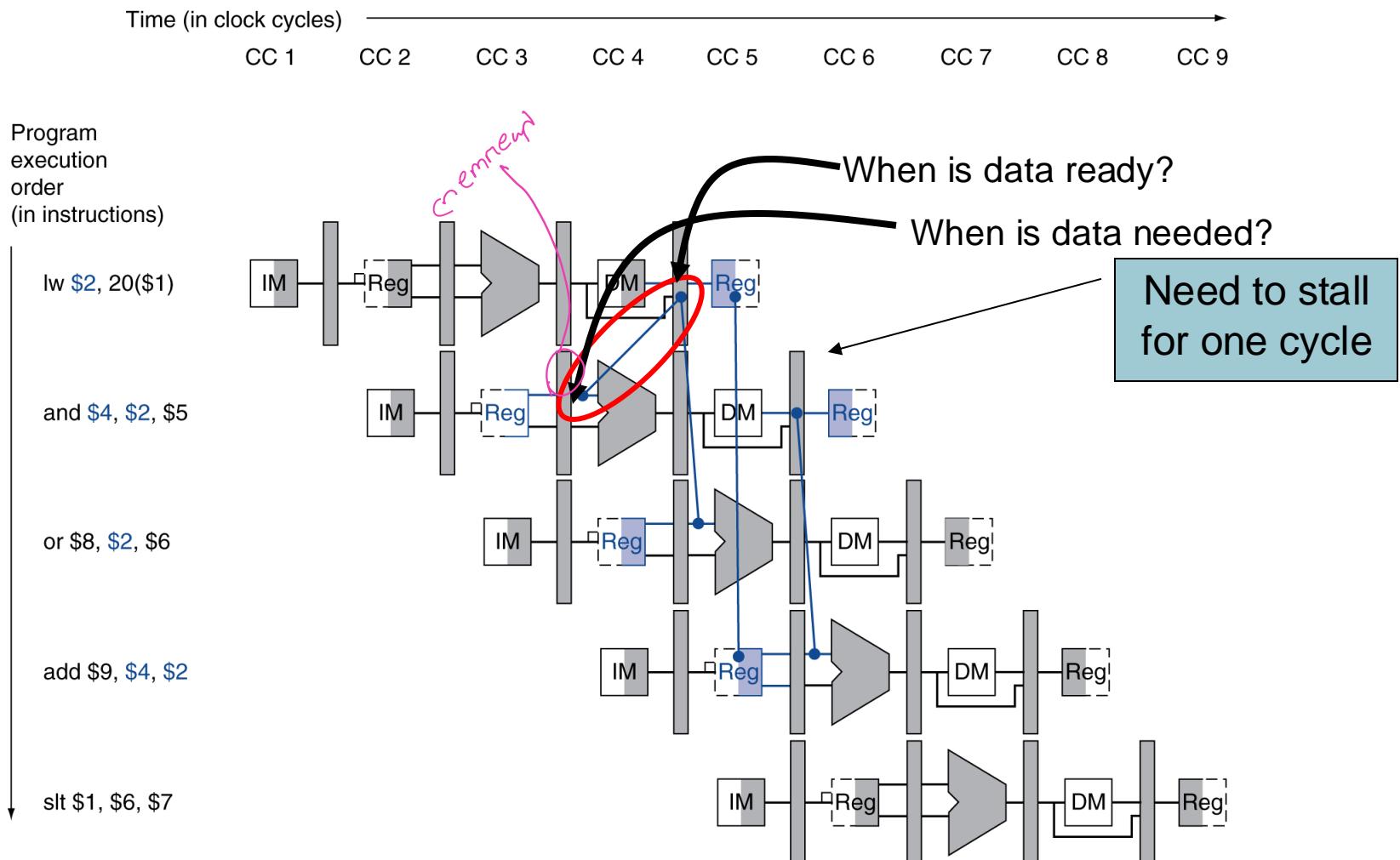
# Load -> Store (Mem to Mem Copy)



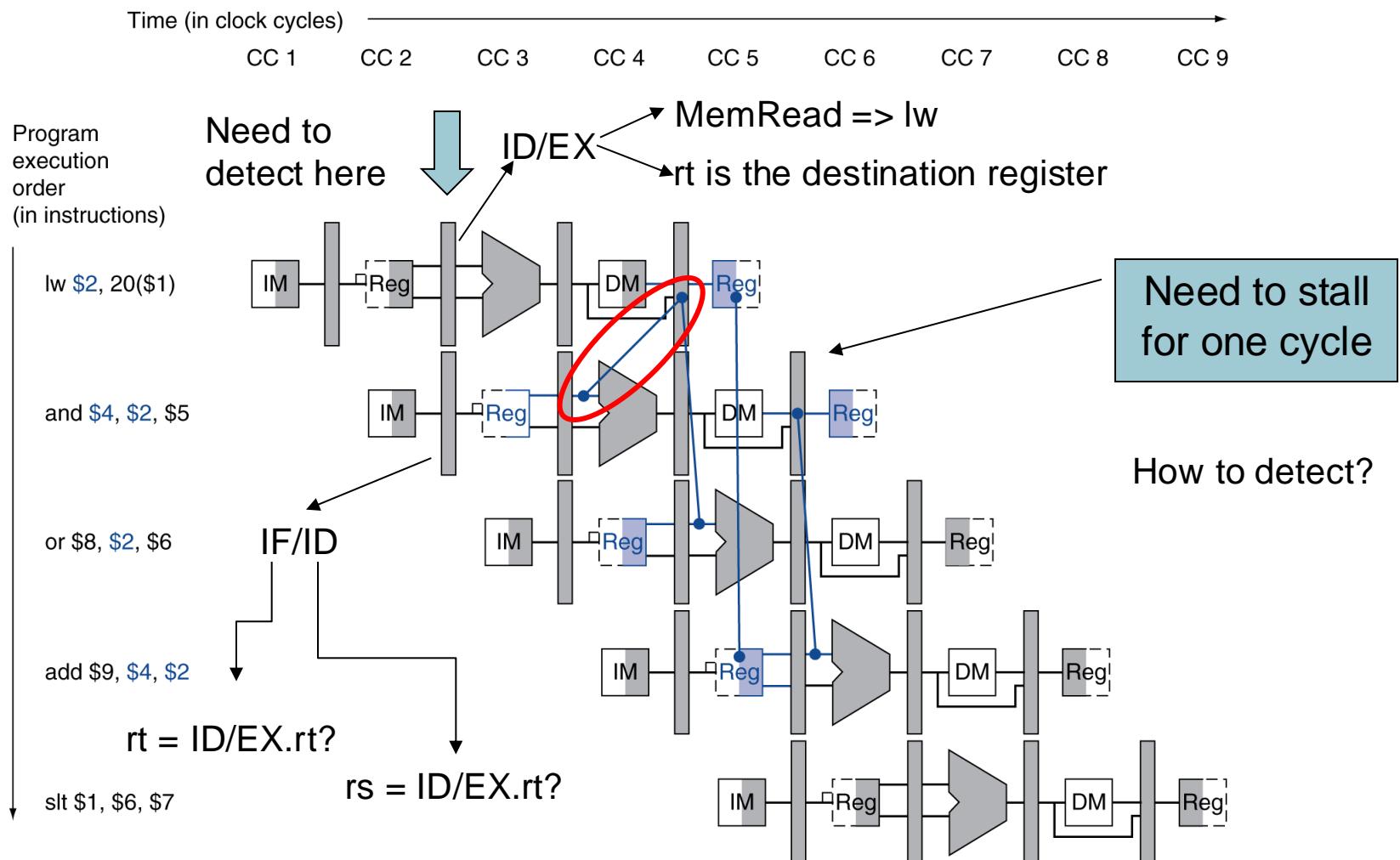
# Load-Use Data Hazard

memread

prev use ref ==  
current use ref or read  
| = instruction load



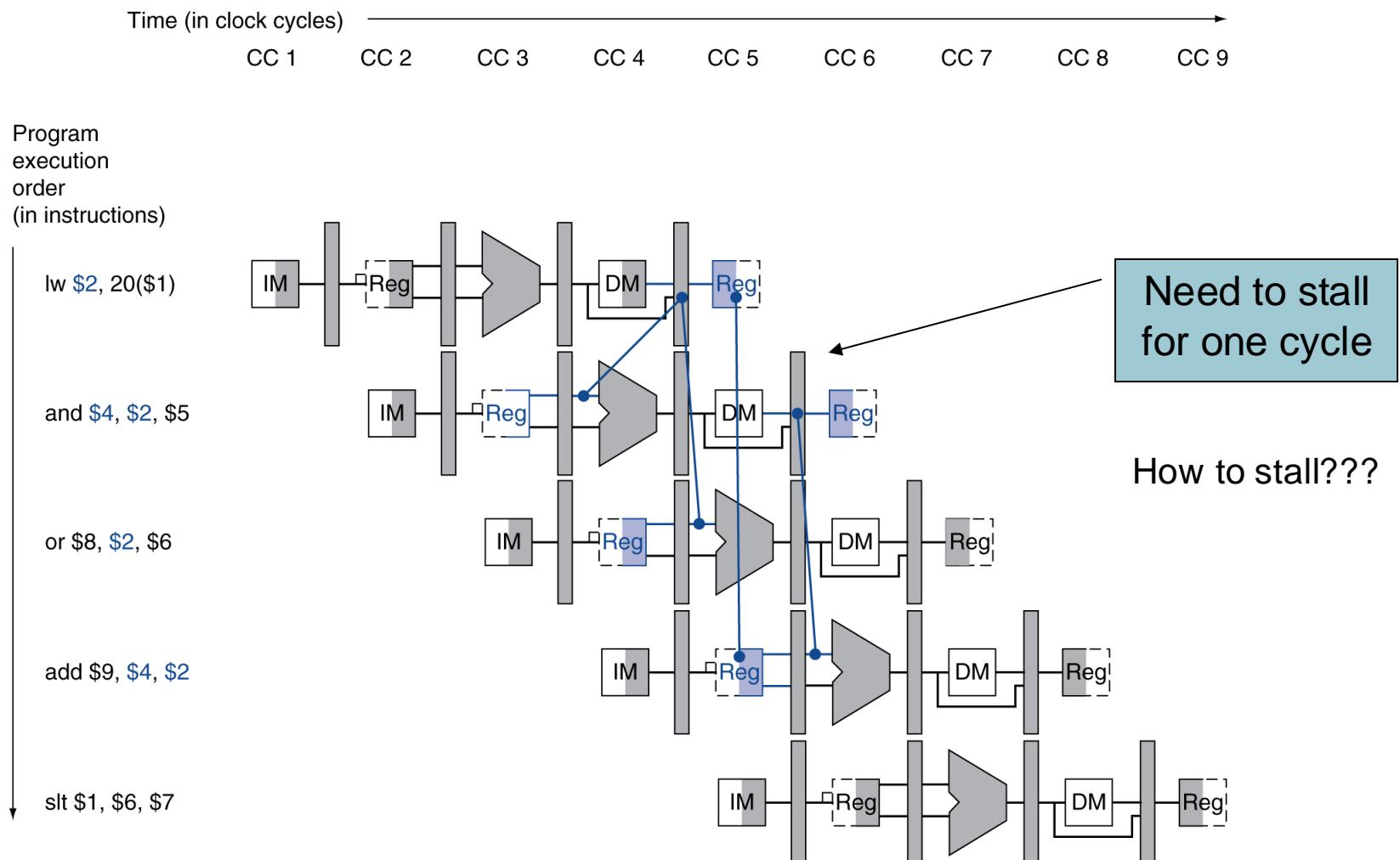
# Load-Use Data Hazard



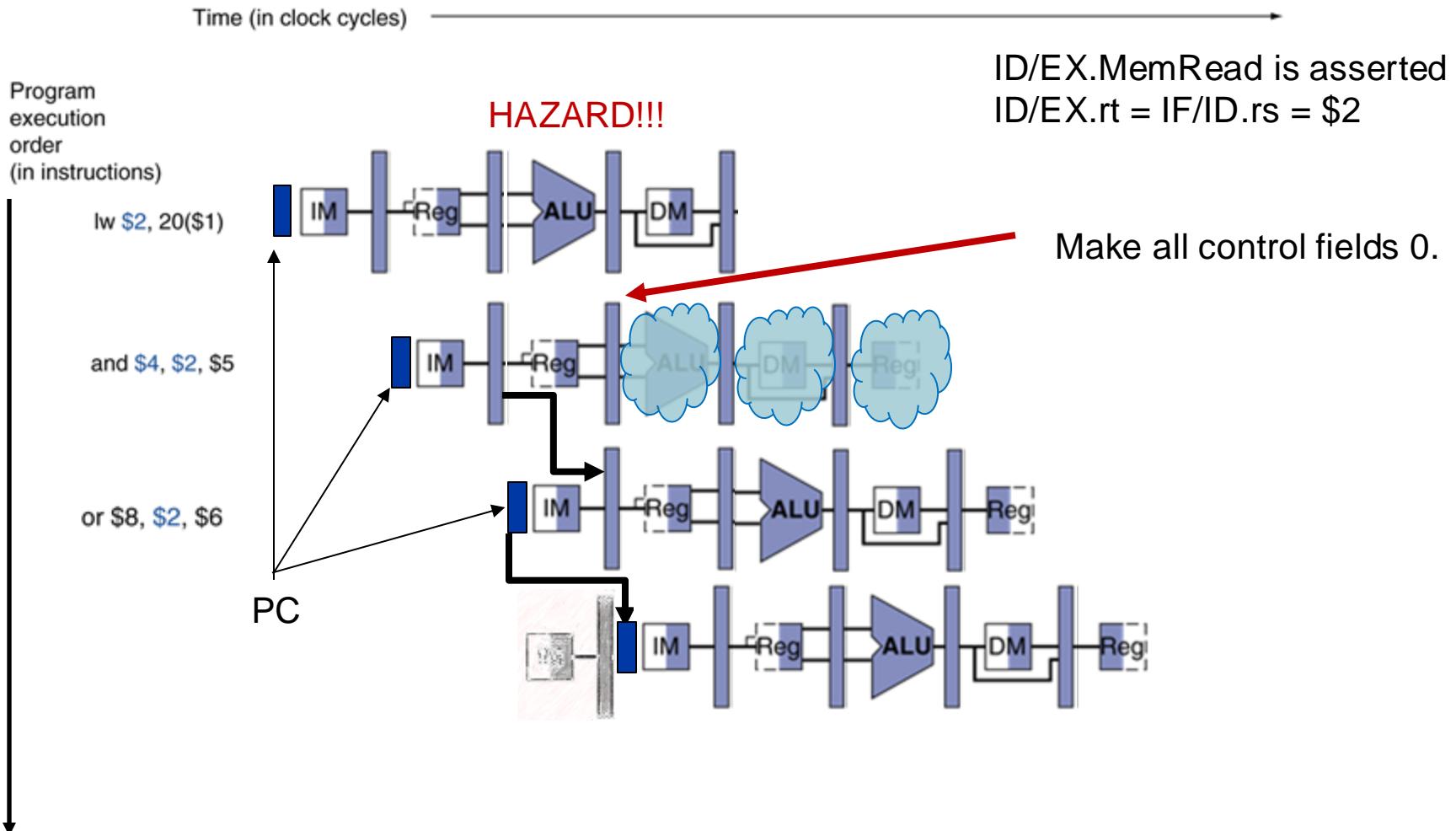
# Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
$$((ID/EX.RegisterRt = IF/ID.RegisterRs) \text{ or } (ID/EX.RegisterRt = IF/ID.RegisterRt))$$
- If detected, stall and insert bubble

# Load-Use Data Hazard



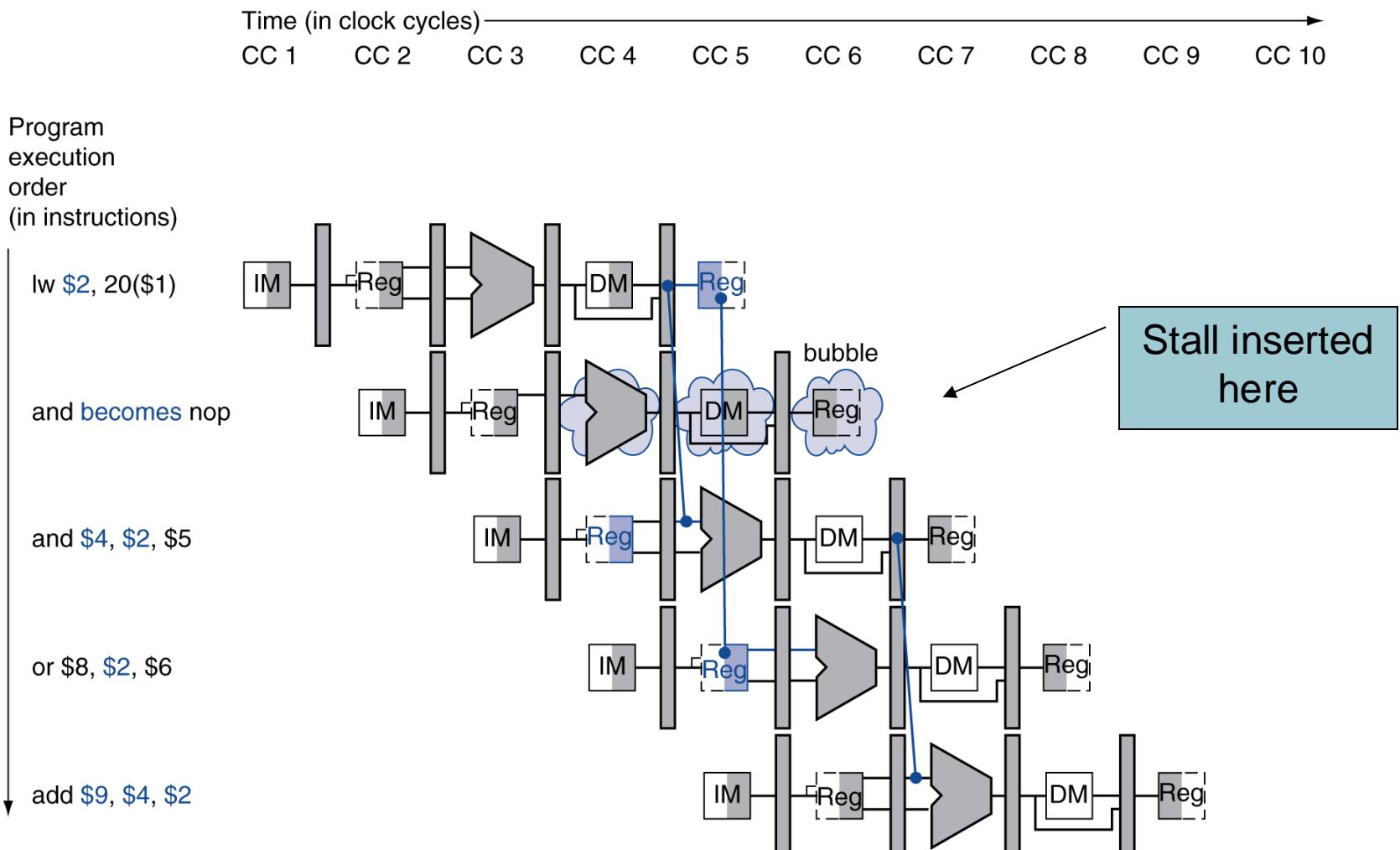
# How to Stall



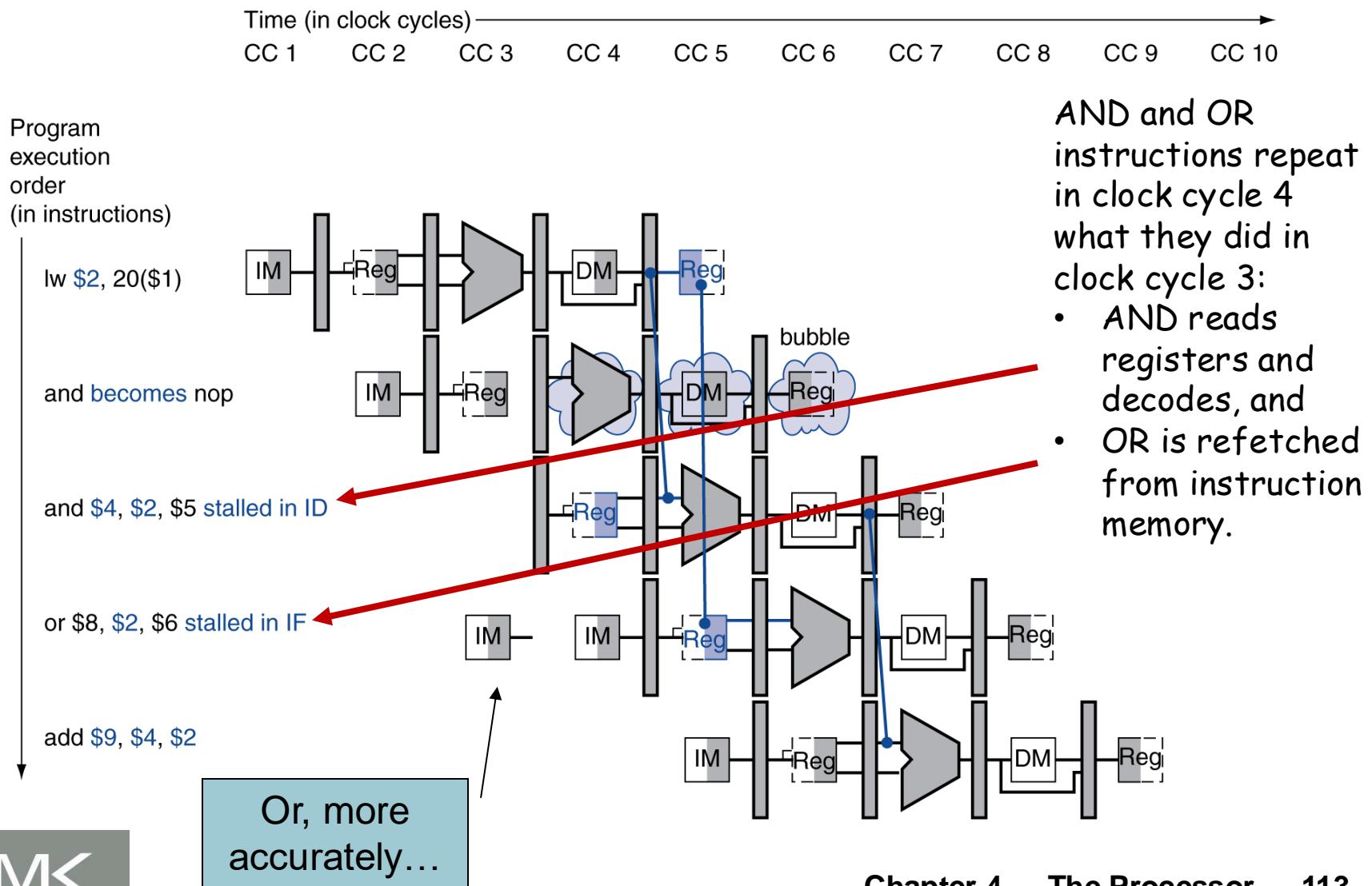
# How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for 1w
    - Can subsequently forward to EX stage

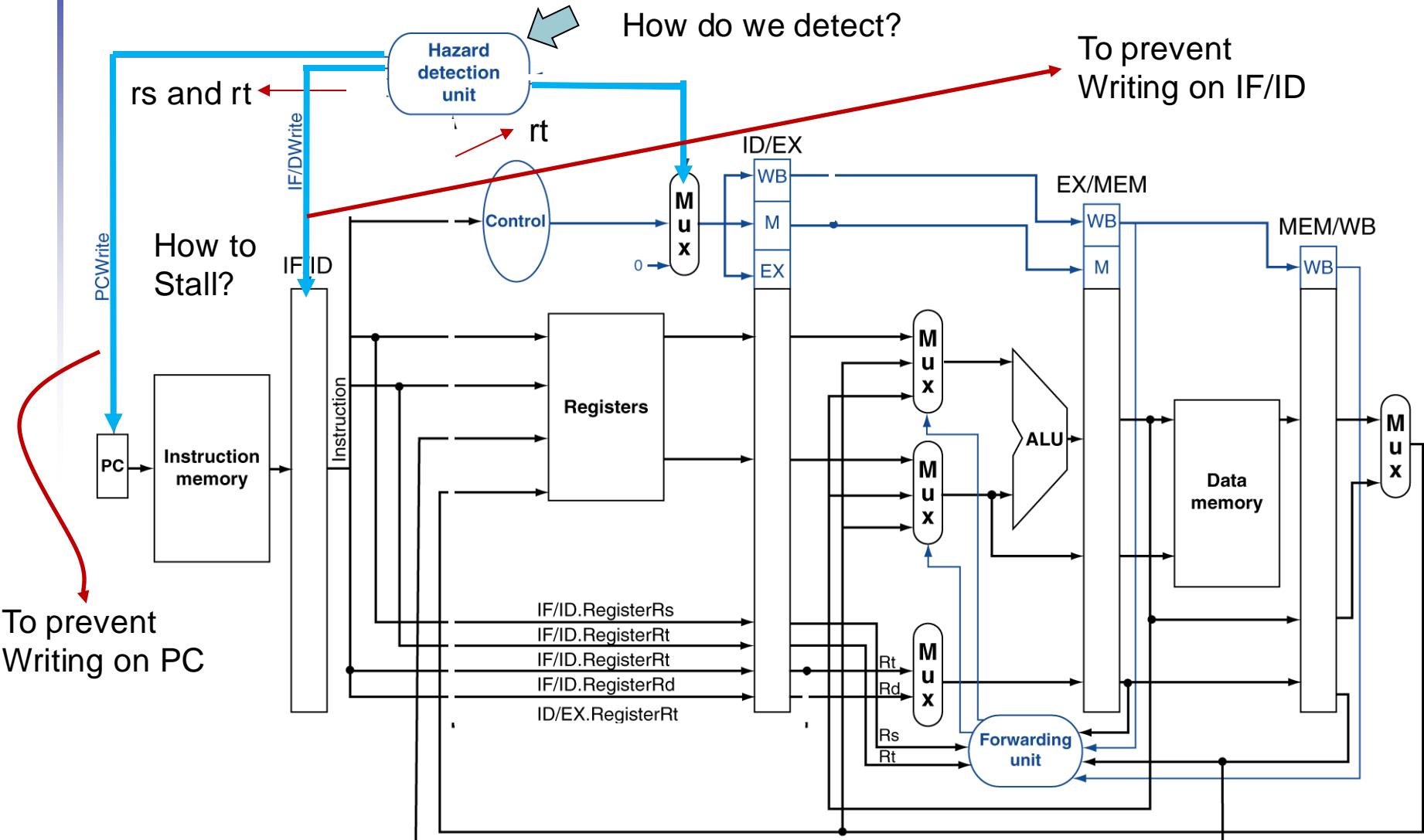
# Stall/Bubble in the Pipeline



# Stall/Bubble in the Pipeline



# Datapath with Hazard Detection



# Stalls and Performance

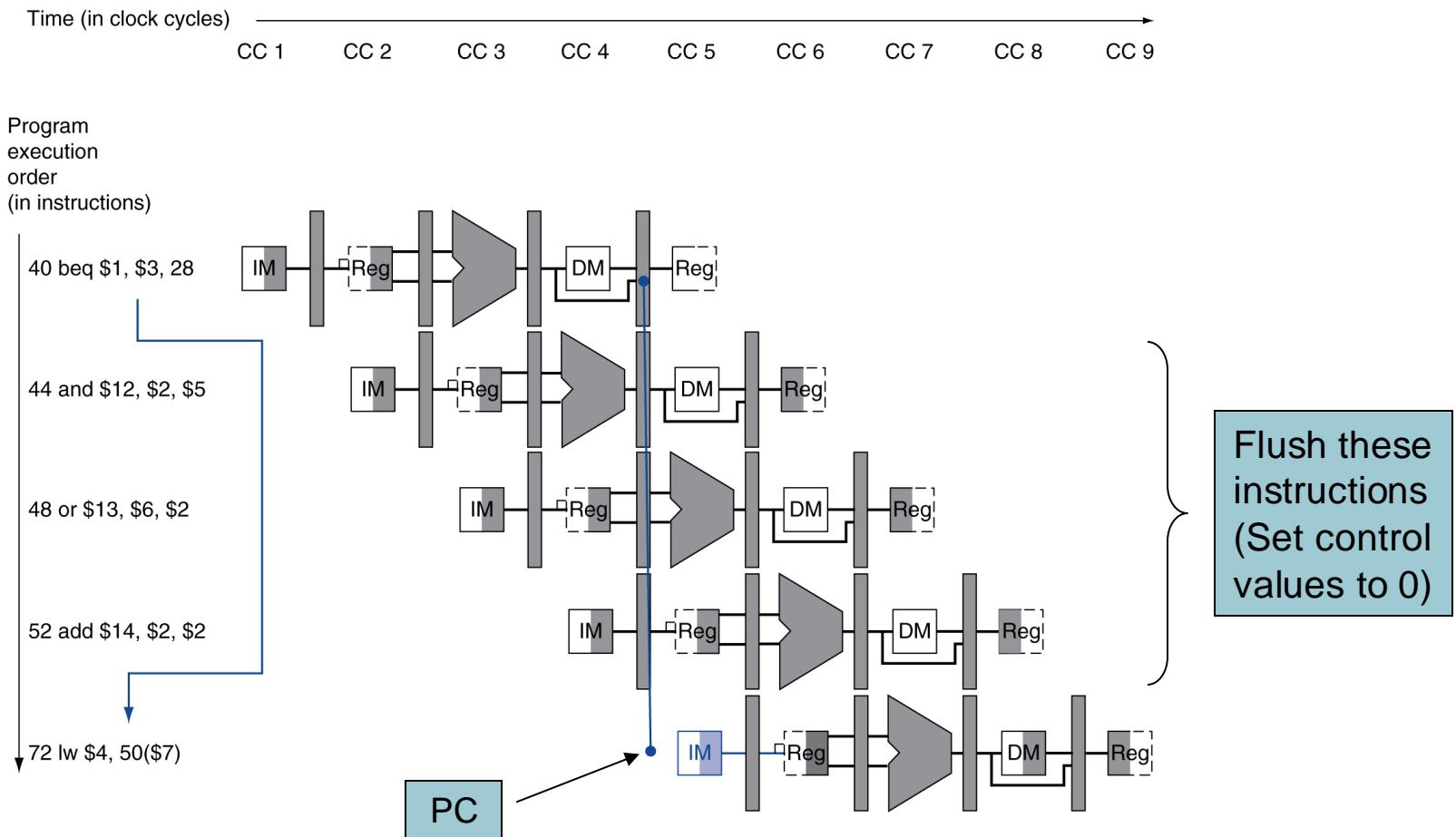
## The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

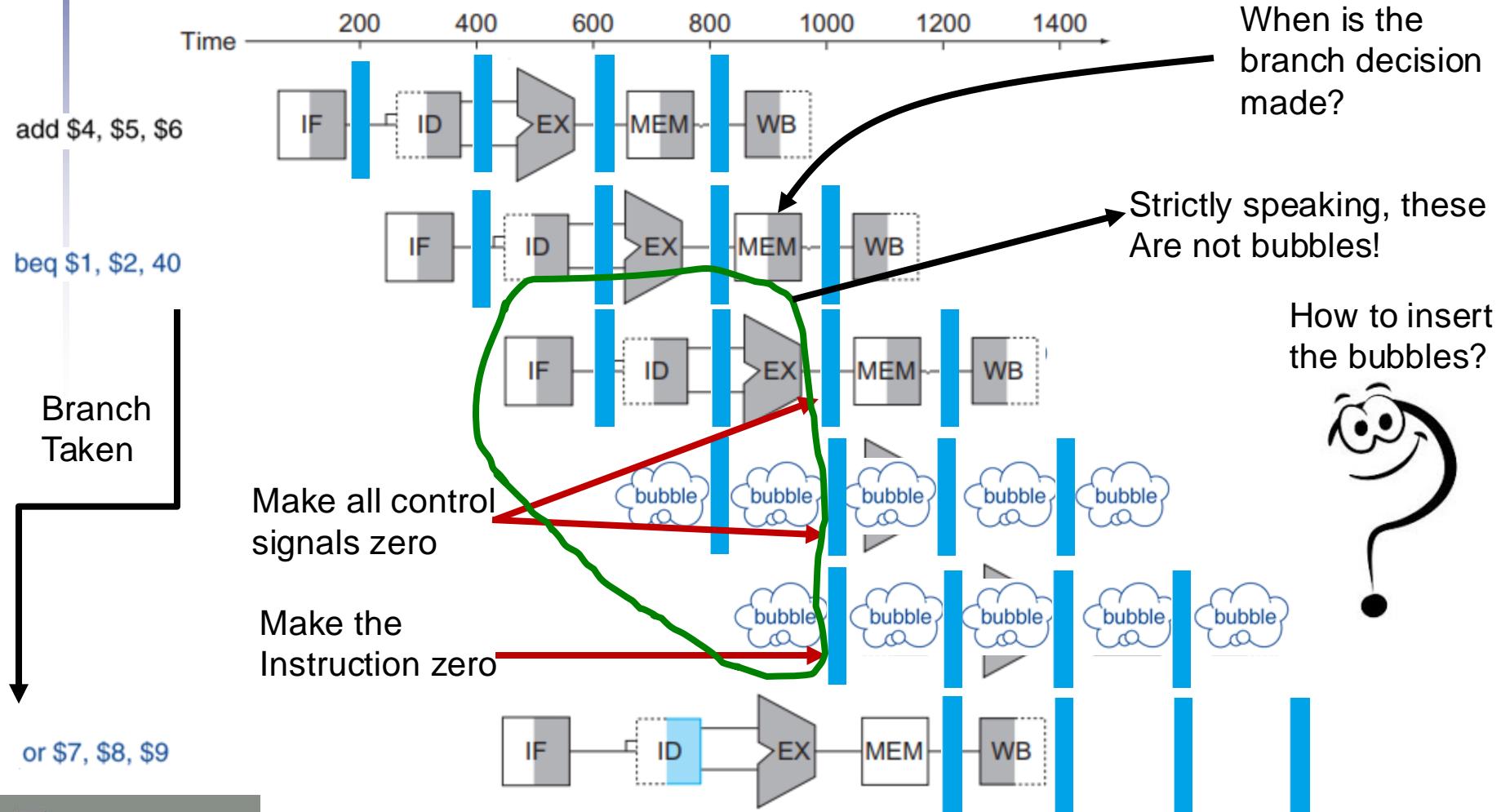
CT  
S2  
W2

# Branch Hazards

- If branch outcome determined in MEM



# Branch Scenario (RECALLING)



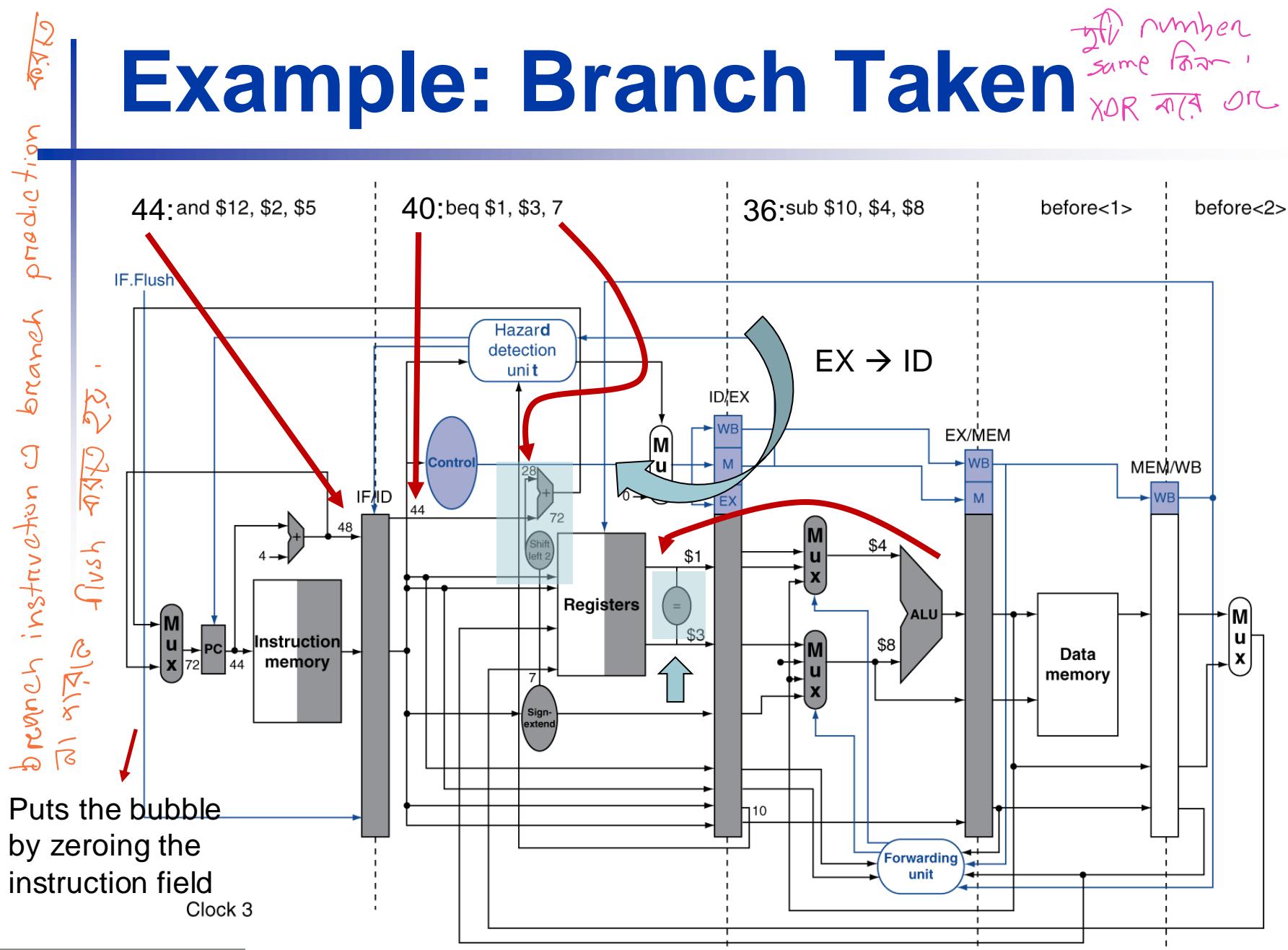
# Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

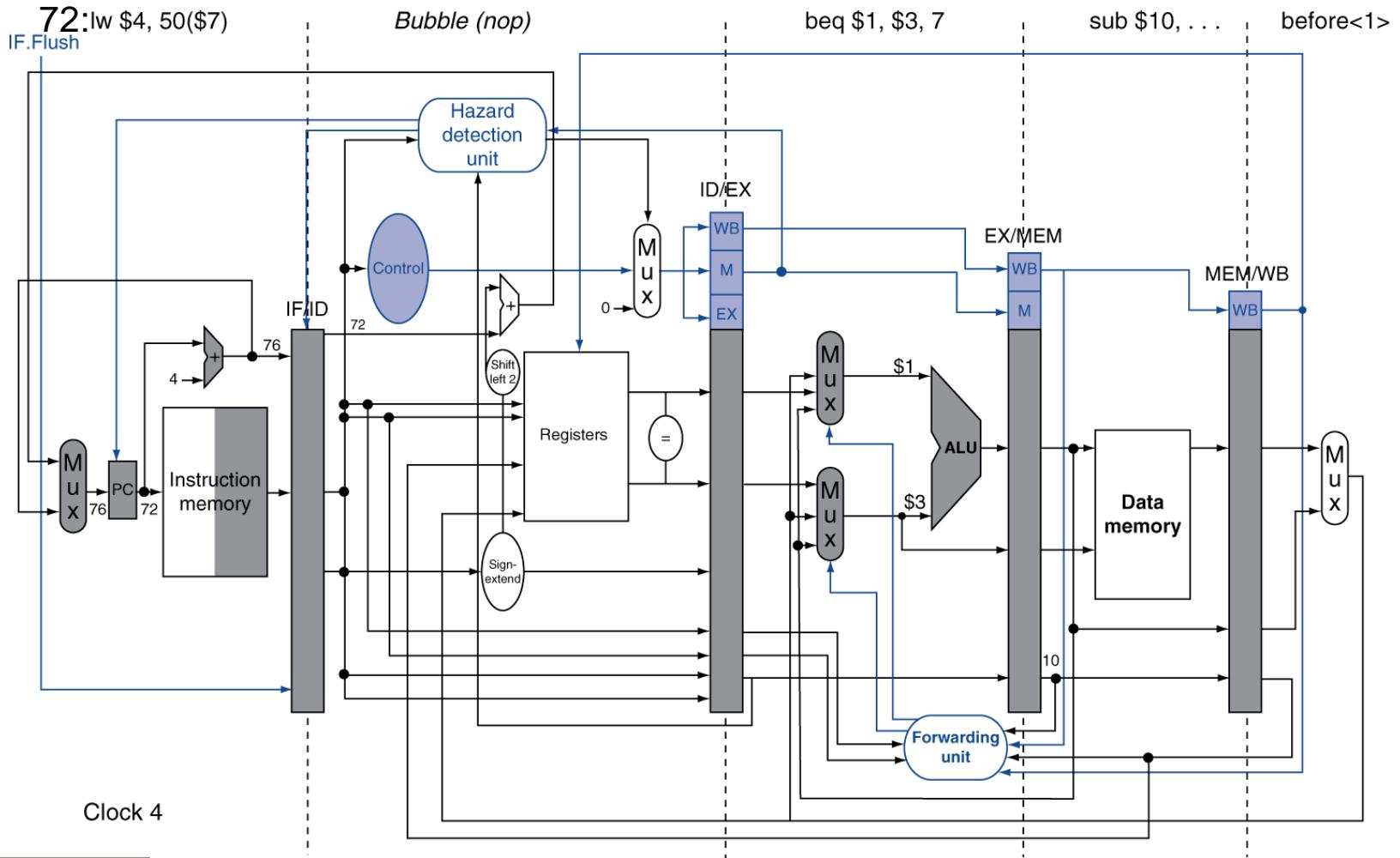
```
36:    sub   $10, $4, $8  
40:    beq   $1,  $3,  7  
44:    and   $12, $2, $5  
48:    or    $13, $2, $6  
52:    add   $14, $4, $2  
56:    slt   $15, $6, $7  
      ...  
72:    lw    $4, 50($7)
```

# Example: Branch Taken

number  
same from:  
XOR করে OR

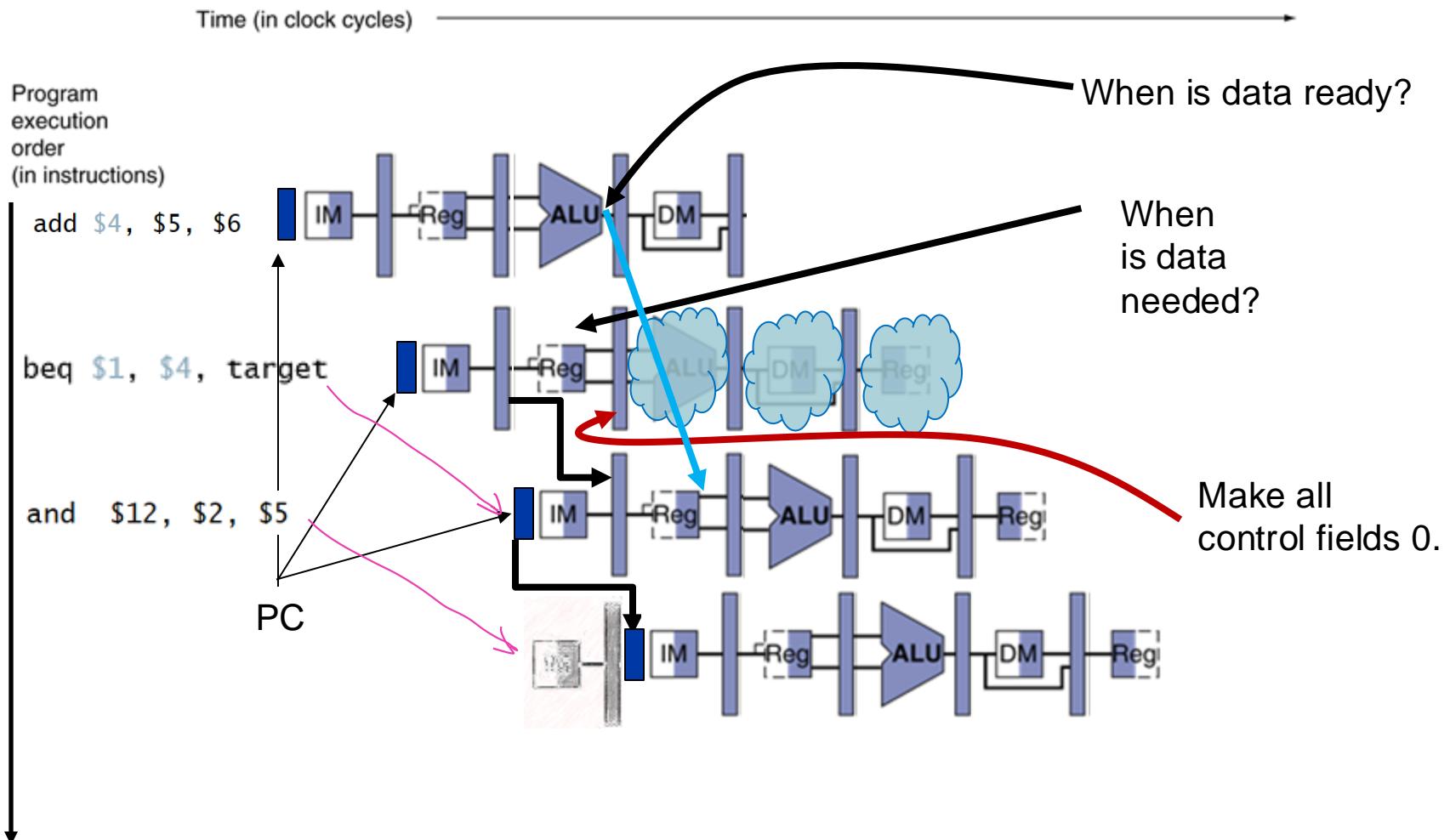


# Example: Branch Taken



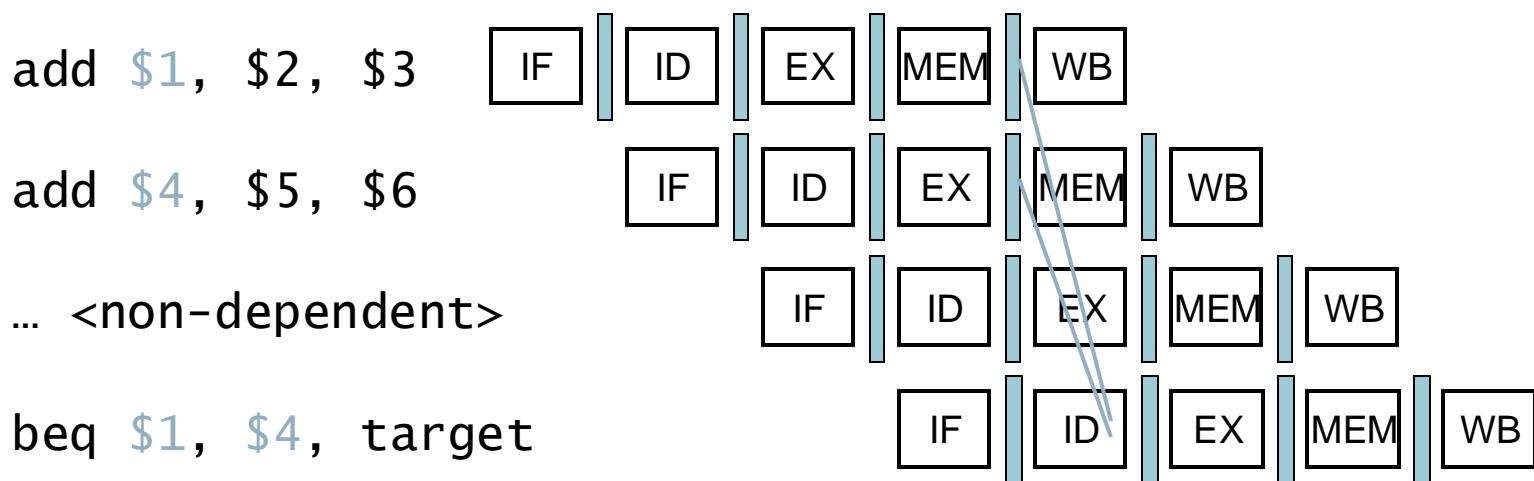
Clock 4

# Data Hazards for Branches



# Data Hazards for Branches

- If a comparison register is a destination of 2<sup>nd</sup> or 3<sup>rd</sup> preceding ALU instruction



- Can resolve using forwarding

# Data Hazards for Branches

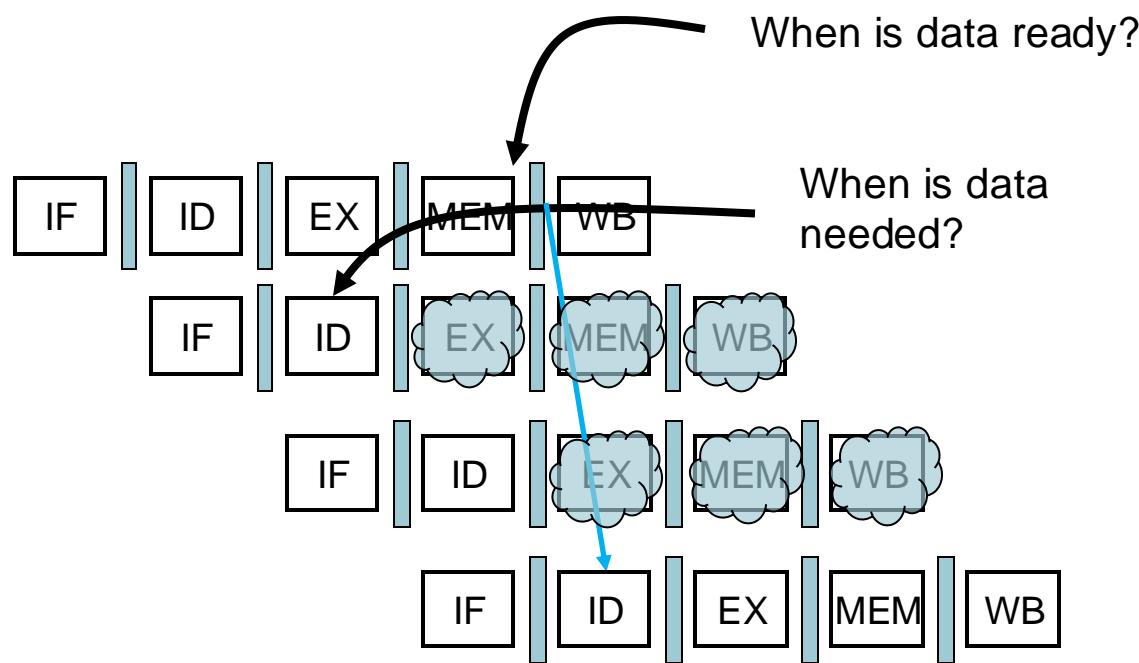
lw \$1, addr

beq \$ta\$1\$ed, target

beq stalled

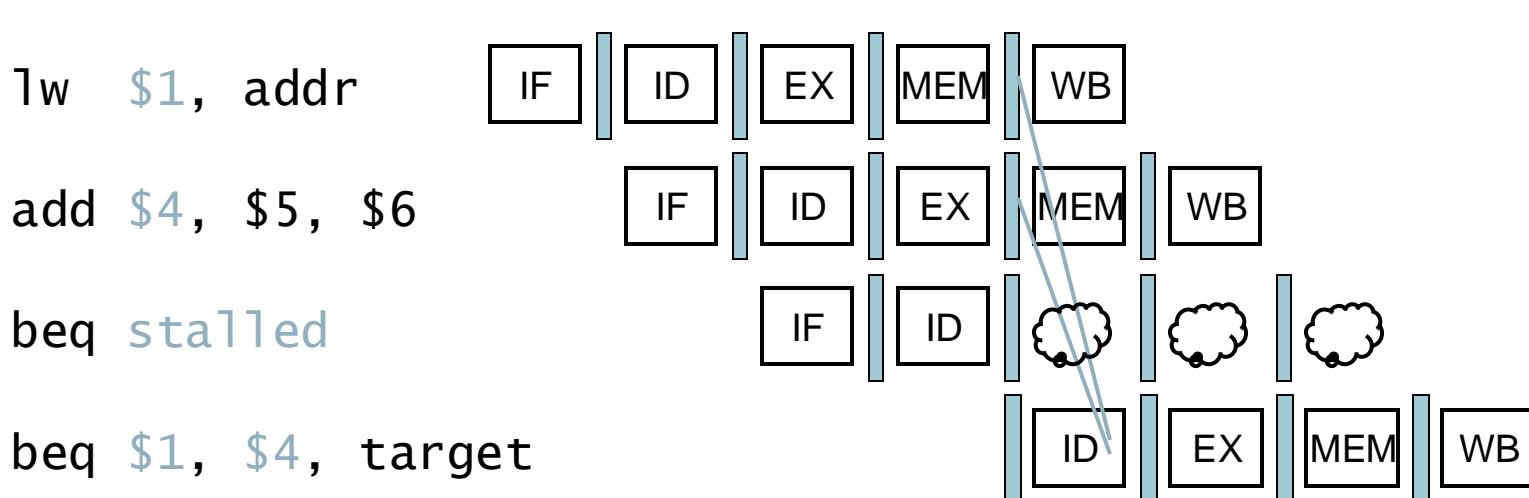
IF.Flush

All controls  
made zero



# Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2<sup>nd</sup> preceding load instruction
  - Need 1 stall cycle



# Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

one chip table use কোর্পস হো'র ' PC  $\Rightarrow$  last 12 bit undefined  
value 0 or 1 ID এ মুখ্যমন্ত্রী branch taken রে'ল প্রিমেয়ার সেন্ট  
12 bit table এ মুখ্যমন্ত্রী রে'ল ।

bed id কো'র কো'র কো'র  
so predict কো'র কো'র  
if কো'র কো'র কো'র

# 1-Bit Predictor: Shortcoming

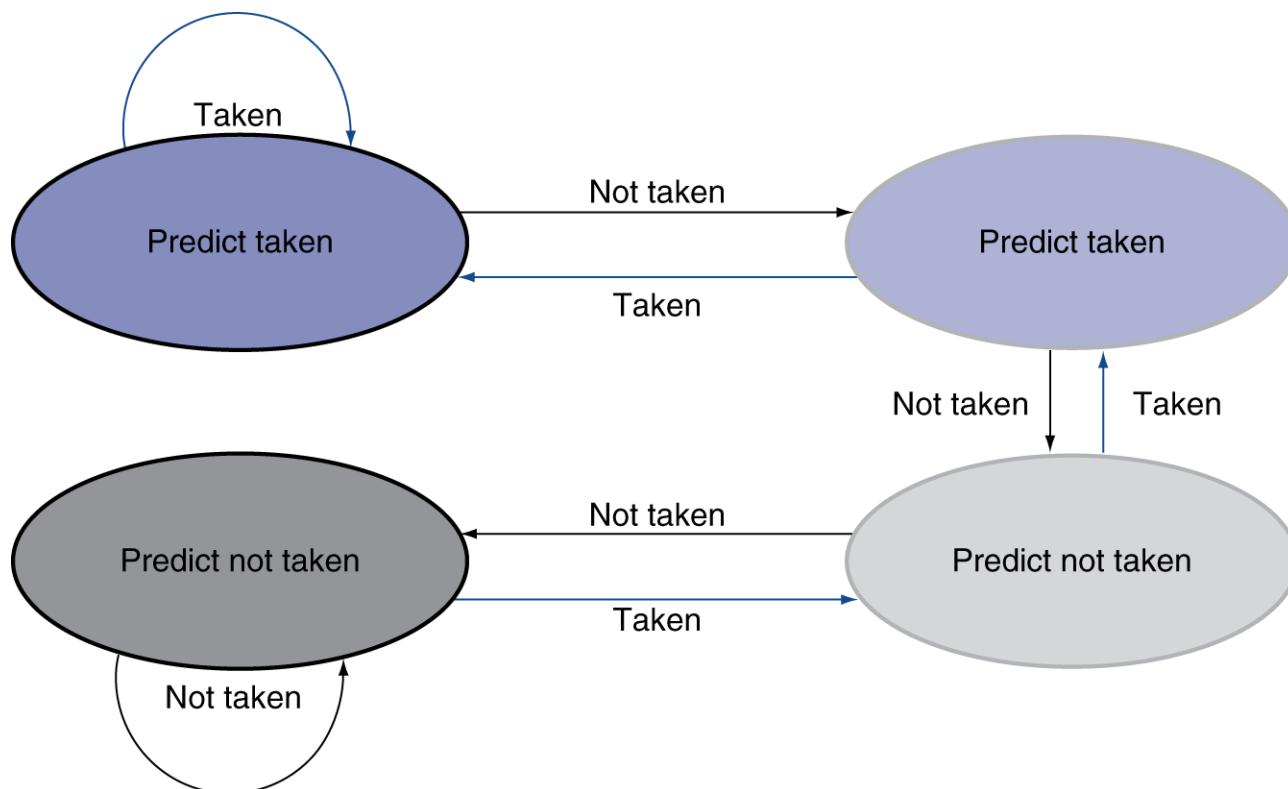
- Consider a loop branch:
  - branches nine times in a row
  - then is not taken once.
- What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?
  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

90% time  
Taken

80% time  
Accurate

# 2-Bit Predictor

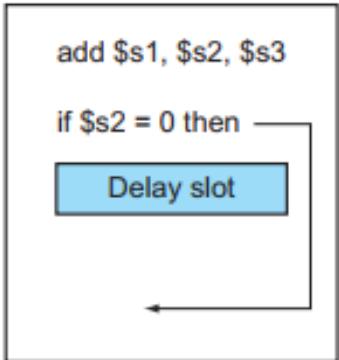
- Only change prediction on two successive mispredictions



a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once.

# Branch Delay Slot

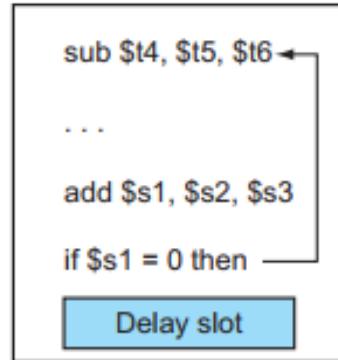
a. From before



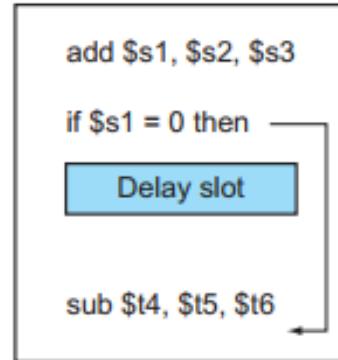
Best Option

BDS is scheduled with an independent instruction from before the branch.

b. From target



c. From fall-through



For (b) & (c):

- usually the target instruction will need to be copied because it can be reached by another path.
- it must be OK to execute the sub instruction when the branch goes in the unexpected direction.
- By “OK” we mean that the work is wasted, but the program will still execute correctly

BDS is scheduled from the target of the branch

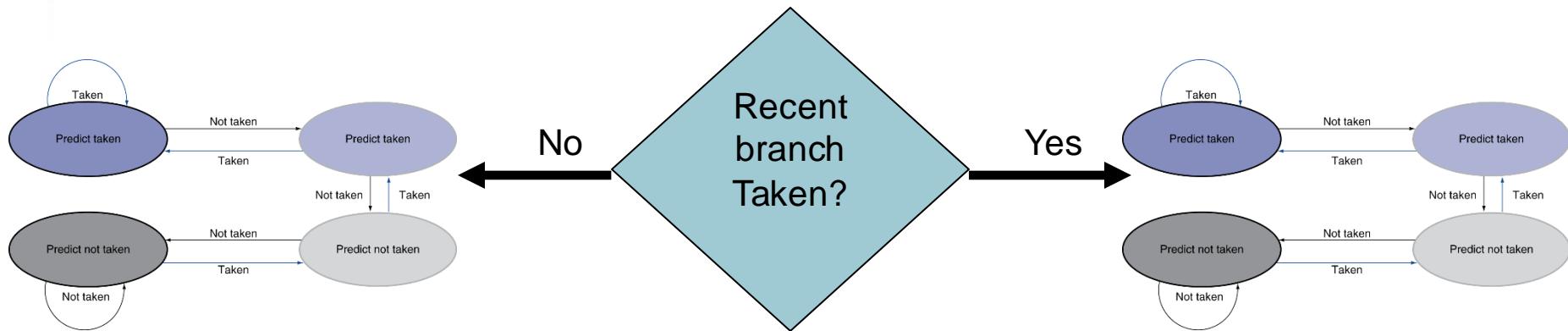
BDS may be scheduled from the not-taken fall-through

# Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

# More on Predicting

- **correlating predictor:** A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.



# More on Predicting

- **Tournament predictor:** A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

# Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

# Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

# An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - .... C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

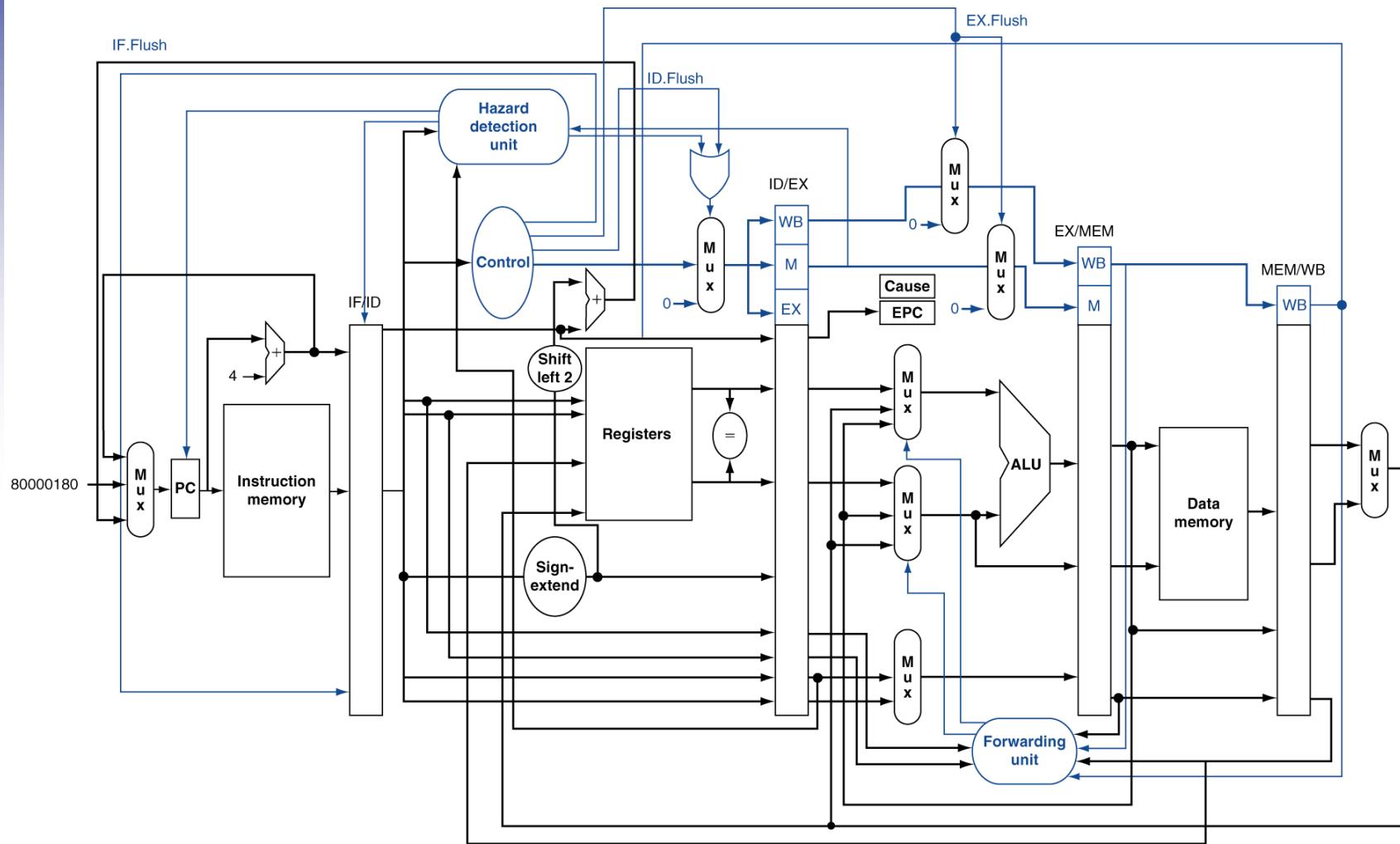
# Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, ...

# Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  - add \$1, \$2, \$1
    - Prevent \$1 from being clobbered
    - Complete previous instructions
    - Flush add and subsequent instructions
    - Set Cause and EPC register values
    - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

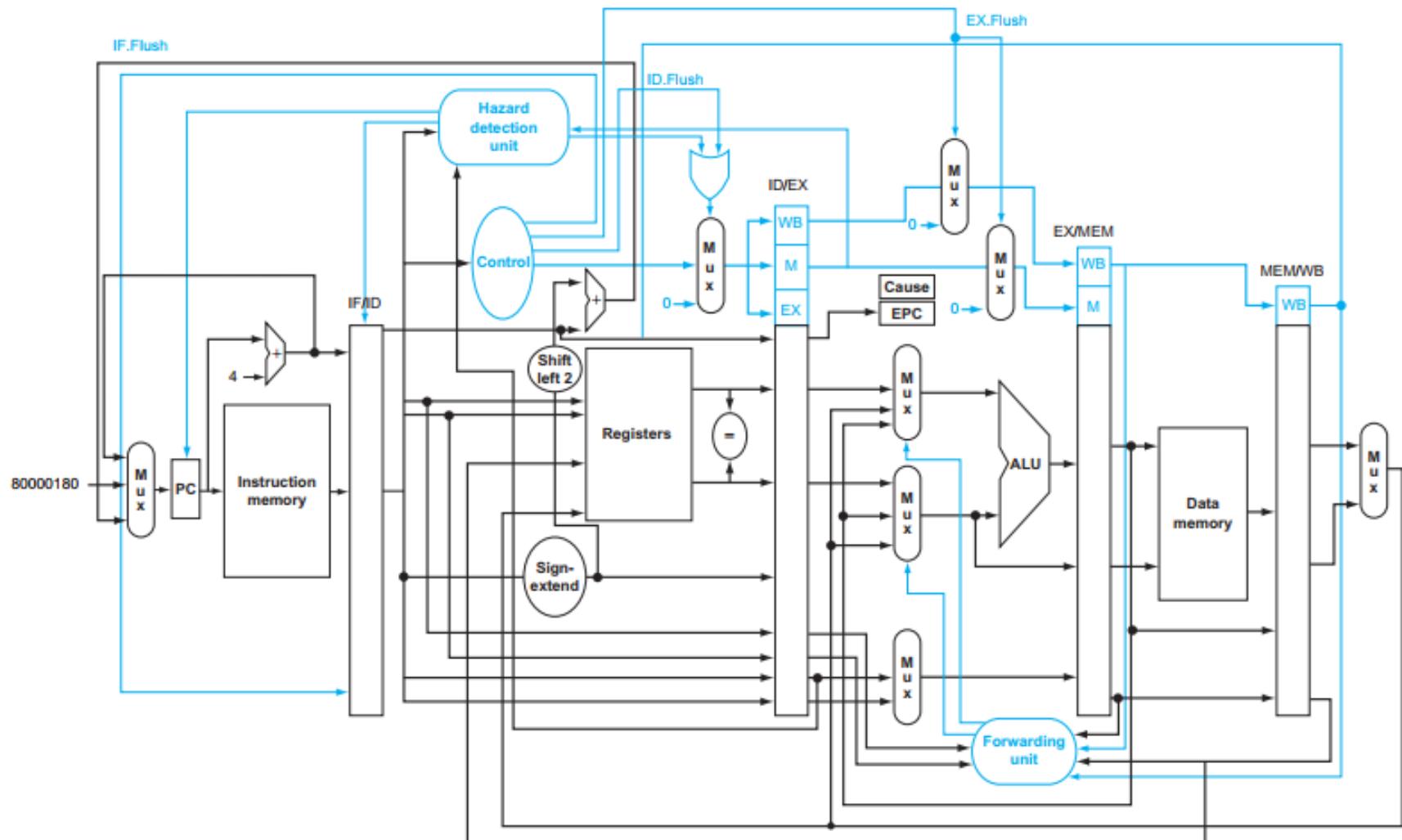
# Pipeline with Exceptions



# Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

# Datapath with controls to handle exceptions



# Exception Example

- Exception on add in

```
40      sub    $11, $2, $4
44      and    $12, $2, $5
48      or     $13, $2, $6
4C      add    $1,   $2,   $1
50      slt    $15, $6, $7
54      lw     $16, 50($7)
```

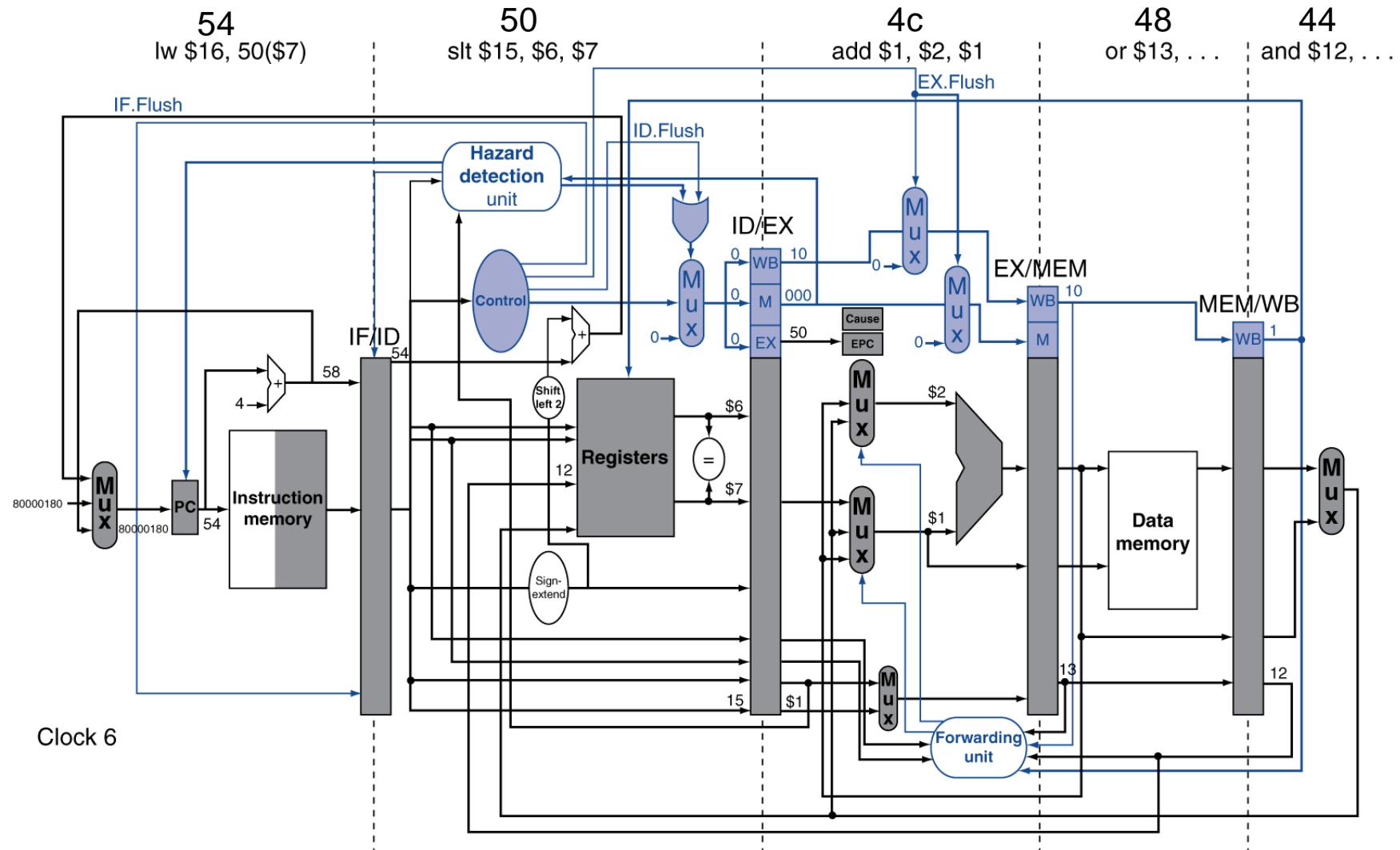
...

- Handler

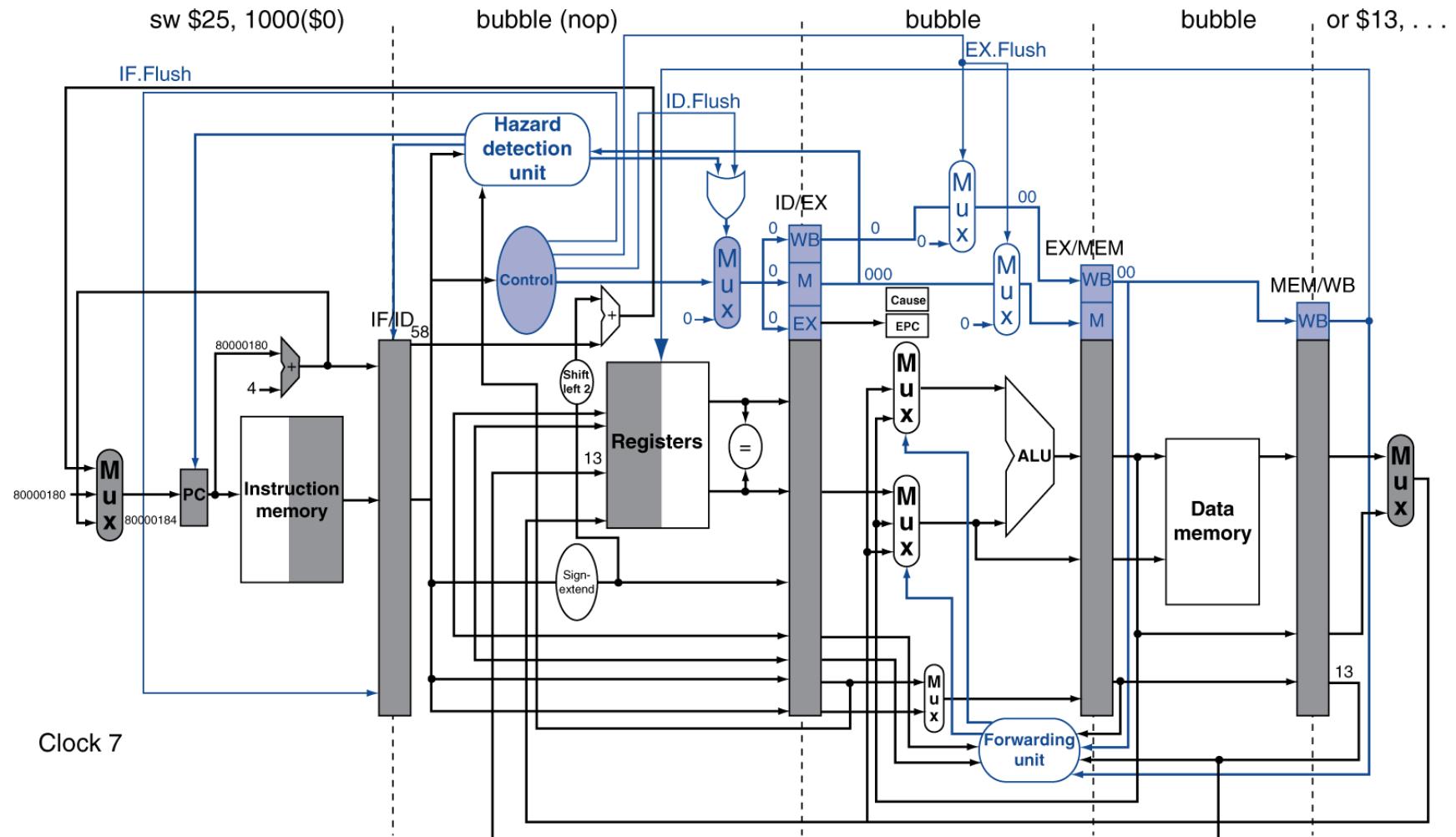
```
80000180    sw    $25, 1000($0)
80000184    sw    $26, 1004($0)
```

...

# Exception Example



# Exception Example



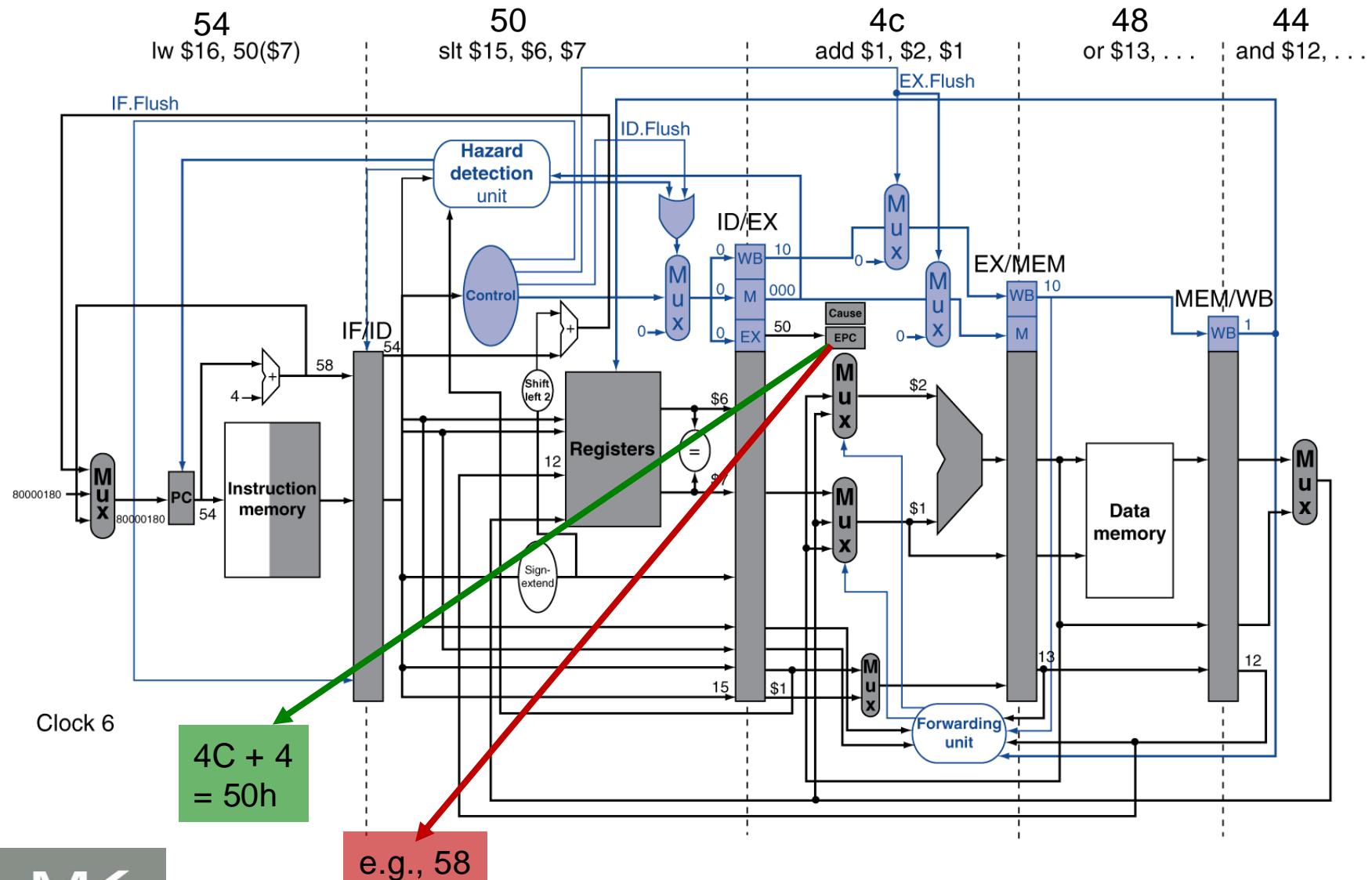
# Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

# Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

# EPC: Precise vs. Imprecise



# Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage  $\Rightarrow$  shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages  $\Rightarrow$  multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

# Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

# Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load followed by a store (whether the load is dependent on the store)
    - Roll back if location is updated

# Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

# Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for ignoring exceptions until it is clear that they really should occur
- Dynamic speculation
  - Can buffer exceptions until instruction completion
    - i.e., the instruction is not speculative anymore

# Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

# Scheduling Static Multiple Issue

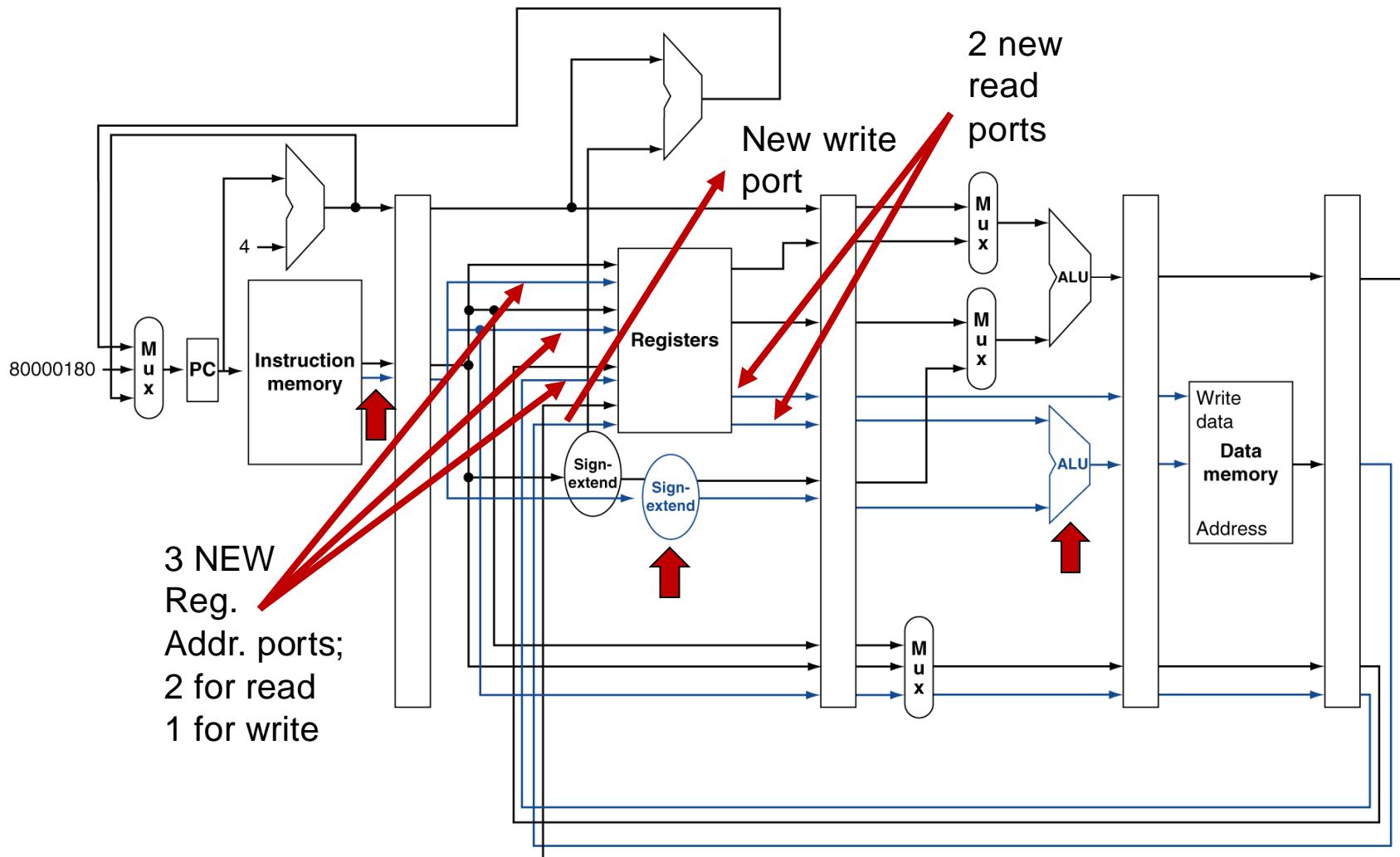
- Compiler must remove/reduce/prevent some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - This will be handled by Hardware (forwarding/Stall)
  - Pad with nop if necessary

# MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store			IF	ID	EX	MEM	WB
n + 16	ALU/branch				IF	ID	EX	MEM
n + 20	Load/store				IF	ID	EX	MEM

# MIPS with Static Dual Issue



# Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - add \$t0, \$s0, \$s1
    - load \$s2, 0(\$t0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - (between two issue packet)
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

# Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw    $t0, 0($s1)      # $t0=array element
       addu $t0, $t0, $s2      # add scalar in $s2
       sw    $t0, 0($s1)      # store result
       addi $s1, $s1,-4        # decrement pointer
       bne  $s1, $zero, Loop   # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw    \$t0, 0(\$s1)	1
	addi \$s1, \$s1,-4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne  \$s1, \$zero, Loop	sw    \$t0, 4(\$s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

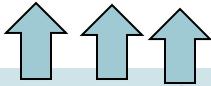
# Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

# Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)	1
	nop	lw \$t1, 12(\$s1)	2
	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1)	3
	addu \$t1, \$t1, \$s2	lw \$t3, 4(\$s1)	4
	addu \$t2, \$t2, \$s2	sw \$t0, 16(\$s1)	5
	addu \$t3, \$t4, \$s2	sw \$t1, 12(\$s1)	6
	nop	sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

Loop: lw \$t0, 0(\$s1)  
addu \$t0, \$t0, \$s2  
sw \$t0, 0(\$s1)  
addi \$s1, \$s1, -4  
bne \$s1, \$zero, Loop



- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

# Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help

# Simple superscalar vs. VLIW

- The code is guaranteed by the hardware to execute correctly.
  - independent of the issue rate
  - or pipeline structure of the processor.
- In VLIW, recompilation may be required for different processor models;
- In other static issue processors, code should run correctly across different implementations
  - but often so poorly as to make compilation effectively required.

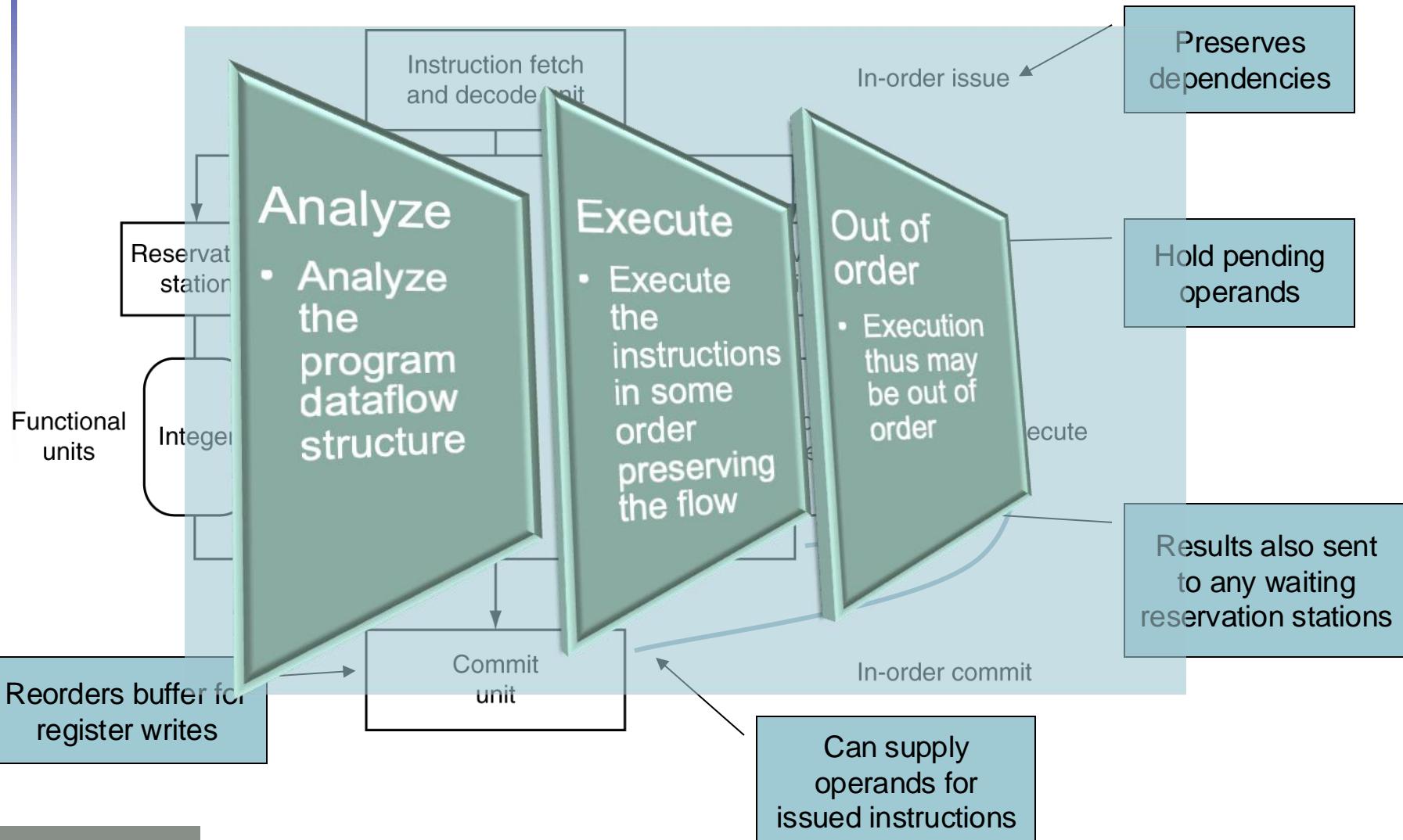
# Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
lw      $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slti   $t5, $s4, 20
```

- Can start sub while addu is waiting for lw

# Dynamically Scheduled CPU



# Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
      - The name of the functional unit is tracked
    - Register update may not be required

# Speculation

- Predict branch and continue issuing
  - Don't commit until branch outcome determined
- Load speculation
  - support speculation on load addresses,
  - allowing load-store reordering,
  - Don't commit load until speculation cleared

# Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

# Does Multiple Issue Work?

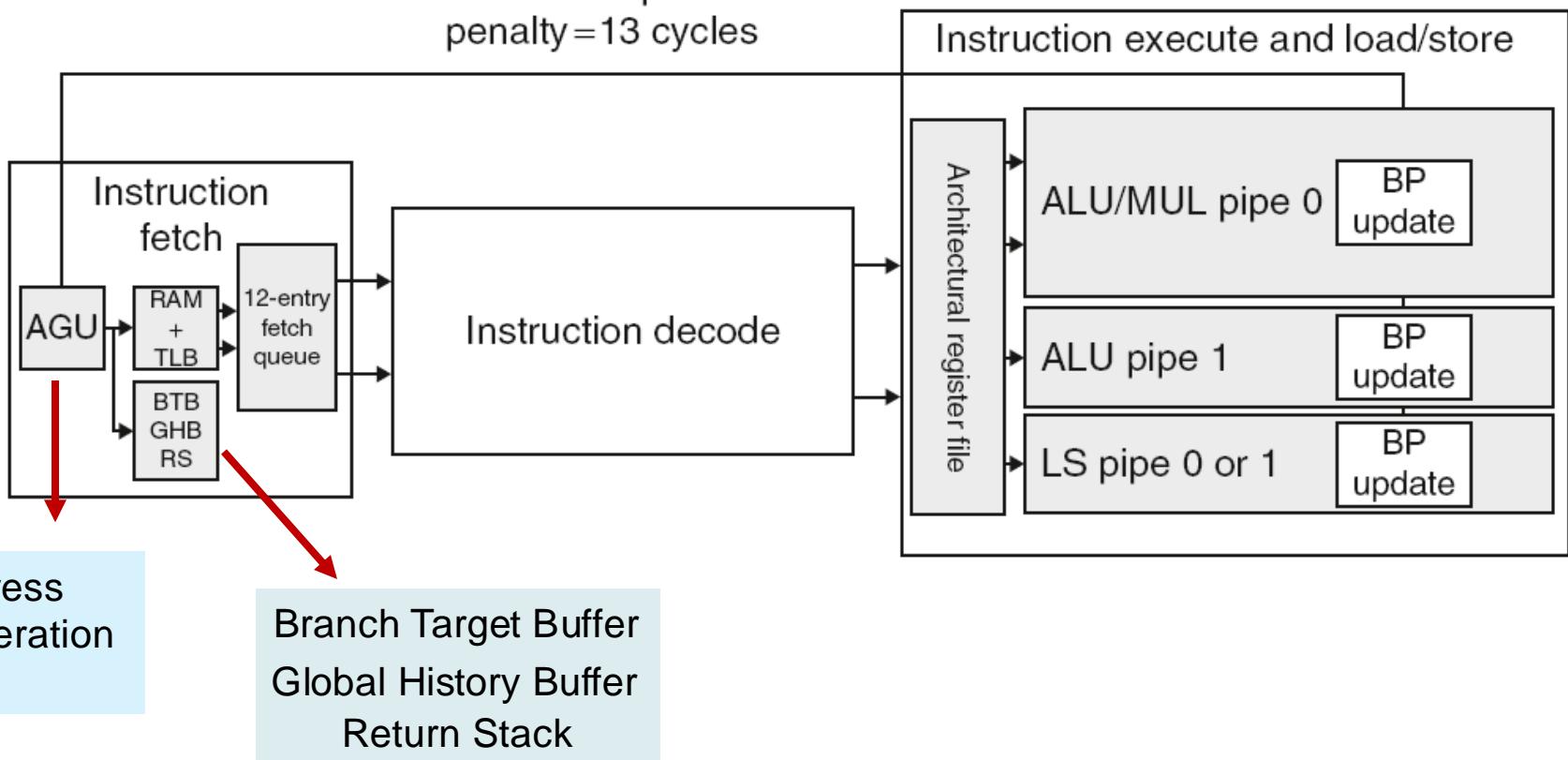
## The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
- Some parallelism is hard to expose
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

# ARM Cortex-A8 Pipeline

F0 F1 F2 D0 D1 D2 D3 D4 E0 E1 E2 E3 E4 E5

Branch mispredict  
penalty = 13 cycles



# Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall