

# Chapter 4

## The Processor

# Outline<sub>1</sub>

---

- Introduction
- Logic Design Conventions
- Building a Datapath
- A Simple Implementation Scheme
- An Overview of Pipelining
- Pipelined Datapath and Control
- Data Hazards: Forwarding versus Stalling
- Control Hazards
- Exceptions
- Parallelism via Instructions

# Outline<sub>2</sub>

---

- Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines
- Going Faster: Instruction-Level Parallelism and Matrix Multiply
- Fallacies and Pitfalls
- Concluding Remarks

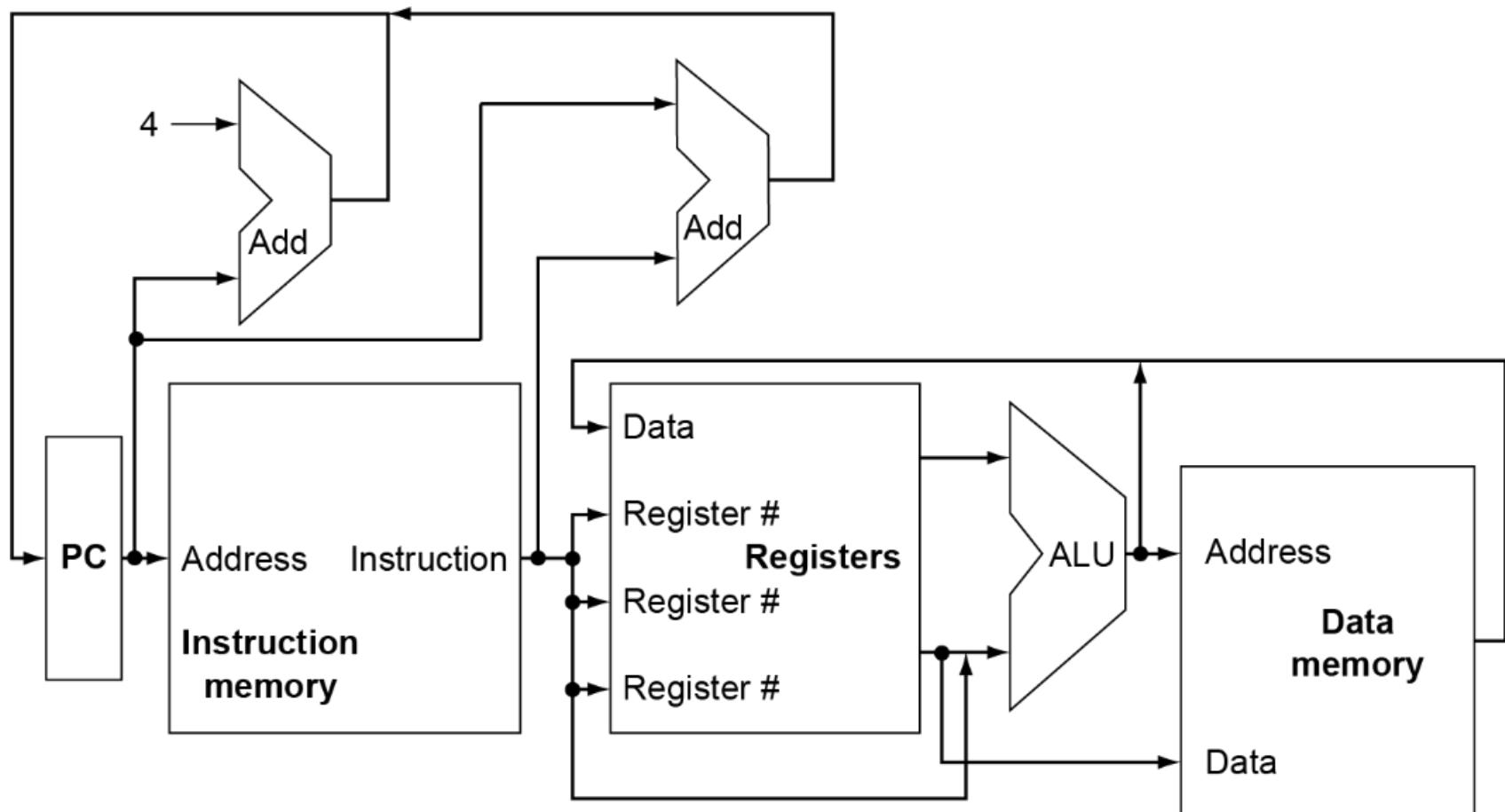
# Introduction

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

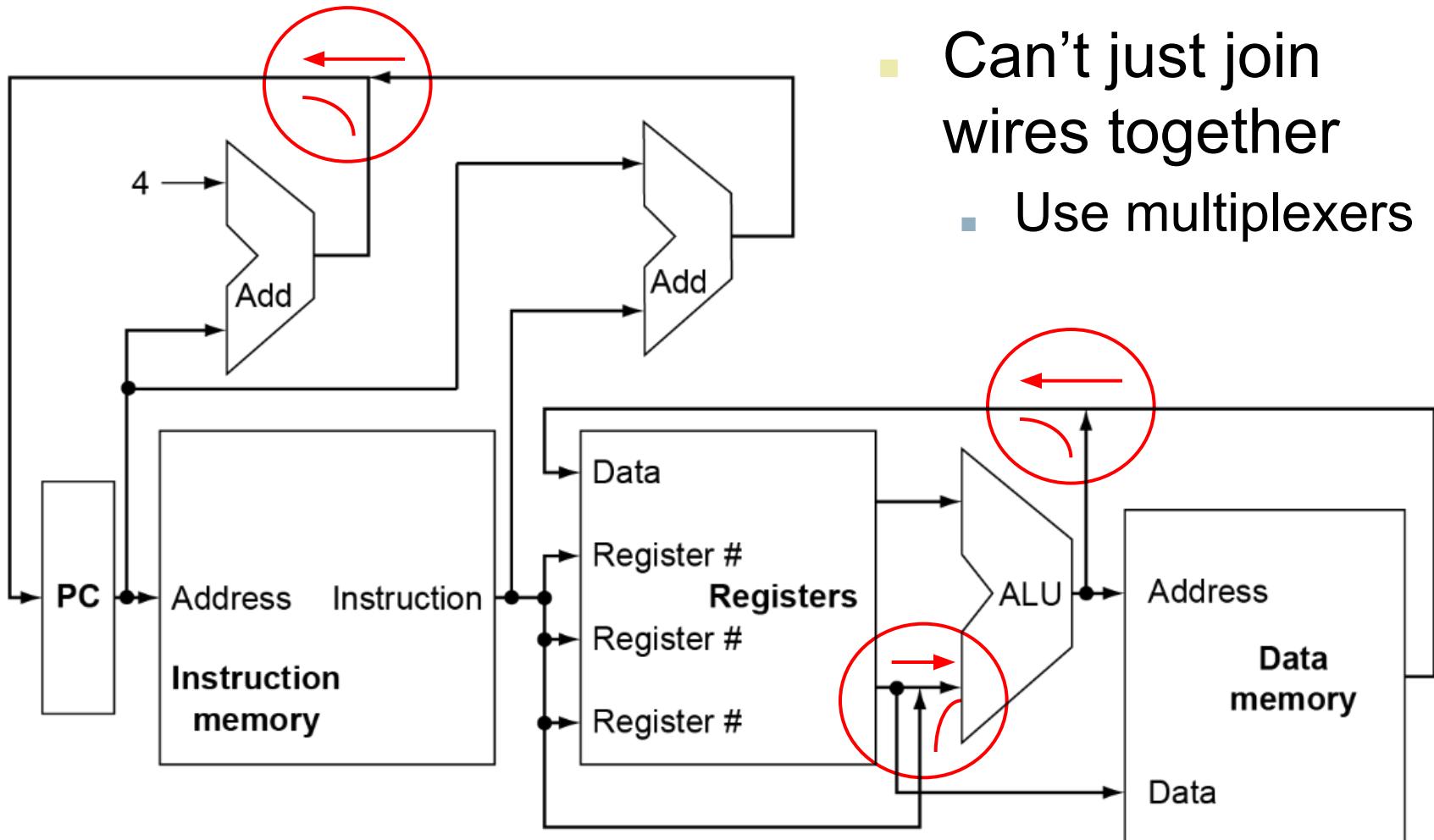
# Instruction Execution

- 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 comparison
  - Access data memory for load/store
  - PC ← target address or PC + 4

# CPU Overview

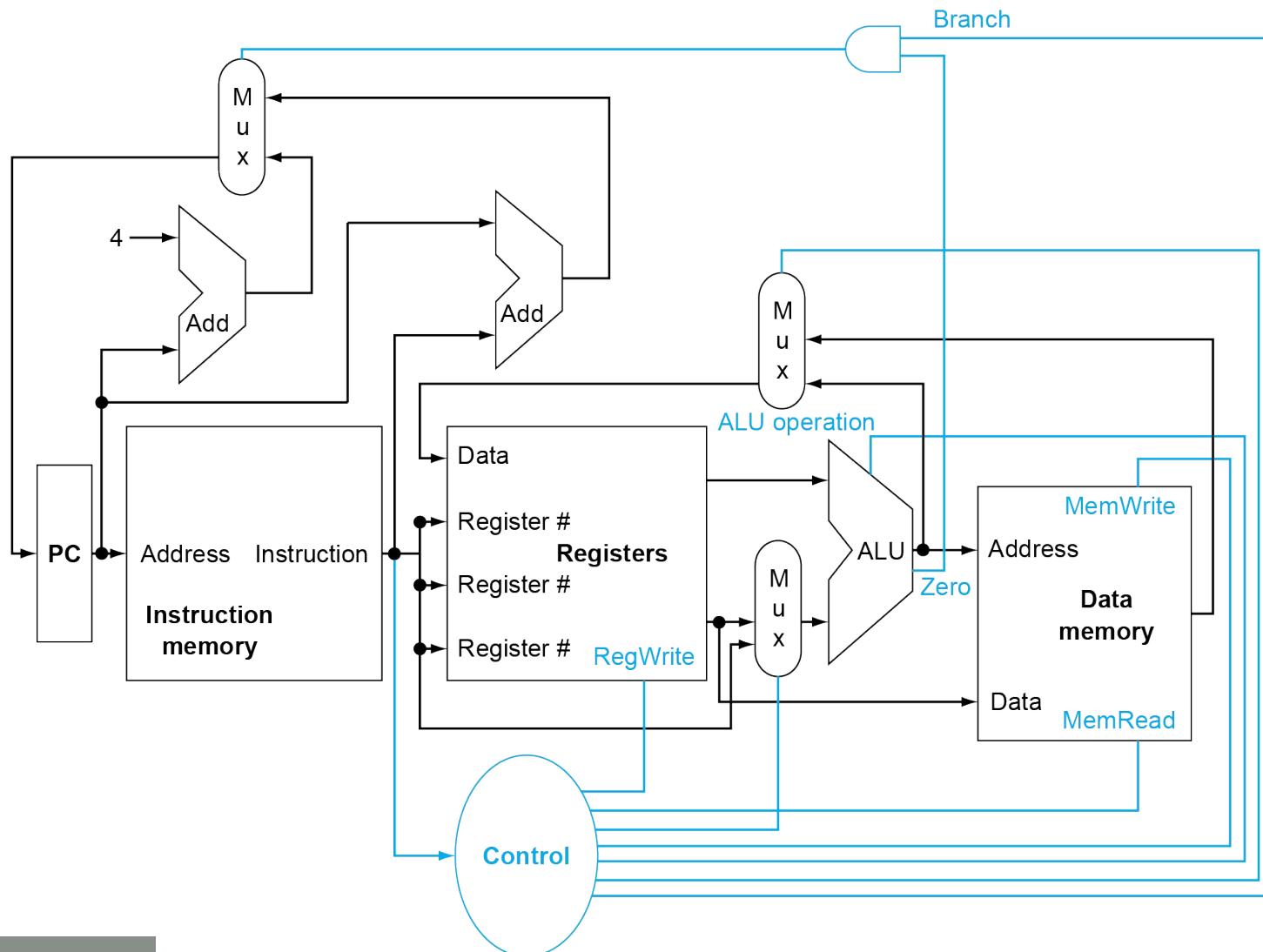


# Multiplexers



- Can't just join wires together
  - Use multiplexers

# Control



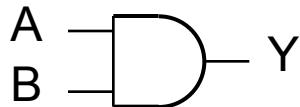
# Logic Design Basics

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

# Combinational Elements

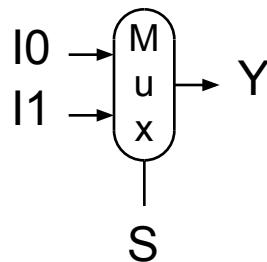
- AND-gate

- $Y = A \& B$



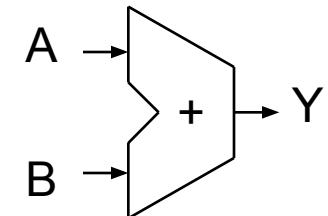
- Multiplexer

- $Y = S ? I_1 : I_0$



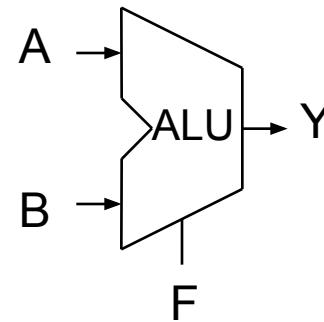
- Adder

- $Y = A + B$



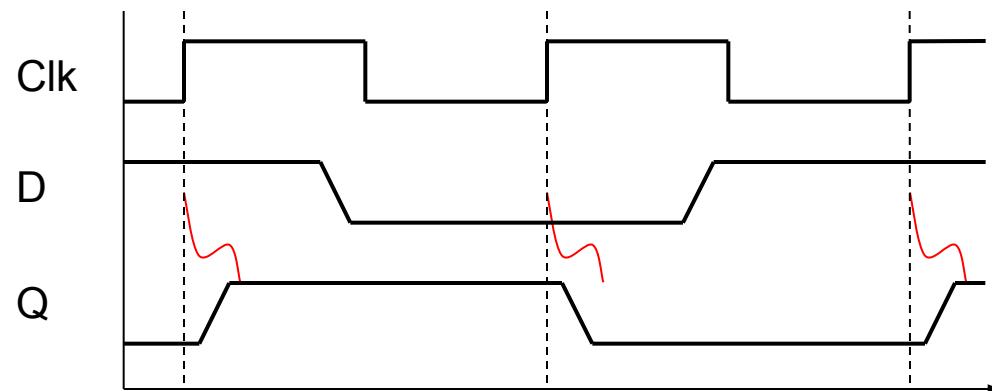
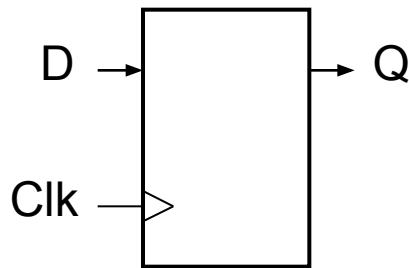
- Arithmetic/Logic Unit

- $Y = F(A, B)$



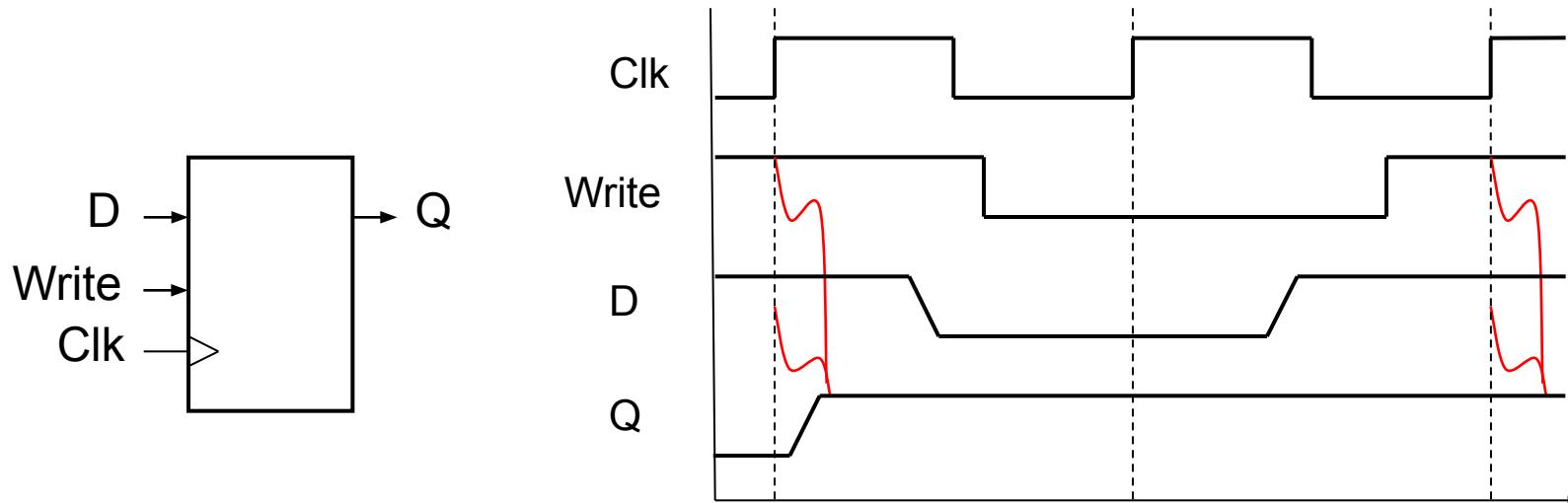
# Sequential Elements<sub>1</sub>

- 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

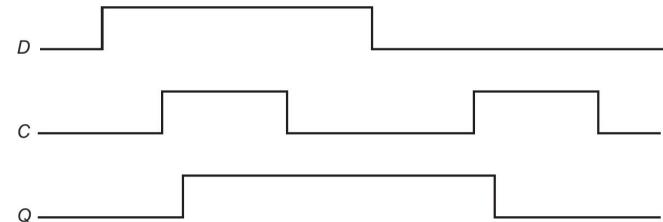
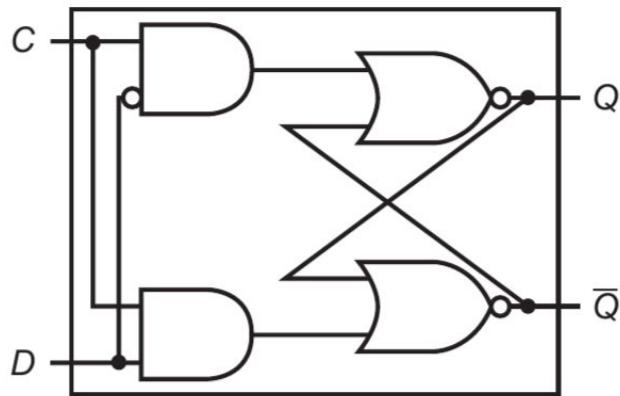


# Sequential Elements<sub>2</sub>

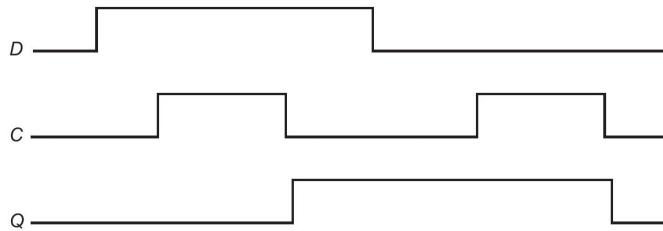
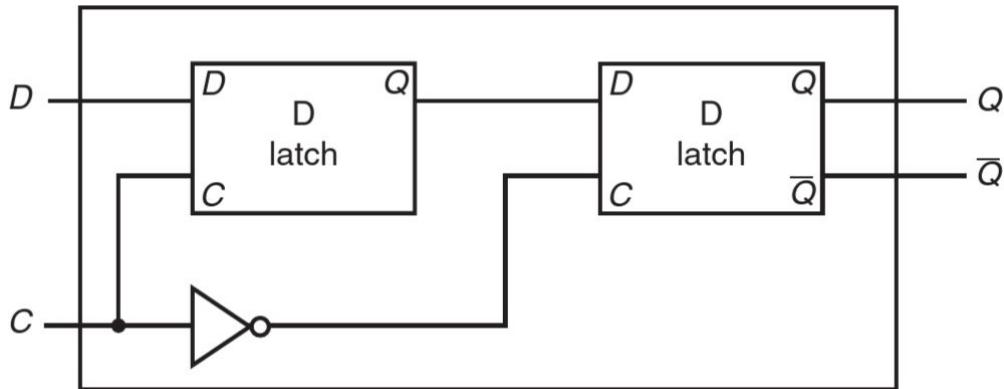
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



# An Implementation



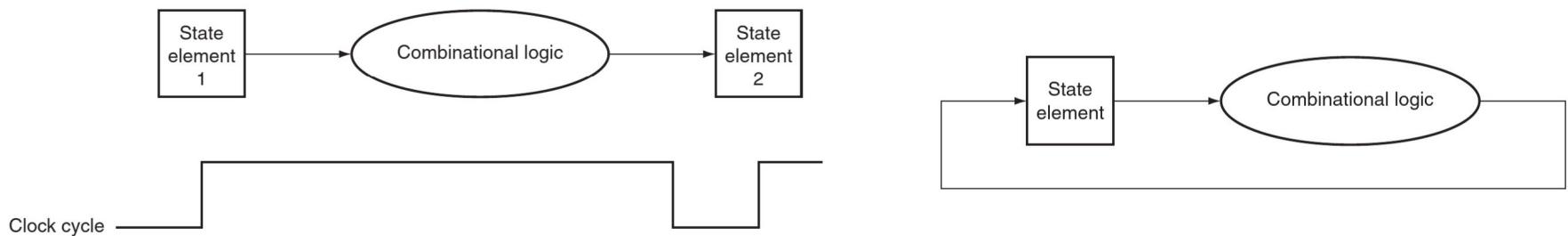
**D Latch:** if  $C == 1$ , set  $Q = D$ ; otherwise ( $C == 0$ ),  $Q$  is latched.



A D Flip-flop with a falling-edge trigger

# Clocking Methodology

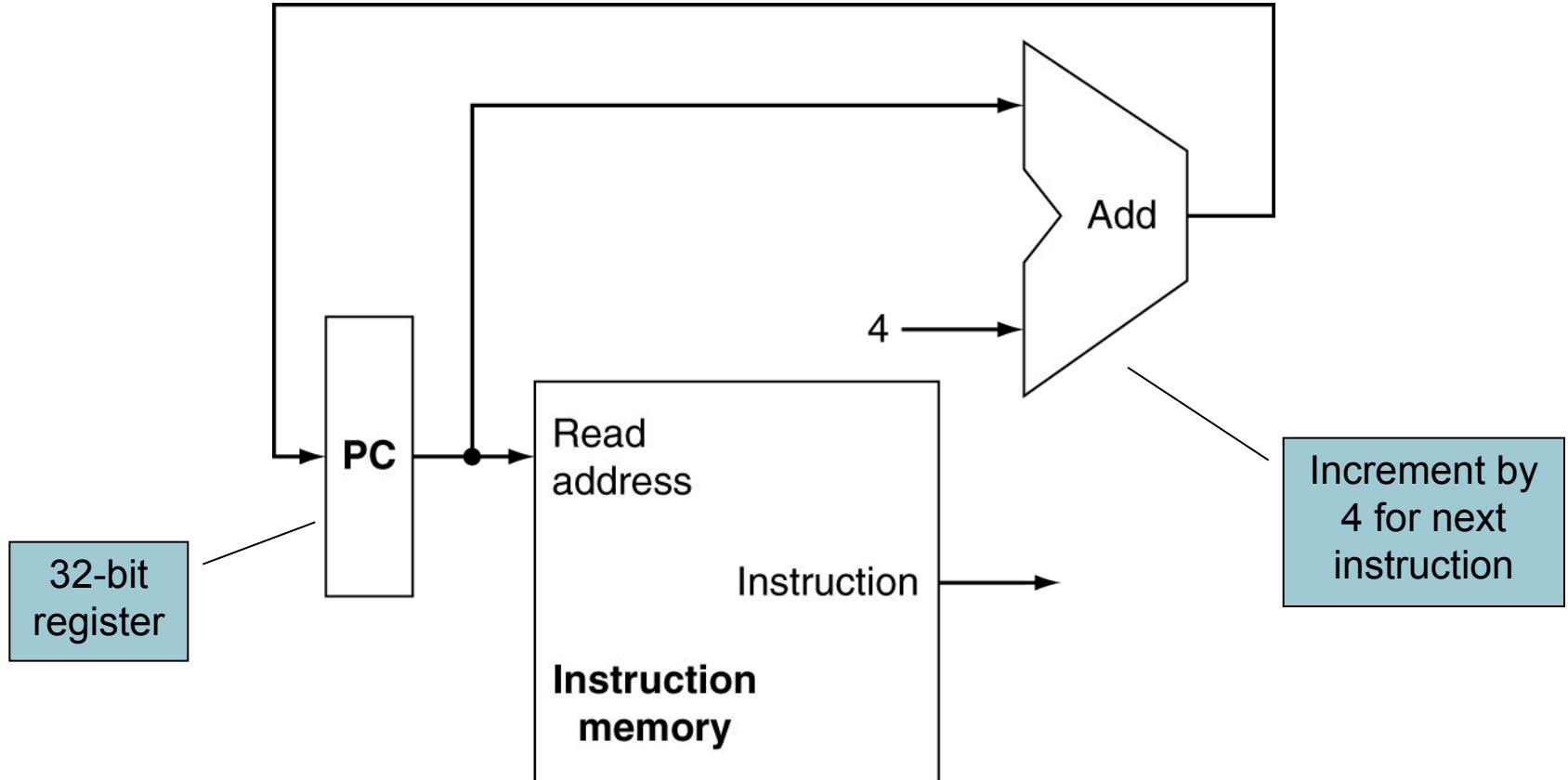
- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period



# Building a Datapath

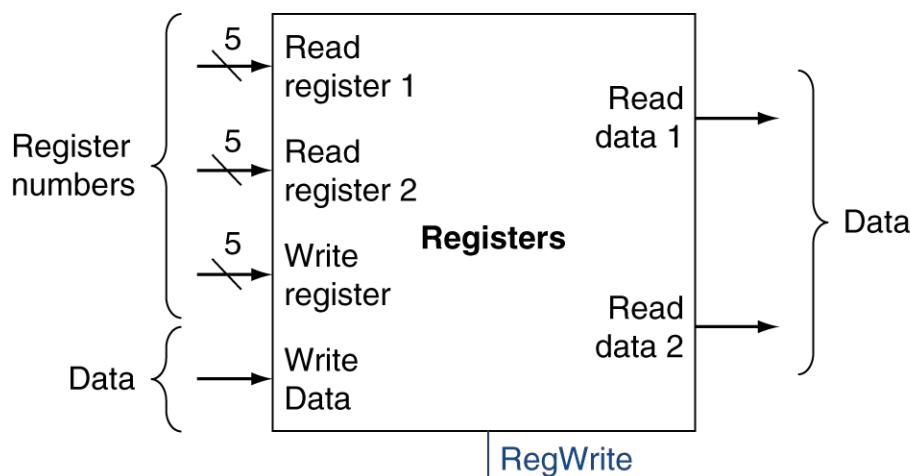
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, ...
- We will build a simple datapath incrementally
  - Refining the overview design

# Instruction Fetch

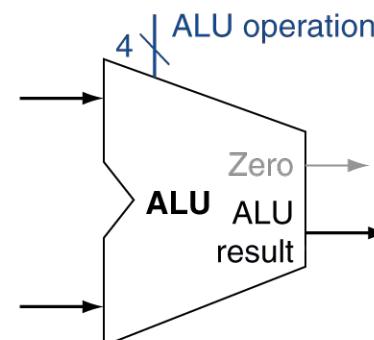


# R-Format Instructions

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

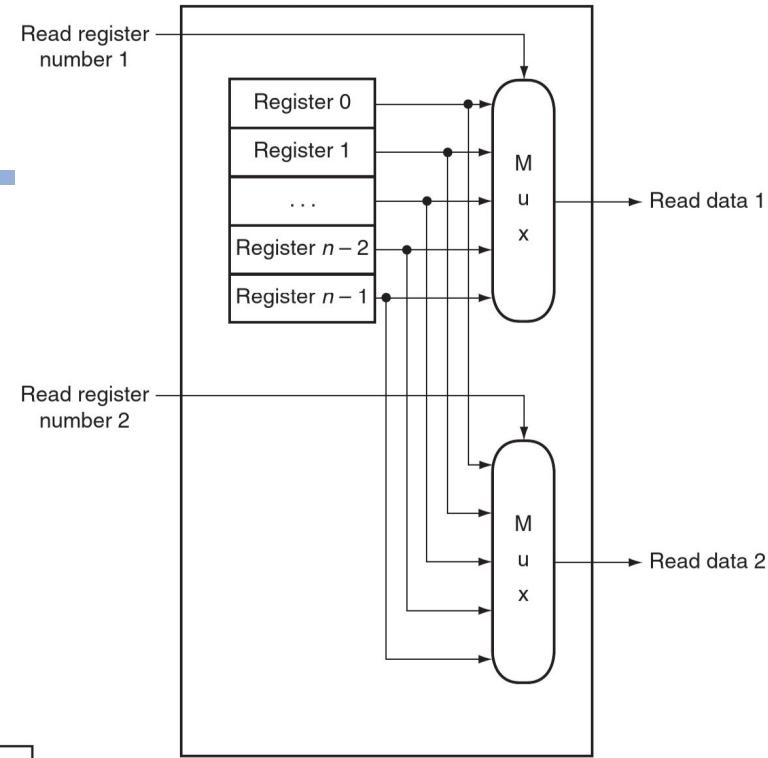
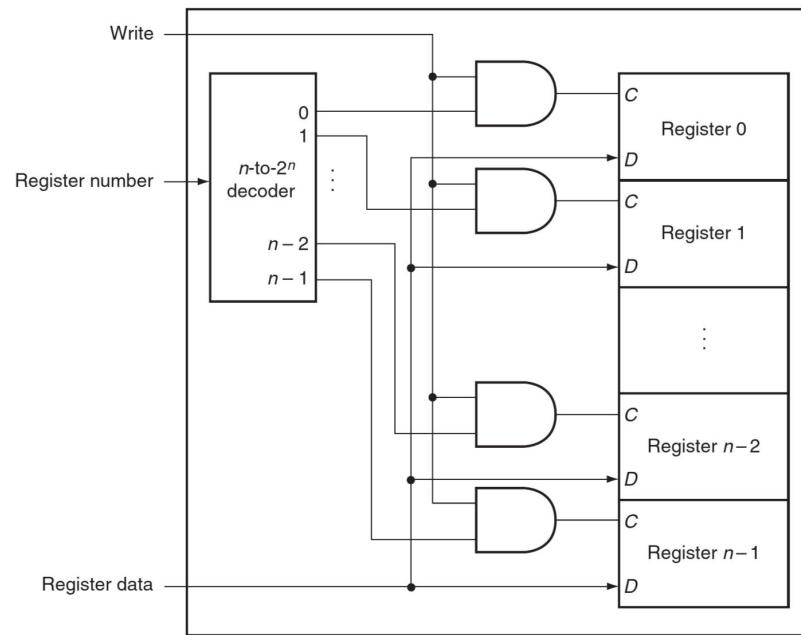
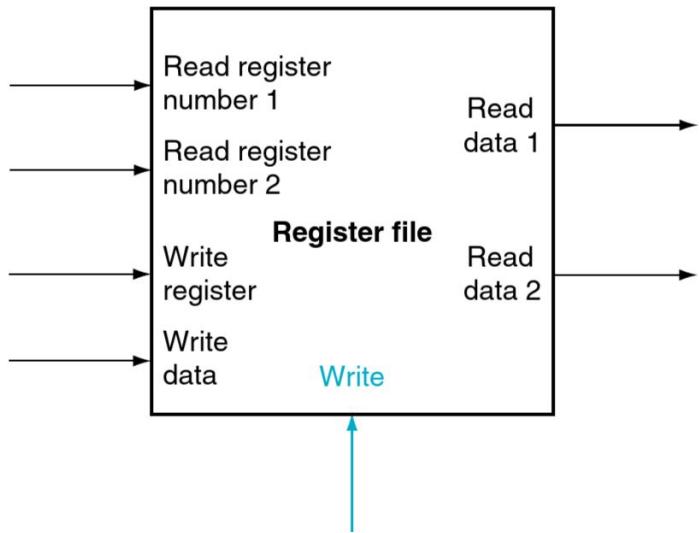


a. Registers



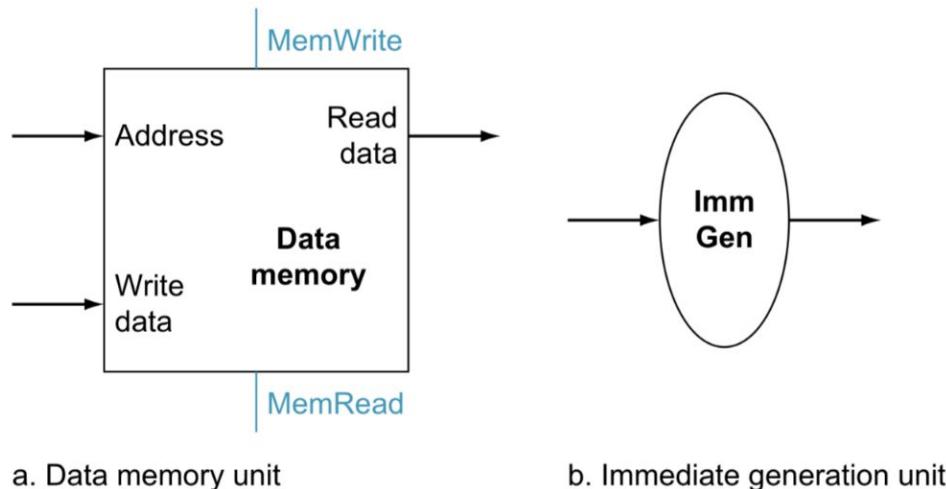
b. ALU

# Register Files



# Load/Store Instructions

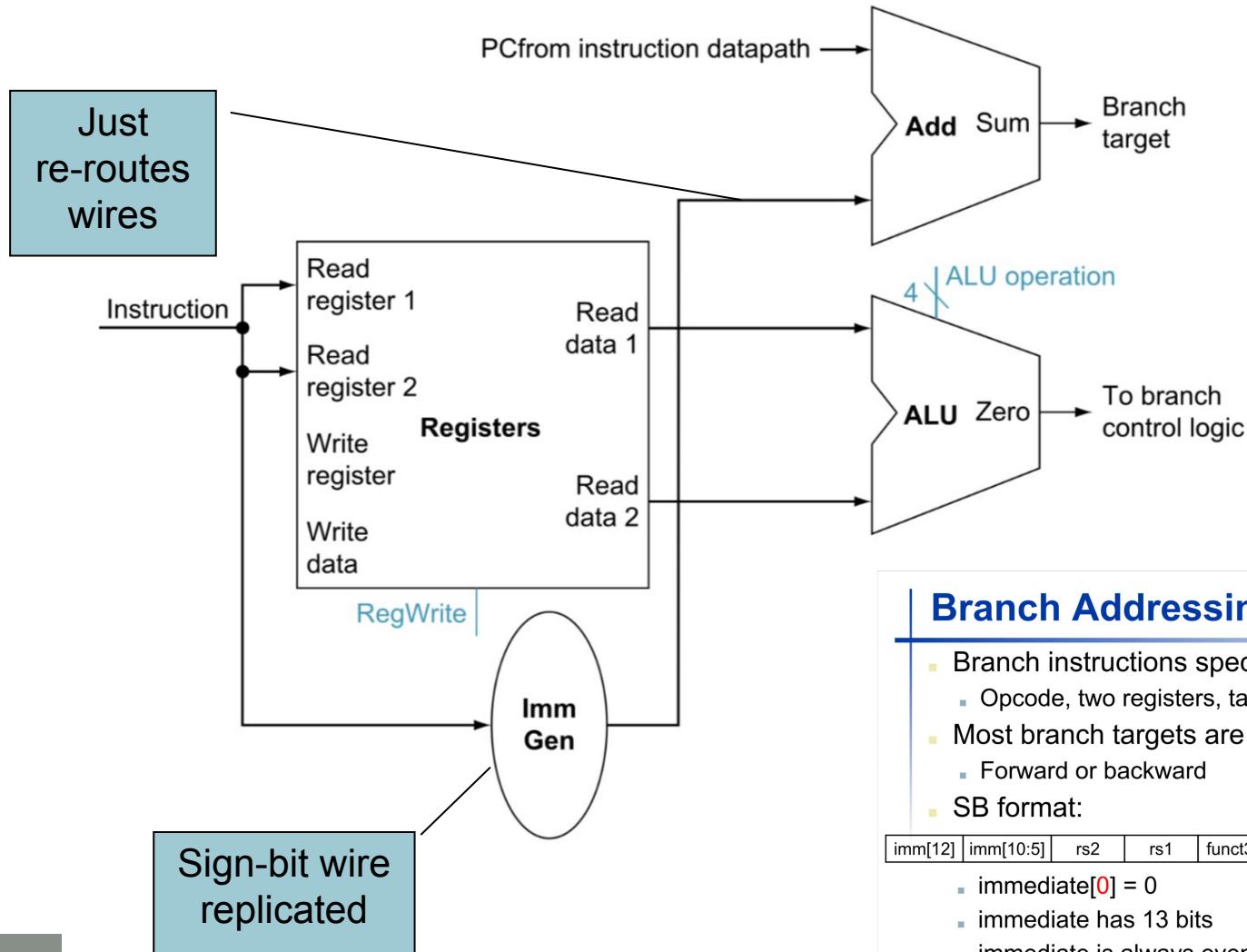
- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



# Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address `beq x10, x11, offset`
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

# Branch Instructions



## Branch Addressing<sub>1</sub>

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:

imm[12] imm[10:5] rs2 rs1 funct3 imm[4:1] imm[11] opcode

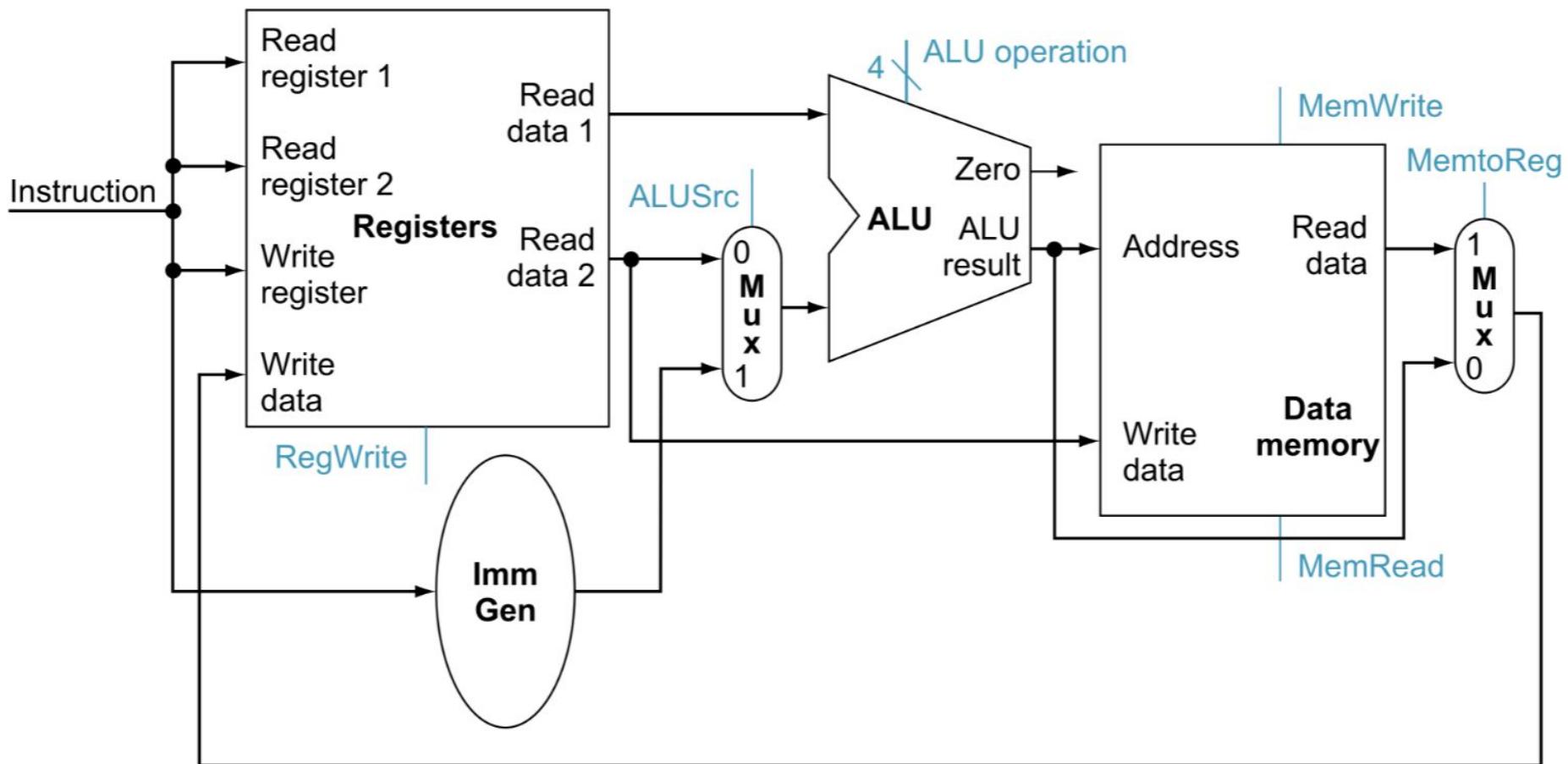
- immediate[0] = 0
- immediate has 13 bits
- immediate is always even

Ex: bne x10, x11, 200 // if x10 != x11, go to pc + 200

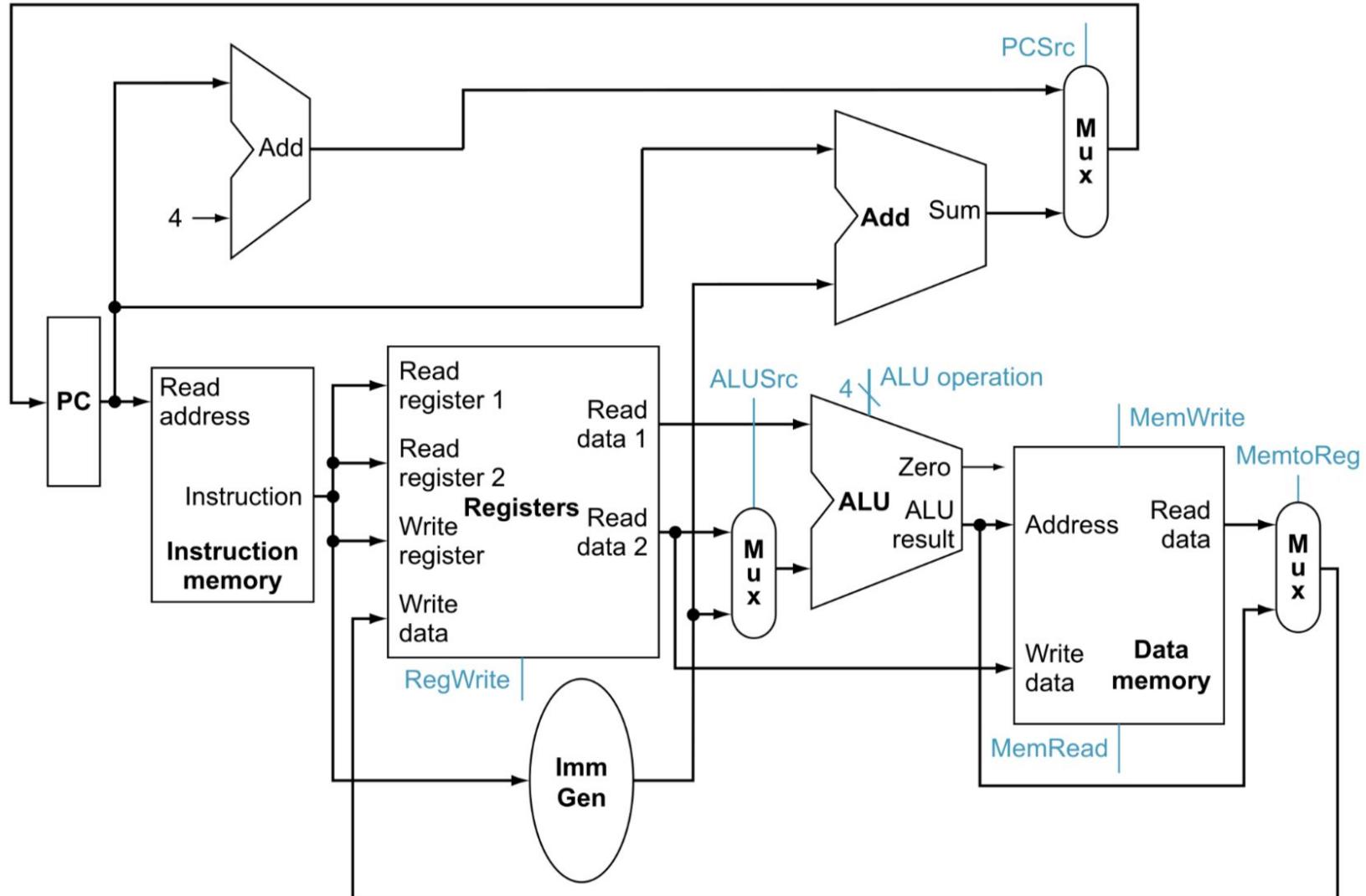
# Composing the Elements

- 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

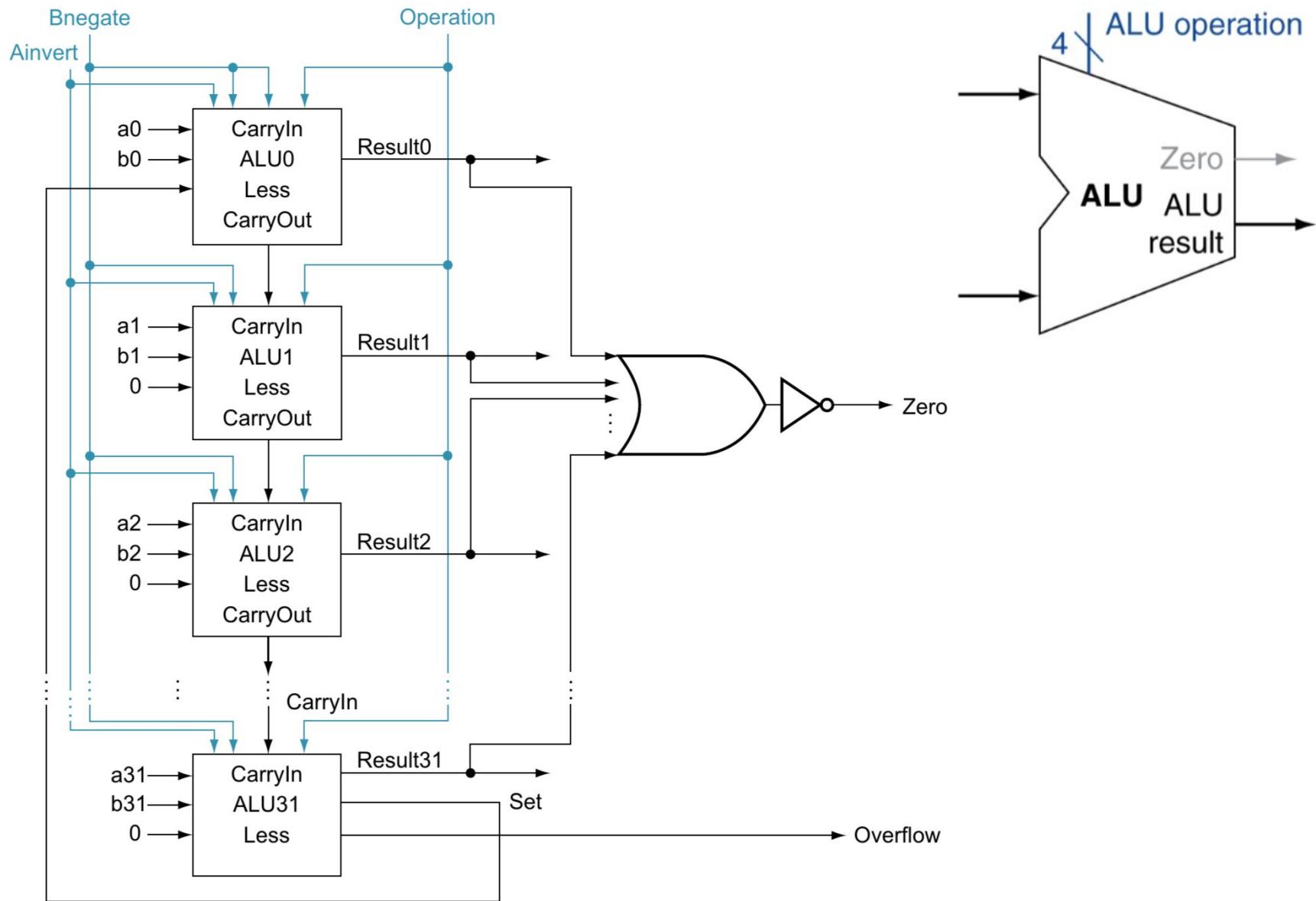
# R-Type/Load/Store Datapath



# Full Datapath



# An ALU with control



# ALU Control

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

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

# ALU Control

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

opcode	ALUOp	Operation	funct7+funct3	ALU function	ALU control
lw	00	load word	XXXXXXXXXX	add	0010
sw	00	store word	XXXXXXXXXX	add	0010
beq	01	branch if equal	XXXXXXXXXX	subtract	0110
R-type	10	add	0000000000	add	0010
		subtract	0100000000	subtract	0110
		AND	0000000111	AND	0000
		OR	0000000110	OR	0001

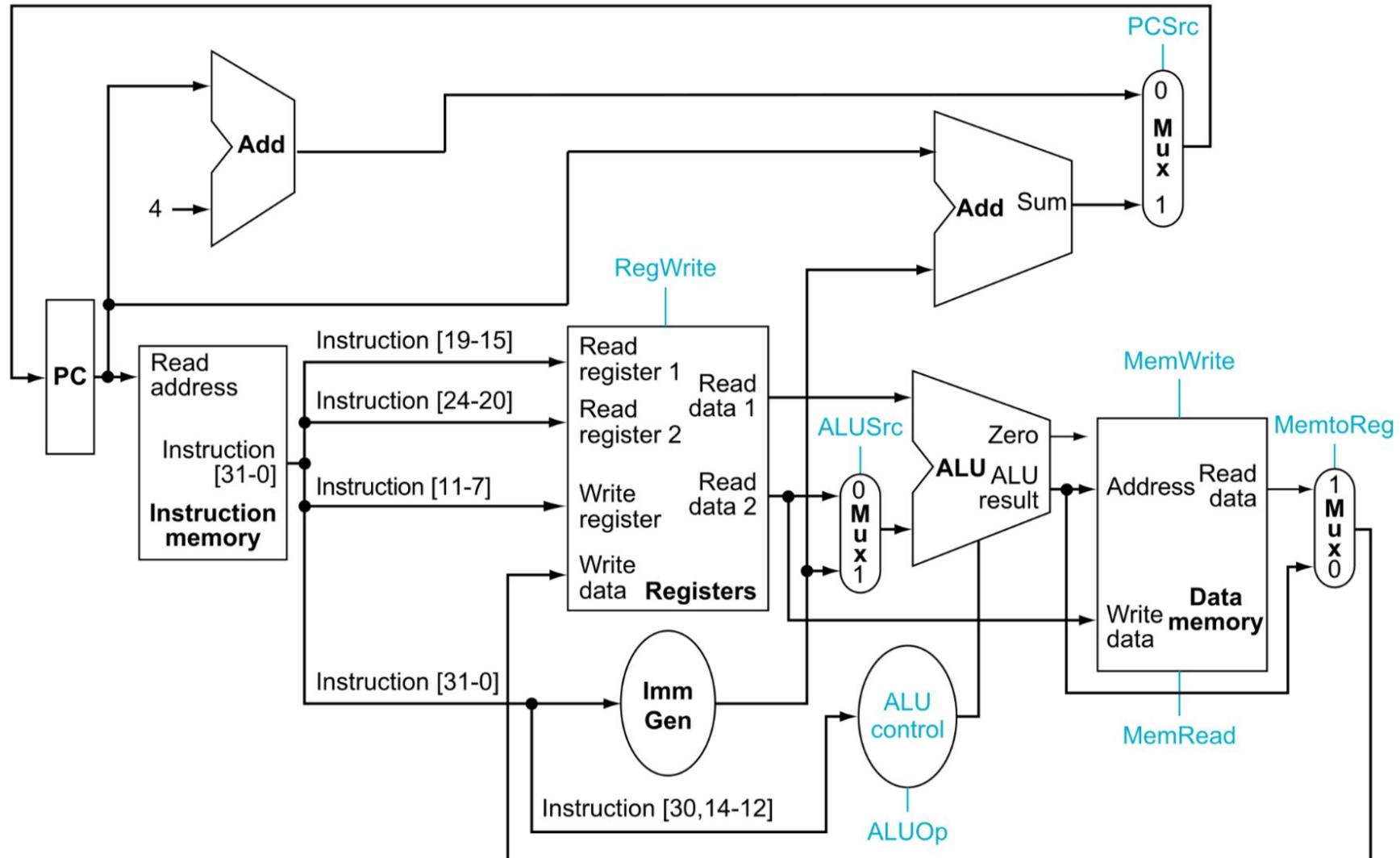
# The Main Control Unit

## Control signals derived from instruction

Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp	Funct7 field											Operation	
	ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	0	1	1	0	0001

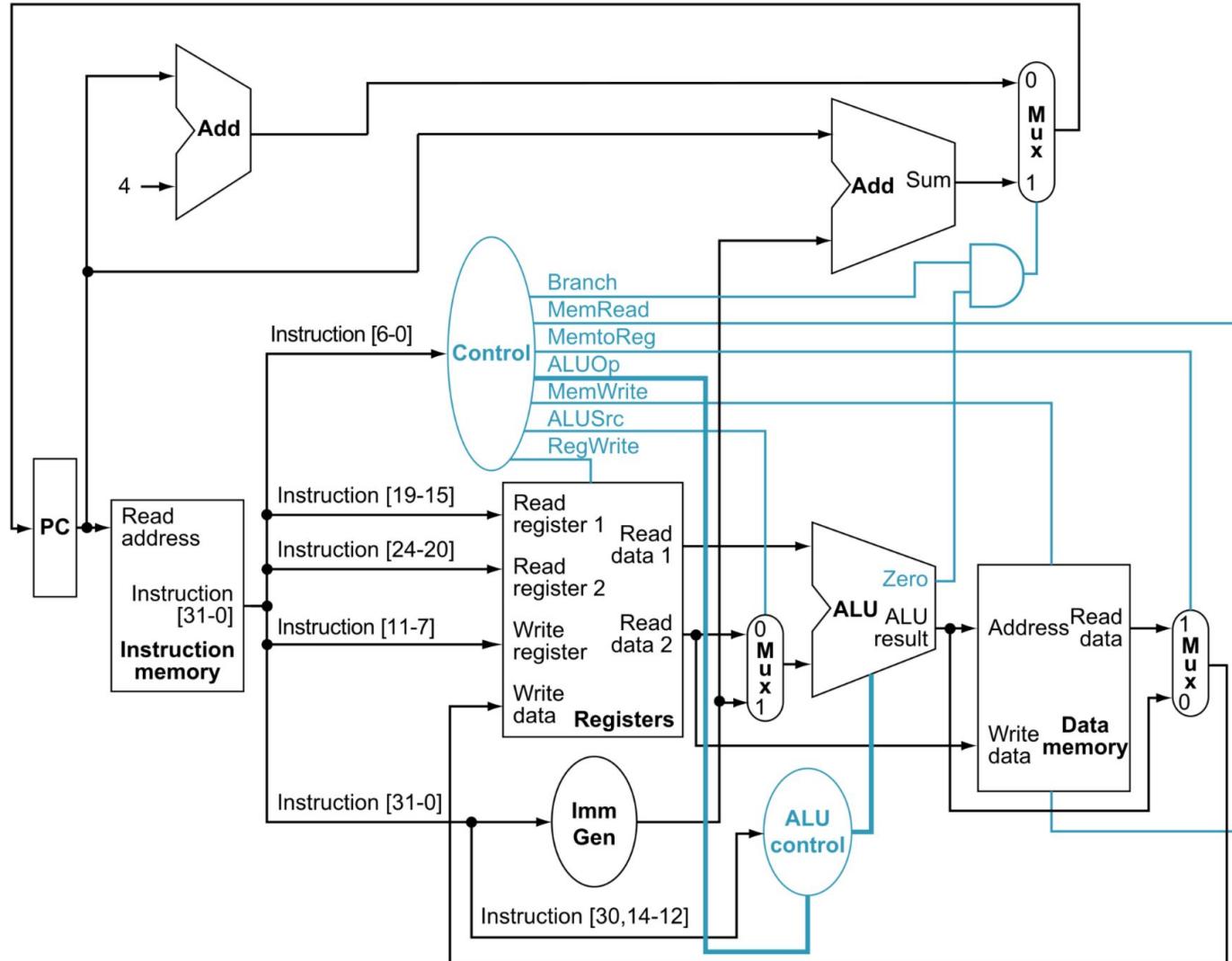
# Datapath With Control<sub>1</sub>



# Datapath With Control<sub>2</sub>

Signal name	Effect when deasserted	Effect when asserted
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, 12 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.

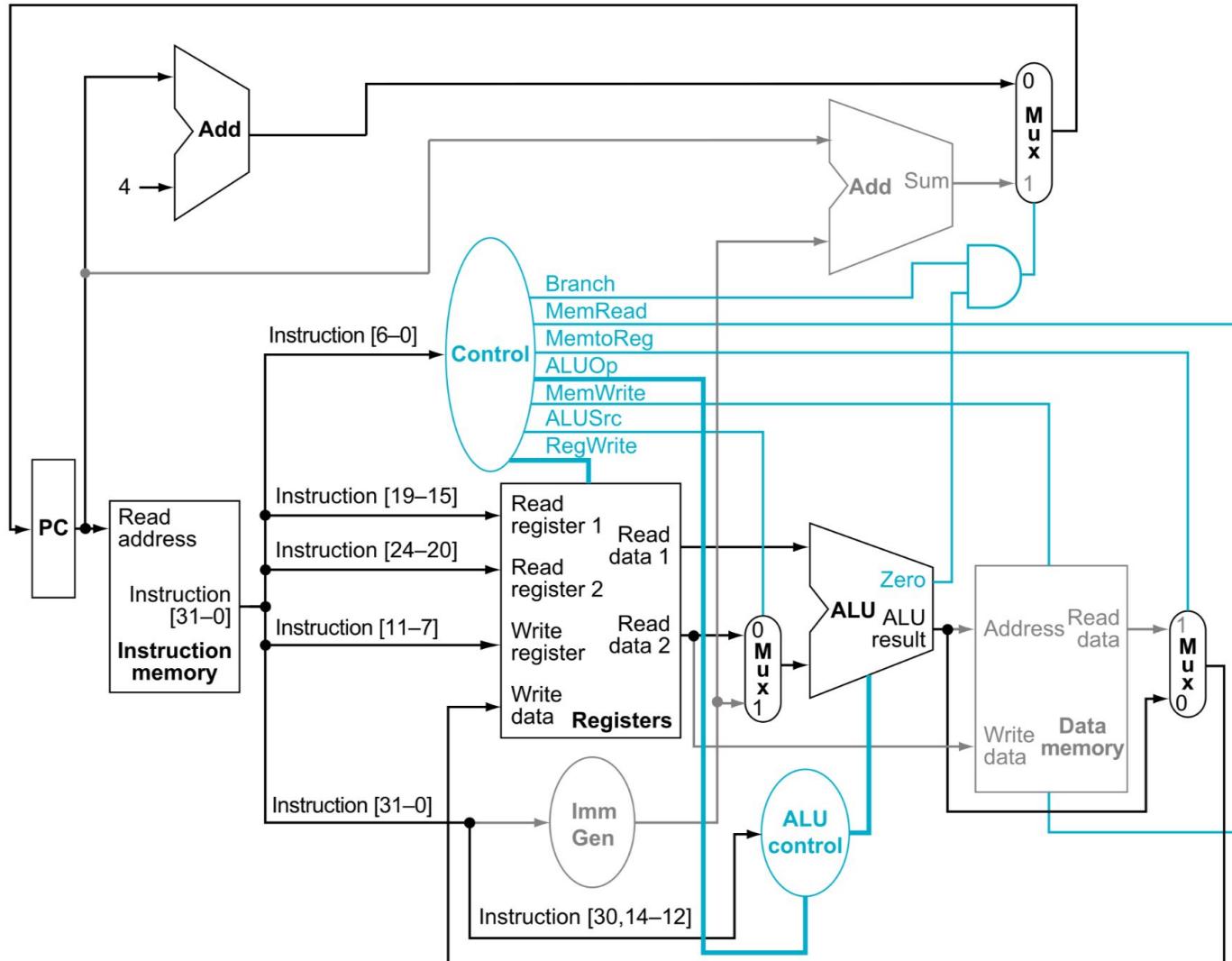
# Datapath With Control<sub>3</sub>



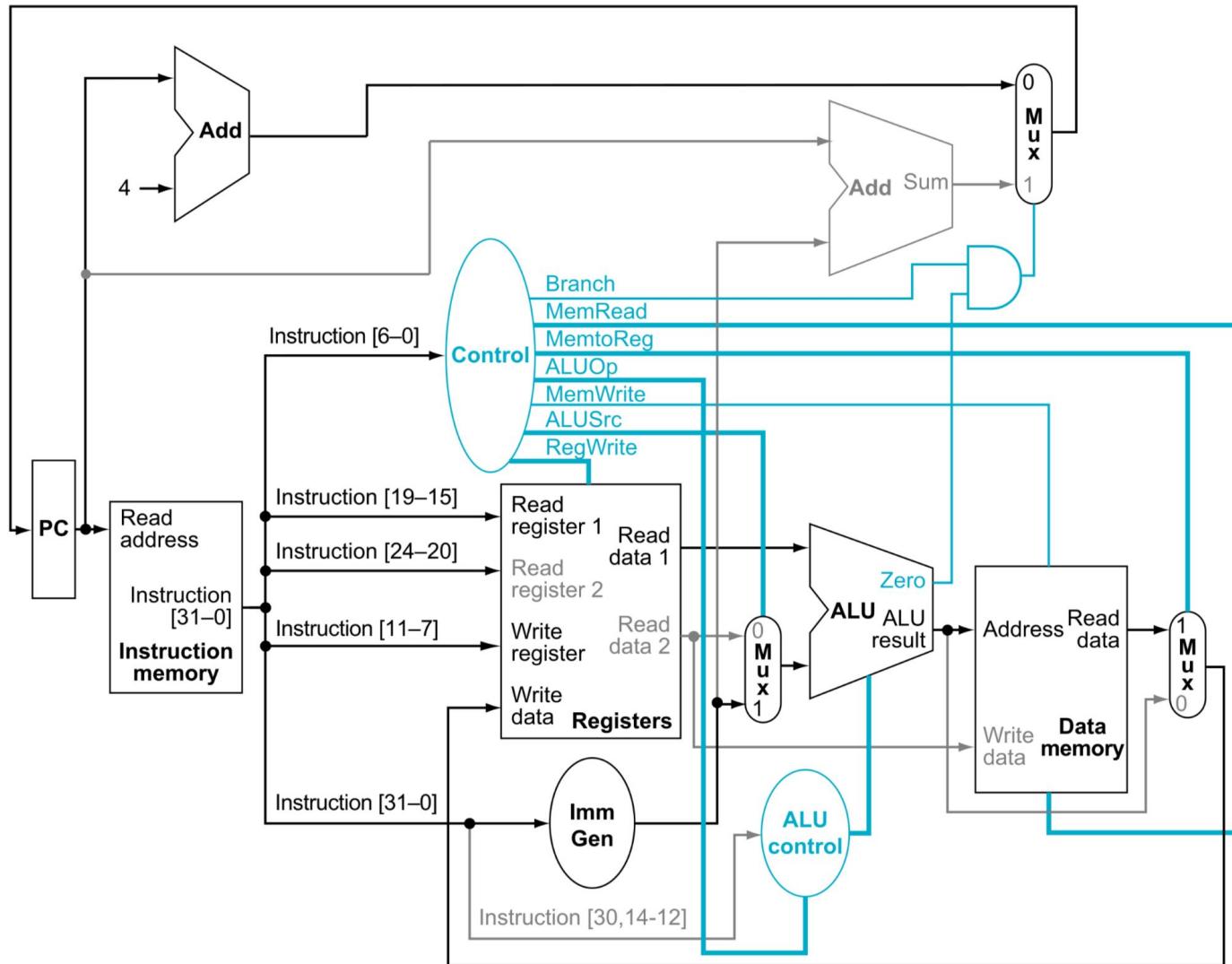
# Datapath With Control<sub>4</sub>

Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

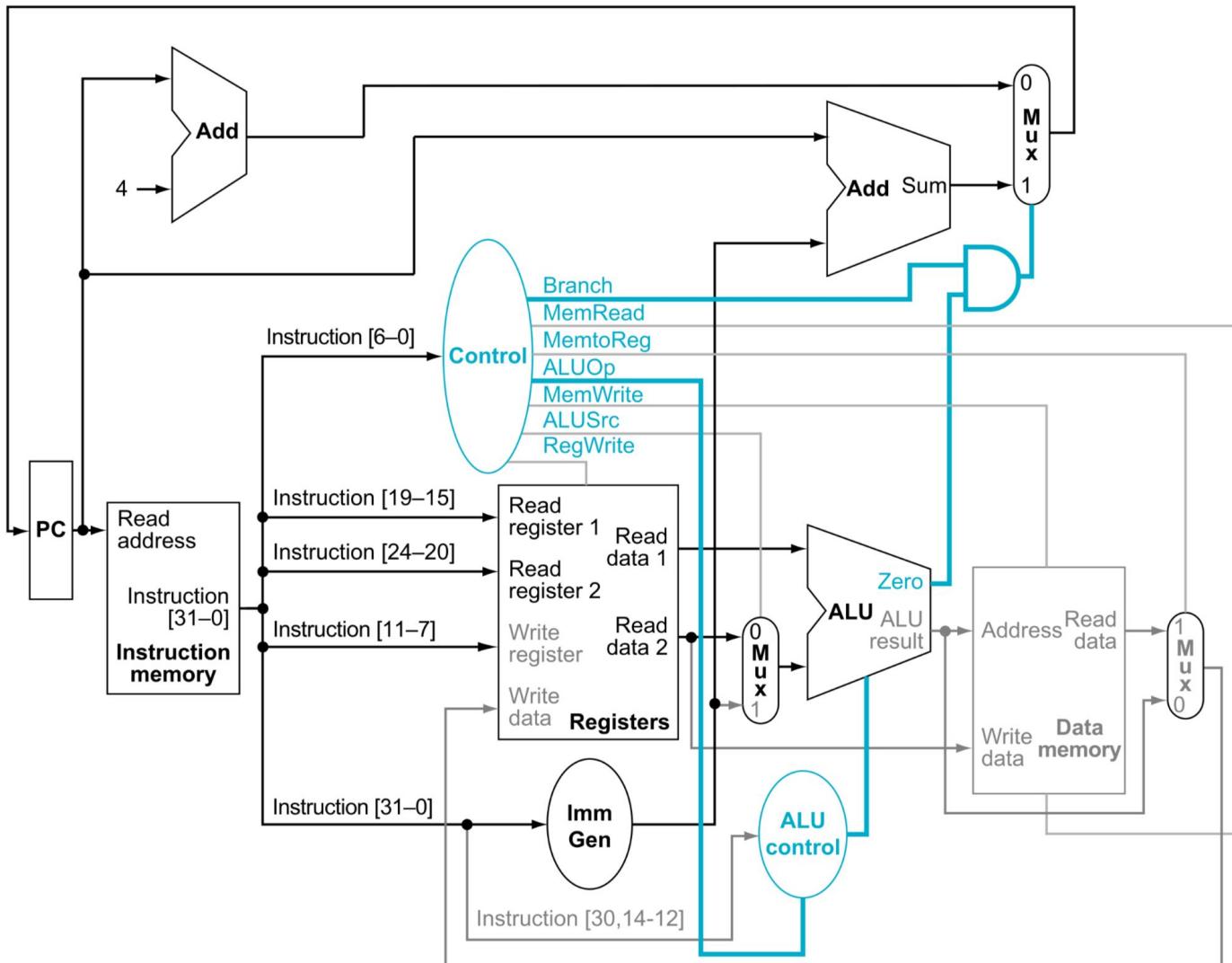
# R-Type Instruction



# Load Instruction



# BEQ Instruction

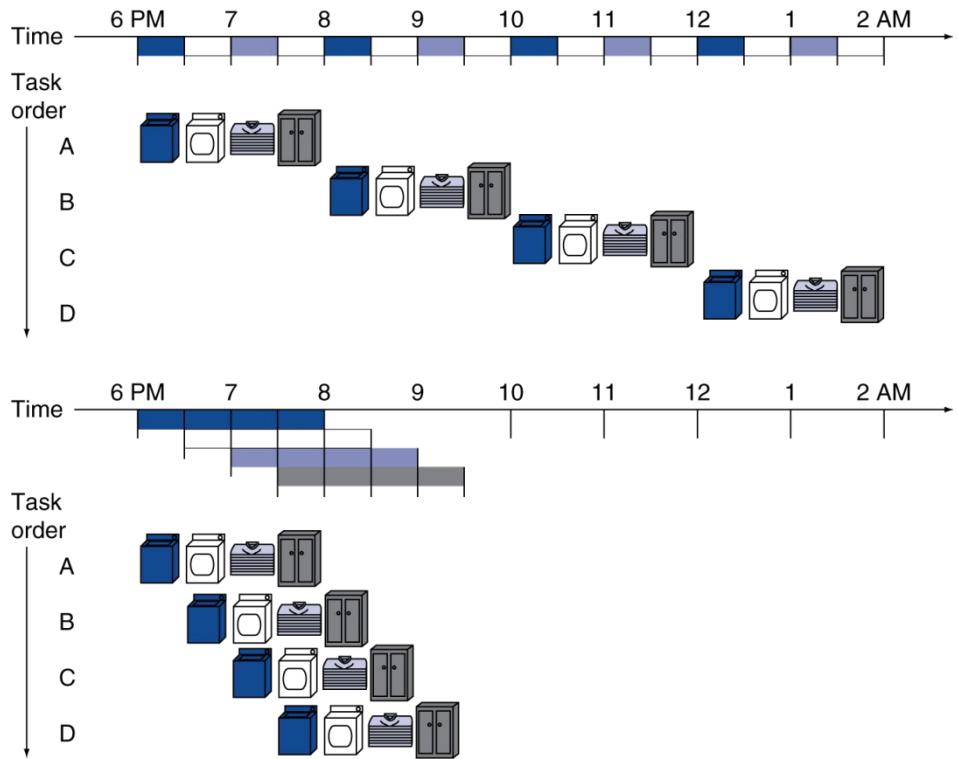


# Performance Issues

- 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

- 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

# RISC-V 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

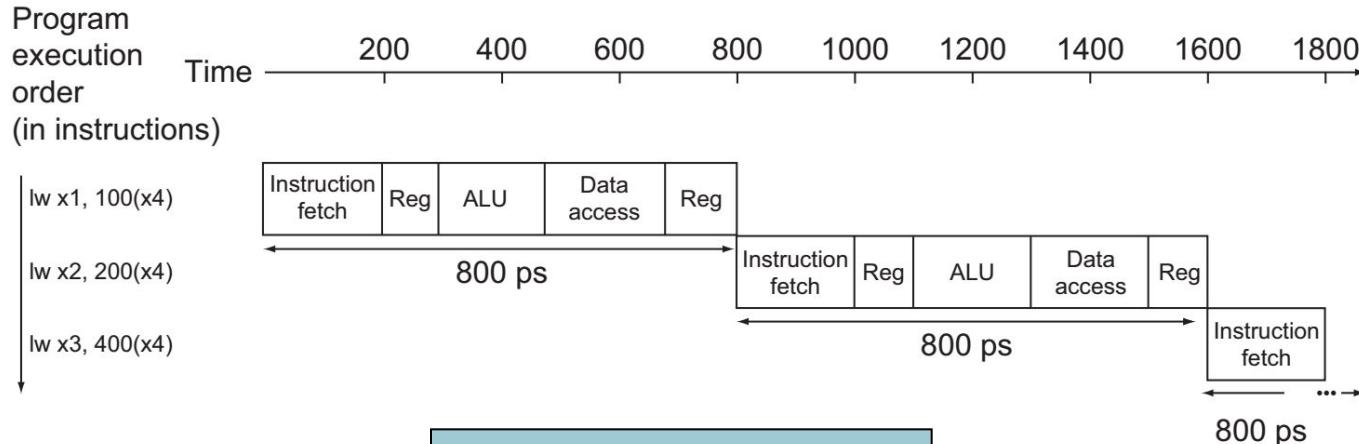
# Pipeline Performance<sub>1</sub>

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

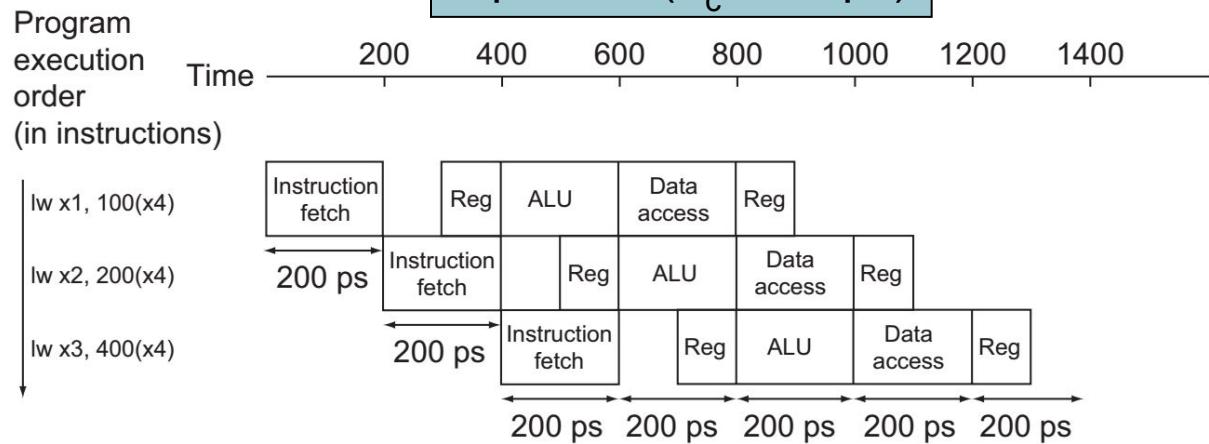
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

# Pipeline Performance<sub>2</sub>

Single-cycle ( $T_c = 800\text{ps}$ )



Pipelined ( $T_c = 200\text{ps}$ )



# 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>  
$$\frac{\text{Number of stages}}{}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

# Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-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

# Hazards

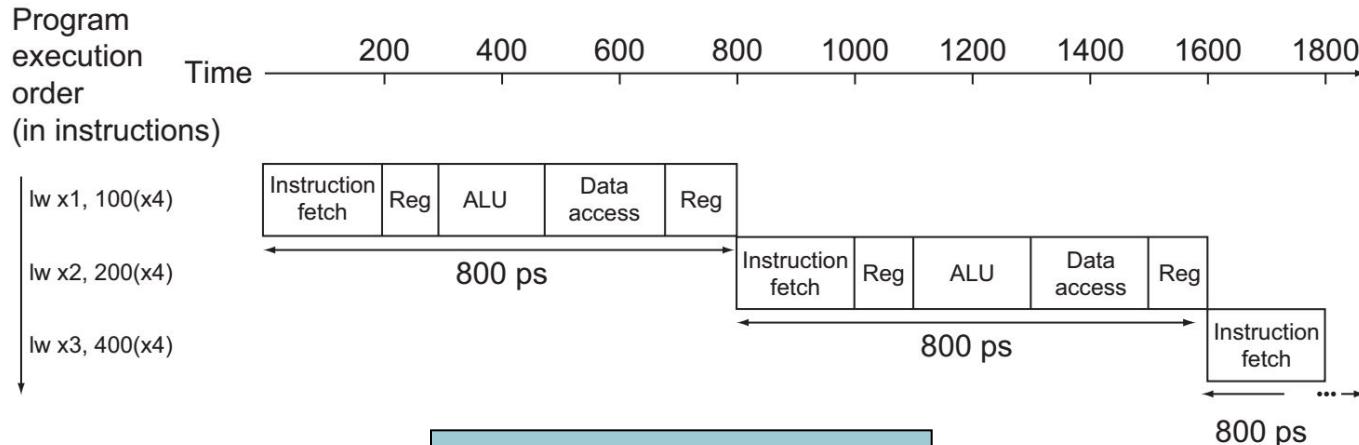
- Situations that prevent starting the next instruction in the next cycle
- **Structure hazards**
  - A required resource is busy
- **Data hazard**
  - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
  - Deciding on control action depends on previous instruction

# Structure Hazards

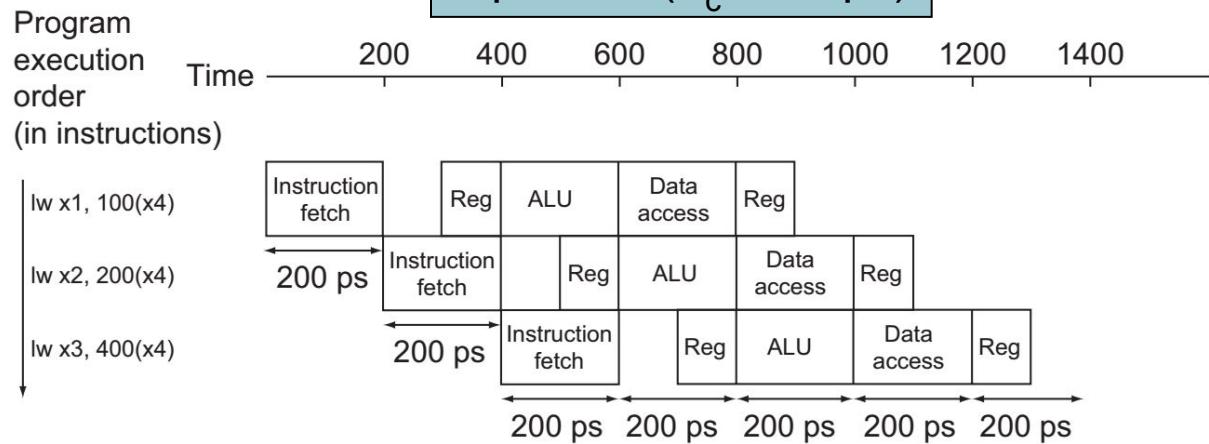
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data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

# Pipeline Performance<sub>2</sub>

Single-cycle ( $T_c = 800\text{ps}$ )

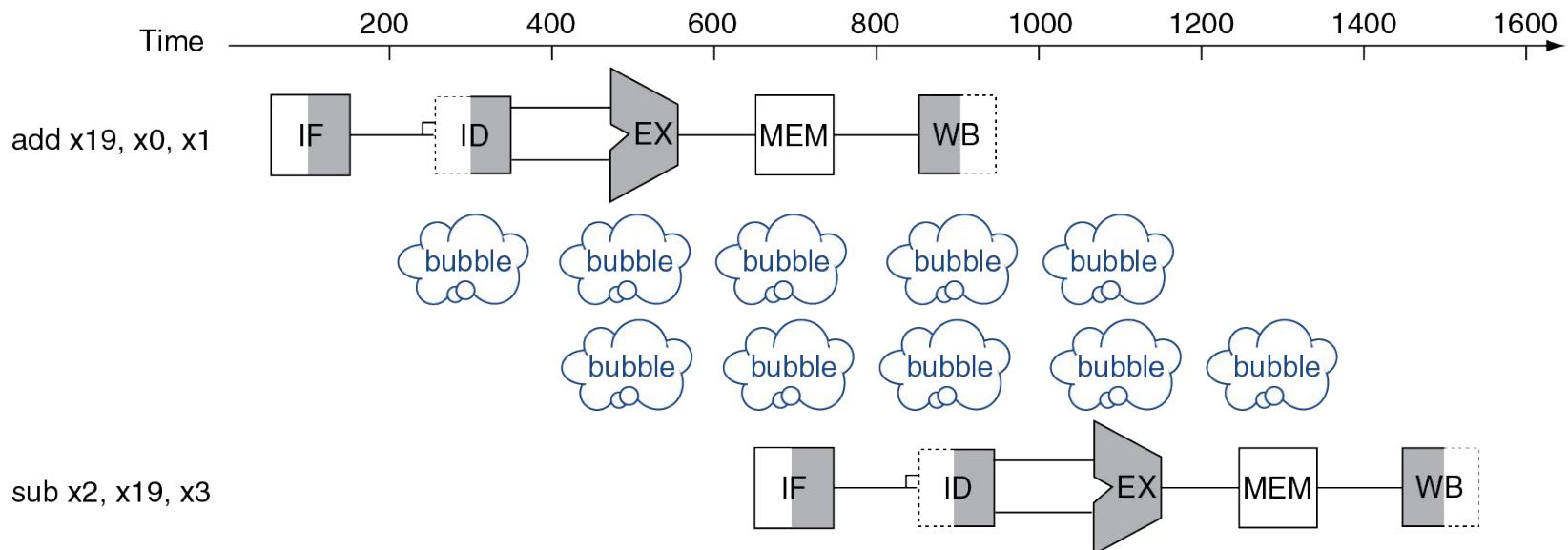


Pipelined ( $T_c = 200\text{ps}$ )



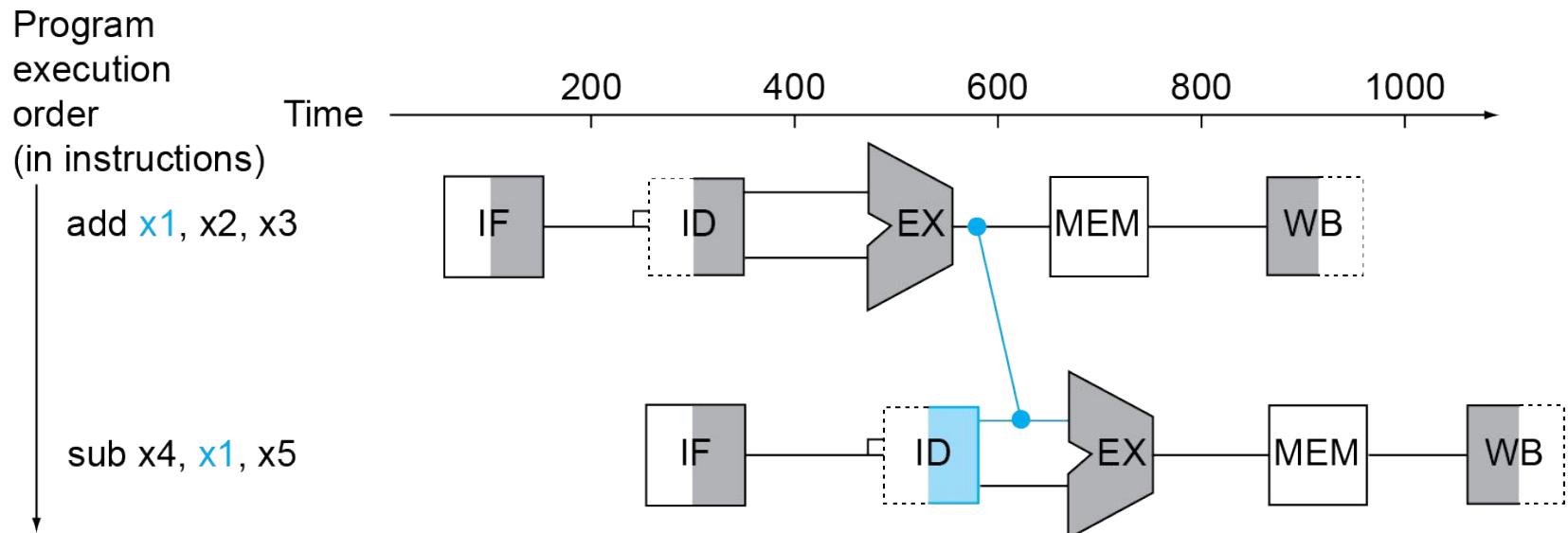
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add **x19**, x0, x1
  - sub x2, **x19**, x3



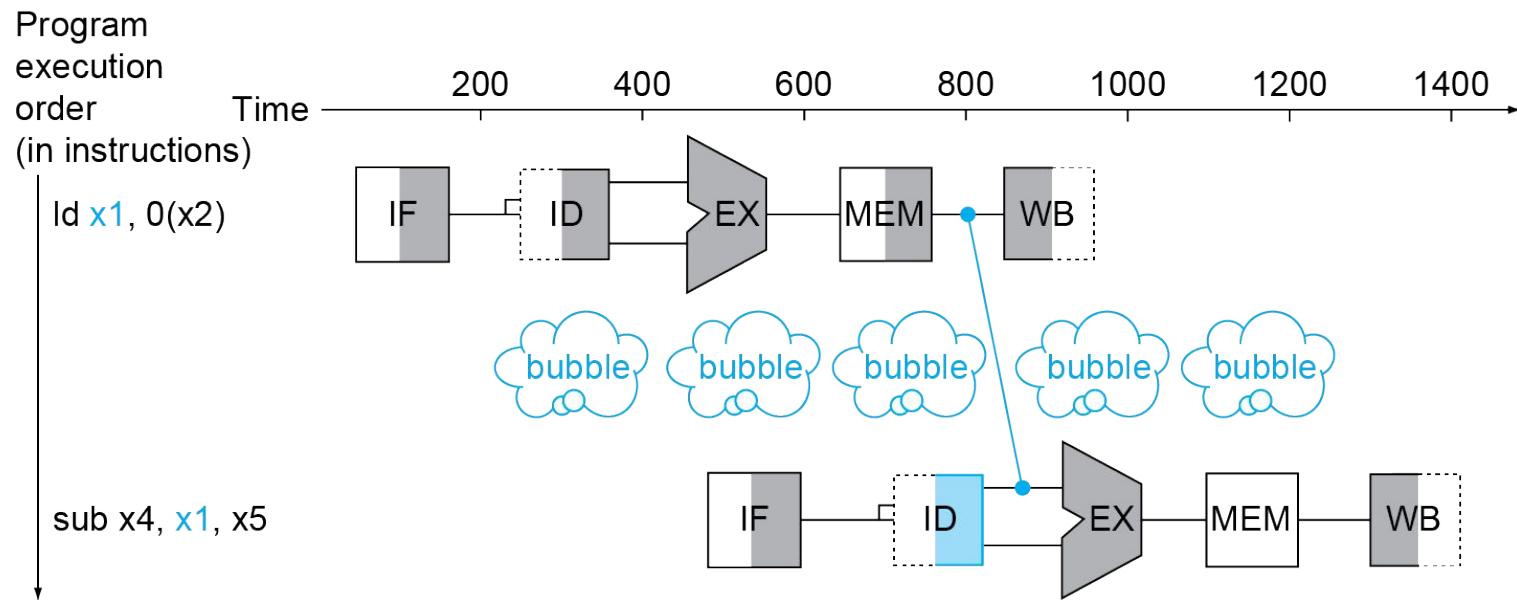
# Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



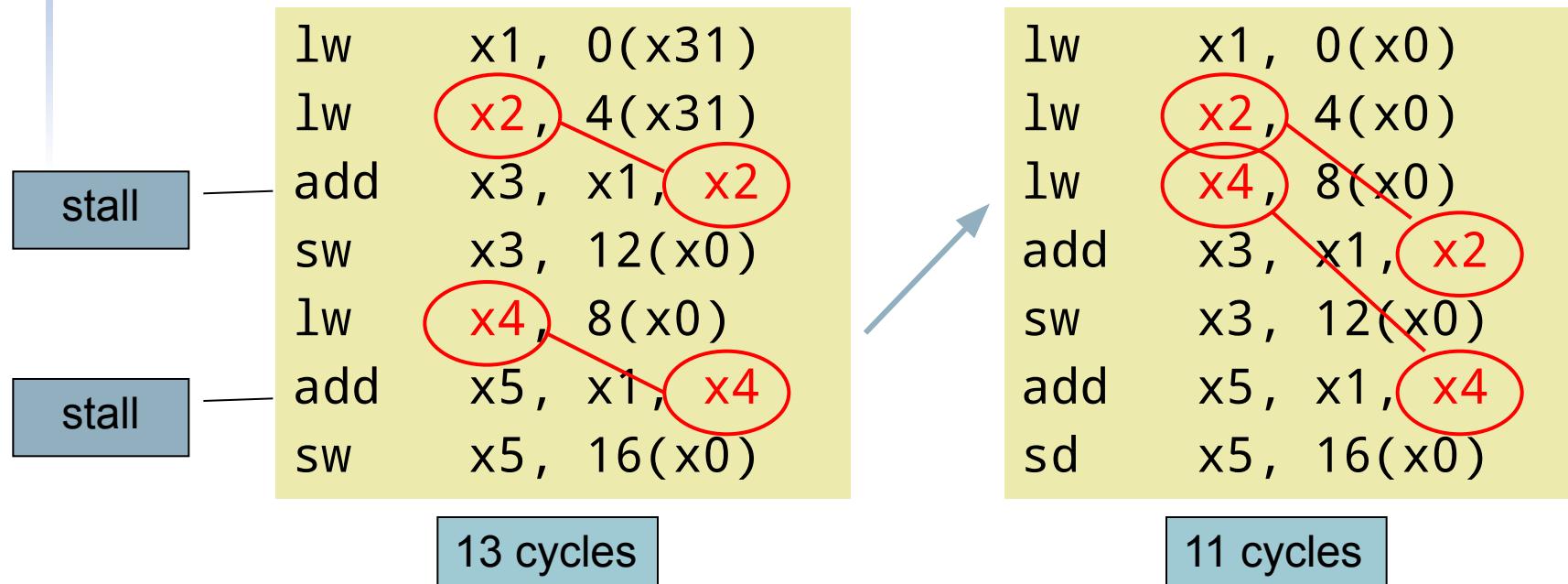
# Load-Use Data Hazard

- 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  $a = b + e; c = b + f;$

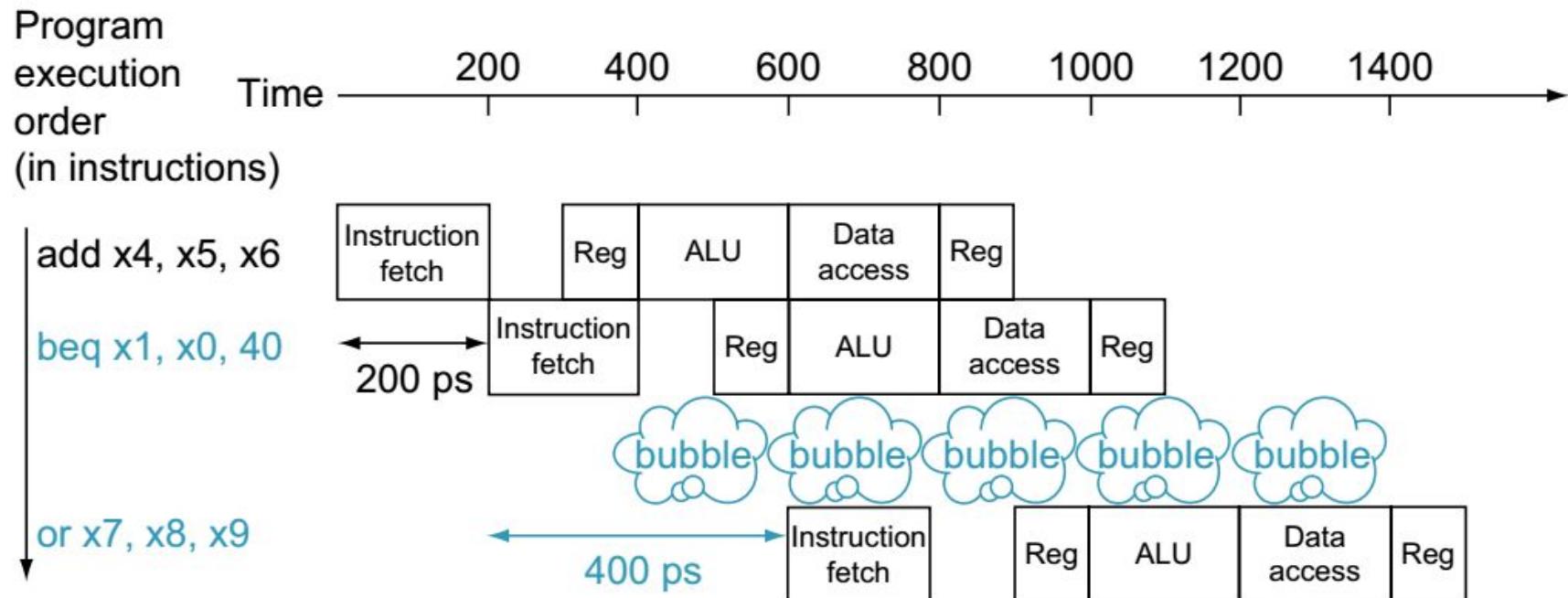


# Control Hazards

- 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 RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

# Stall on Branch

- Wait until branch outcome determined before fetching next instruction

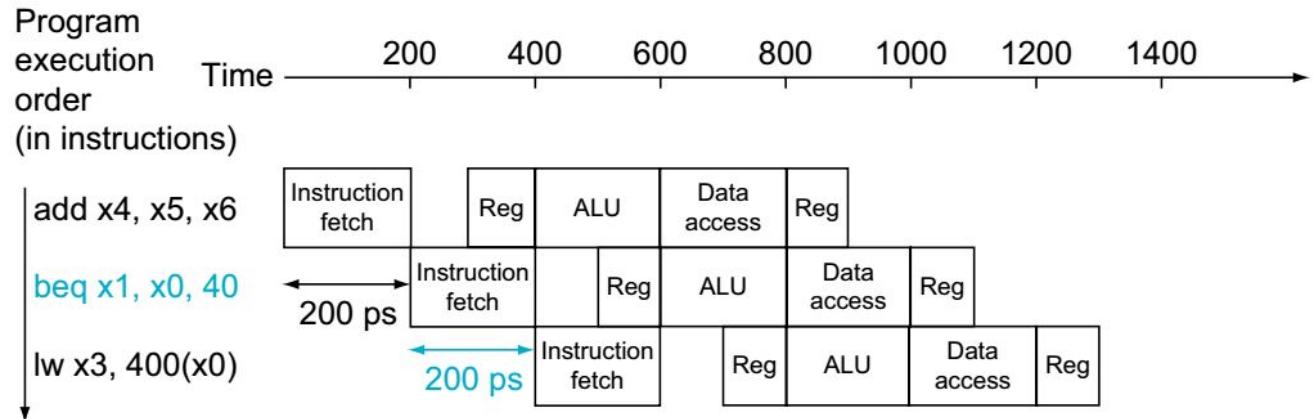


# Branch Prediction

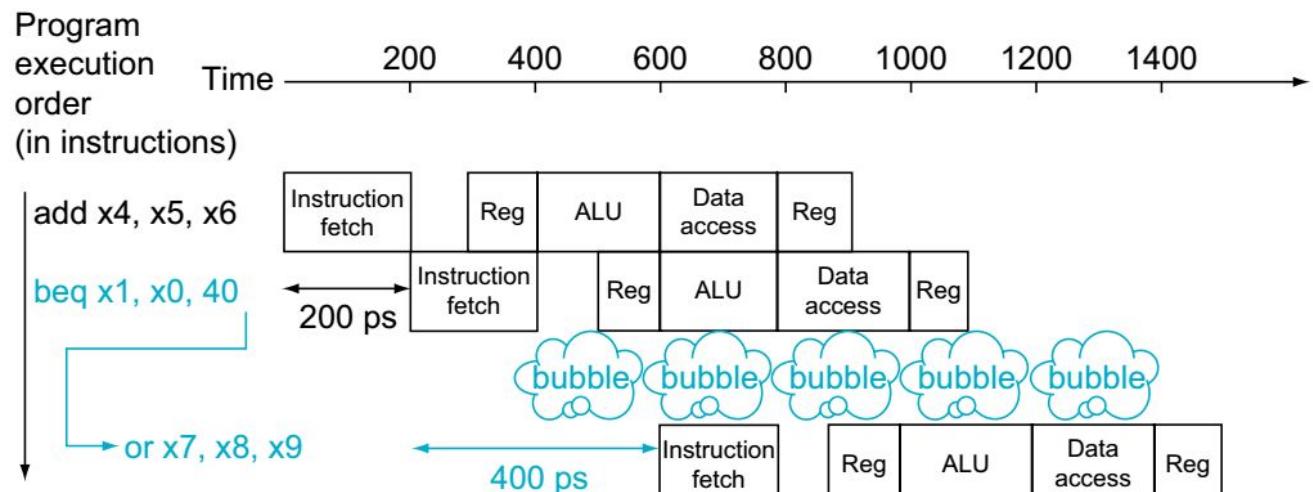
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# RISC-V with Predict Not Taken

Prediction correct



Prediction Incorrect



# More-Realistic Branch Prediction

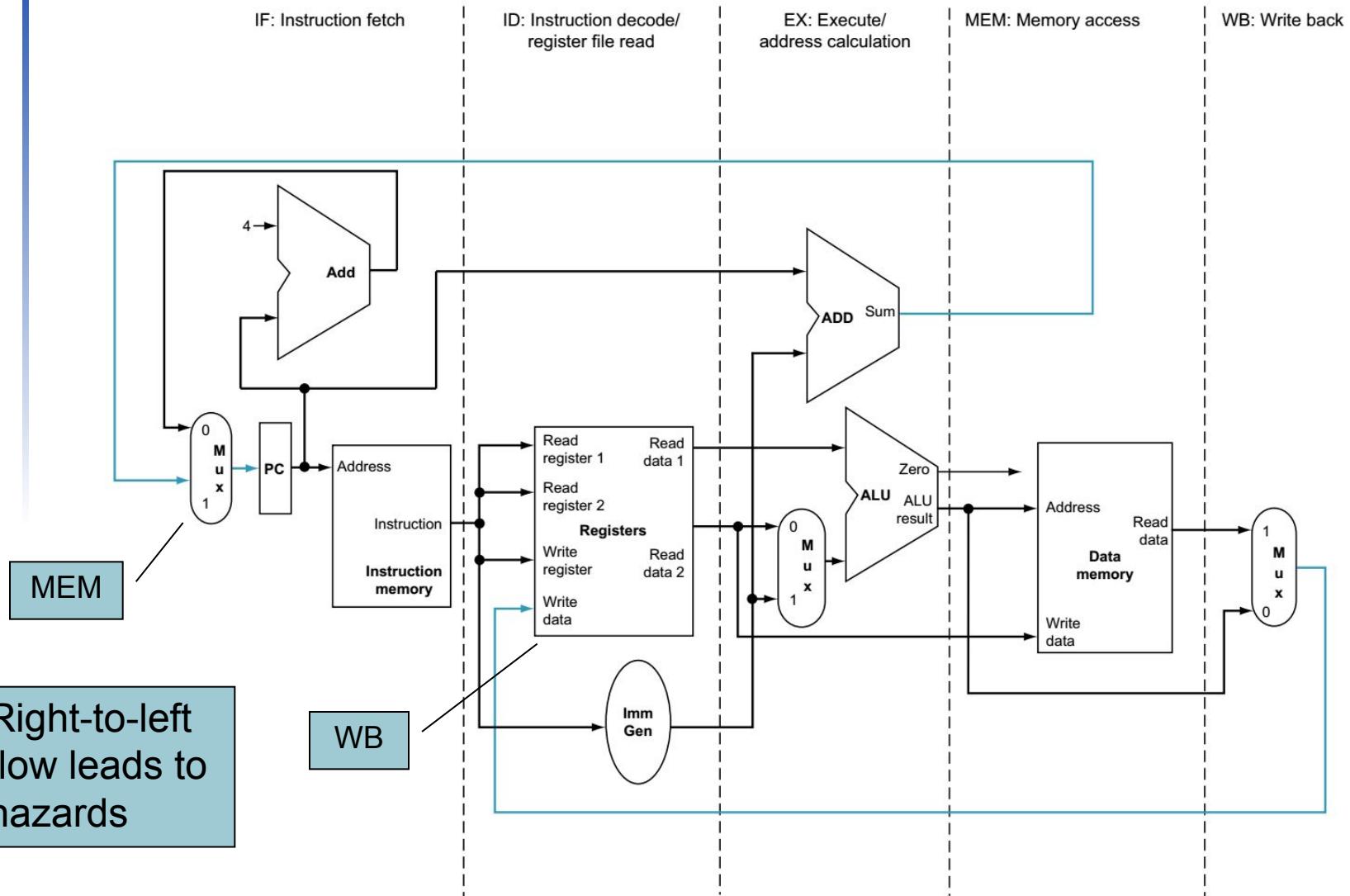
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each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

# Pipeline Summary

## The BIG Picture

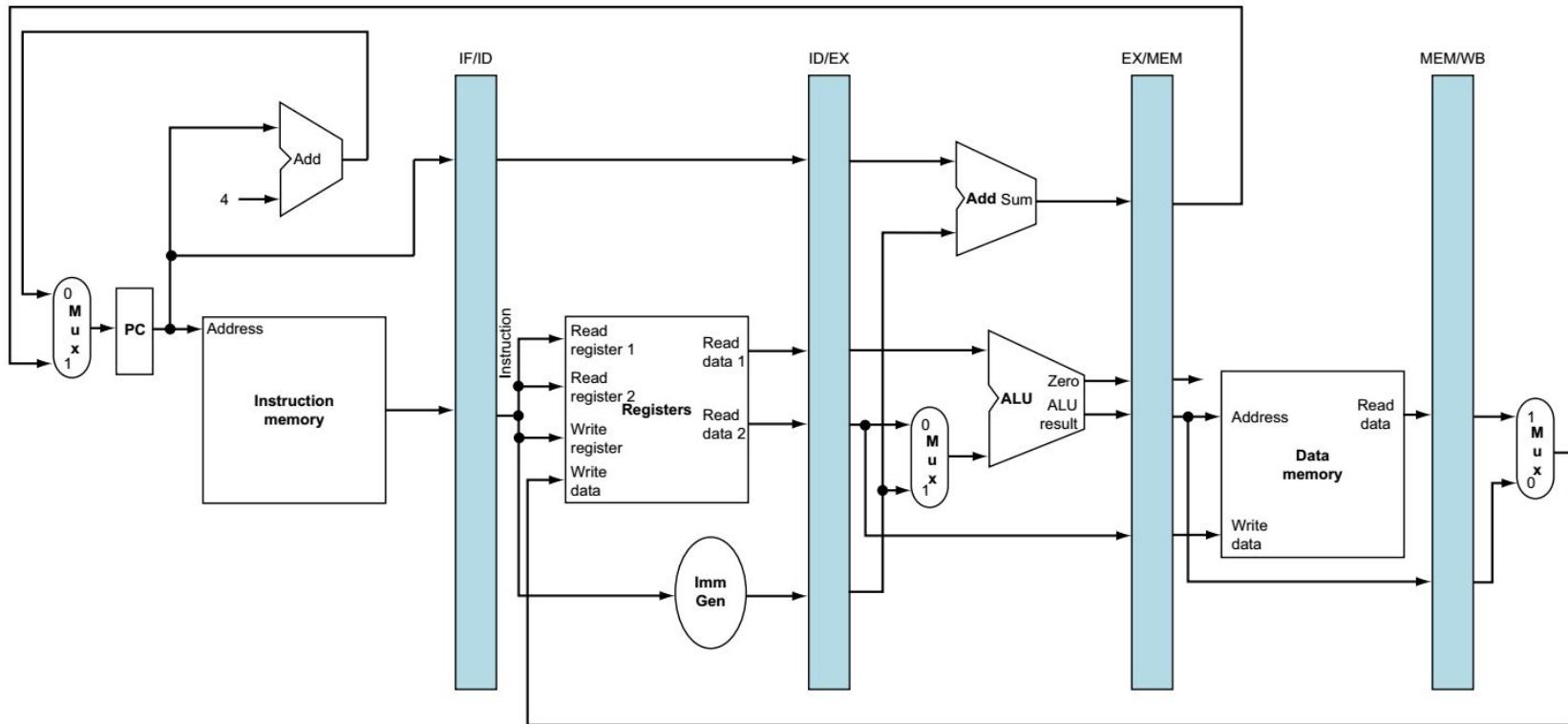
- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

# RISC-V Pipelined Datapath



# Pipeline registers

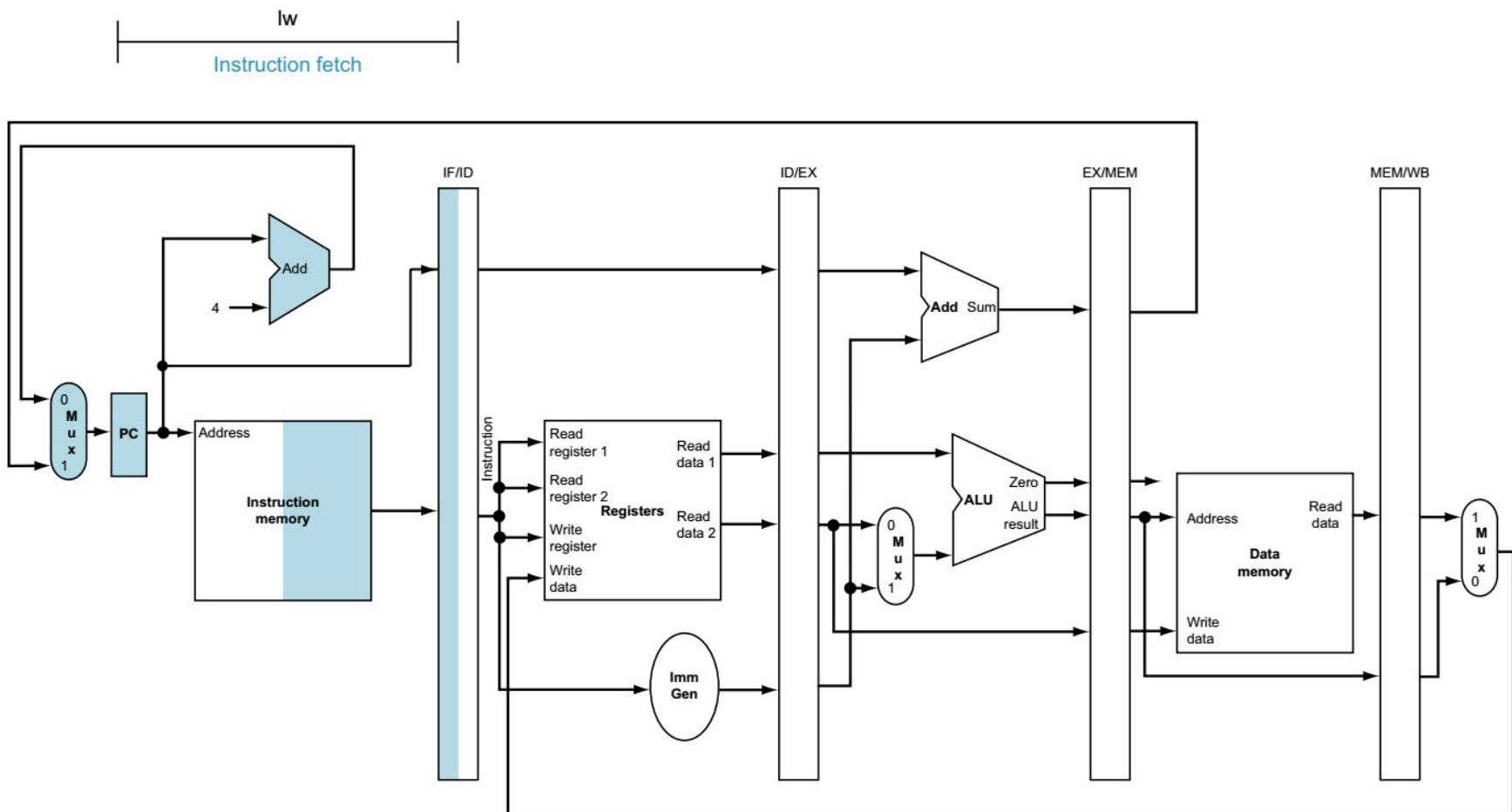
- Need registers between stages
  - To hold information produced in previous cycle



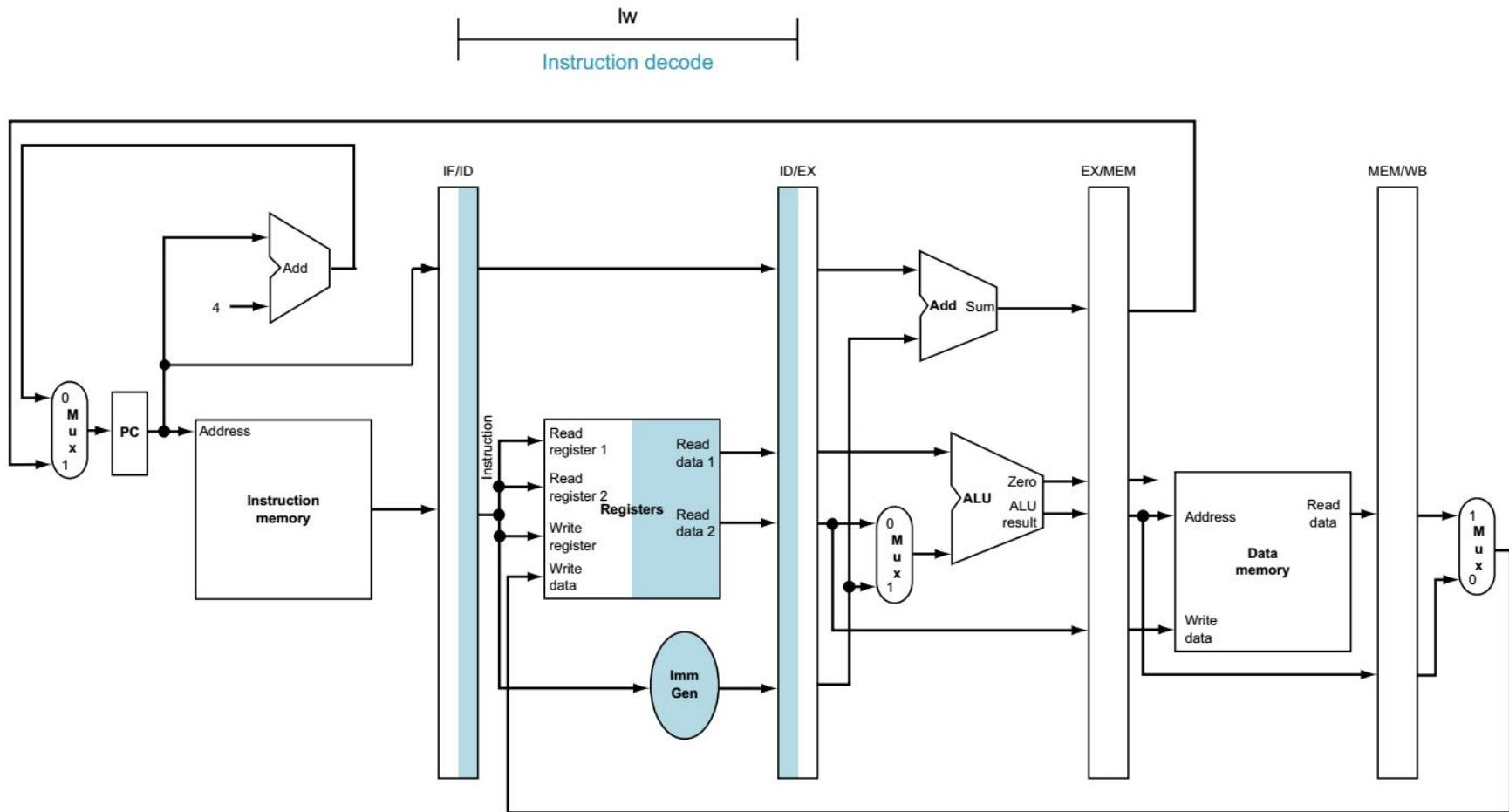
# Pipeline Operations

- 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 operations over time
- We’ll look at “single-clock-cycle” diagrams for load & store

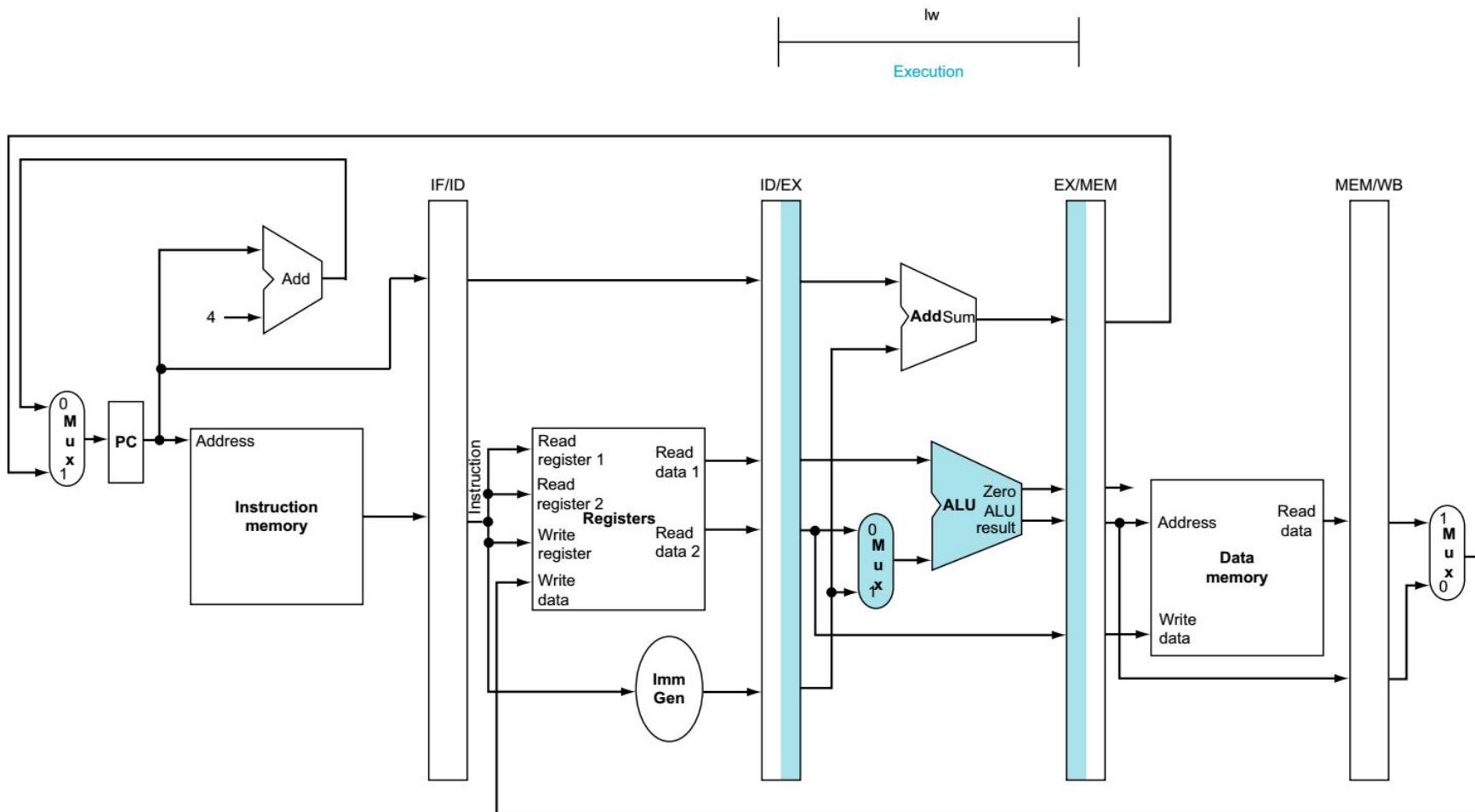
# IF for Load, Store, ...



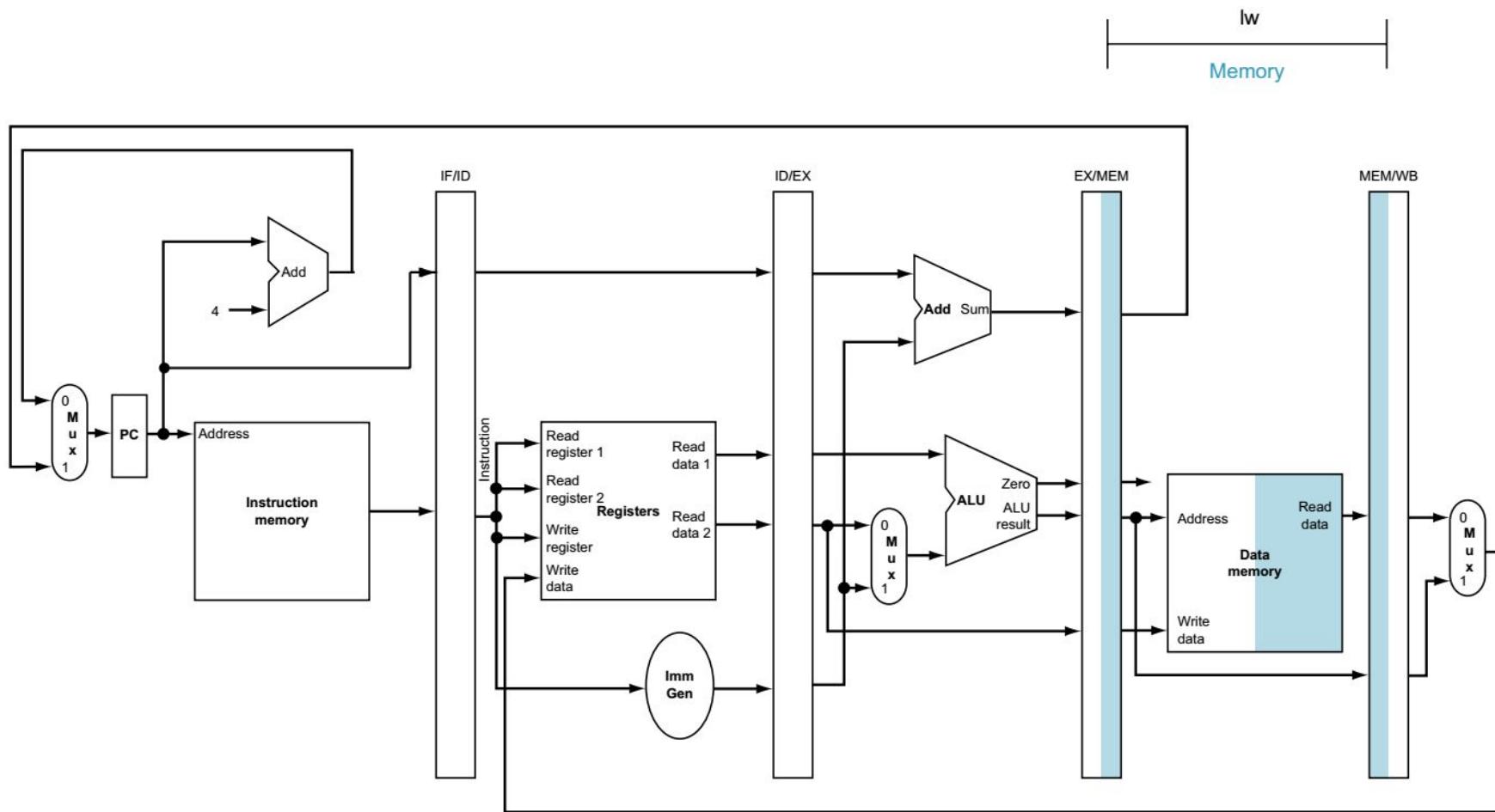
# ID for Load, Store, ...



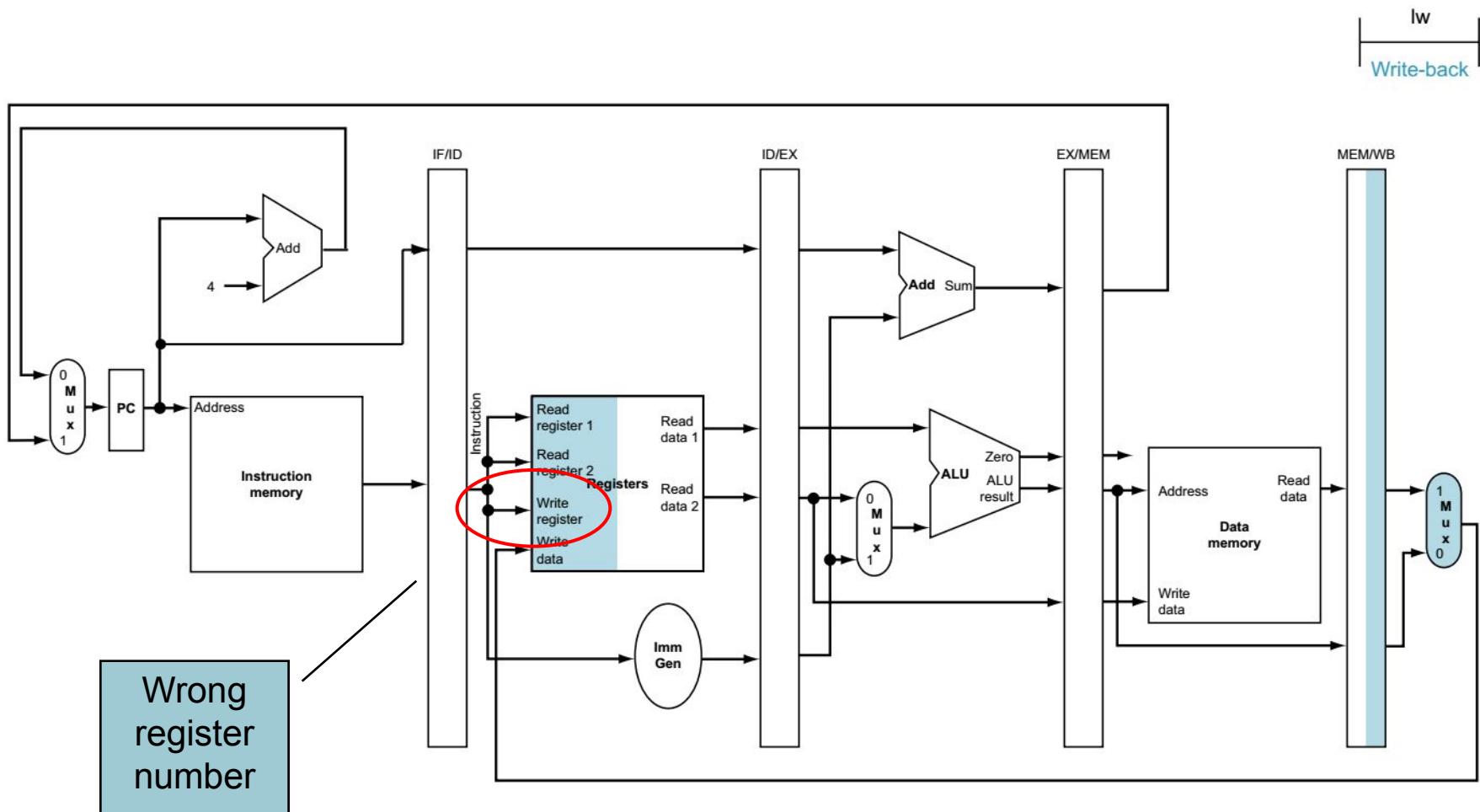
# EX for Load



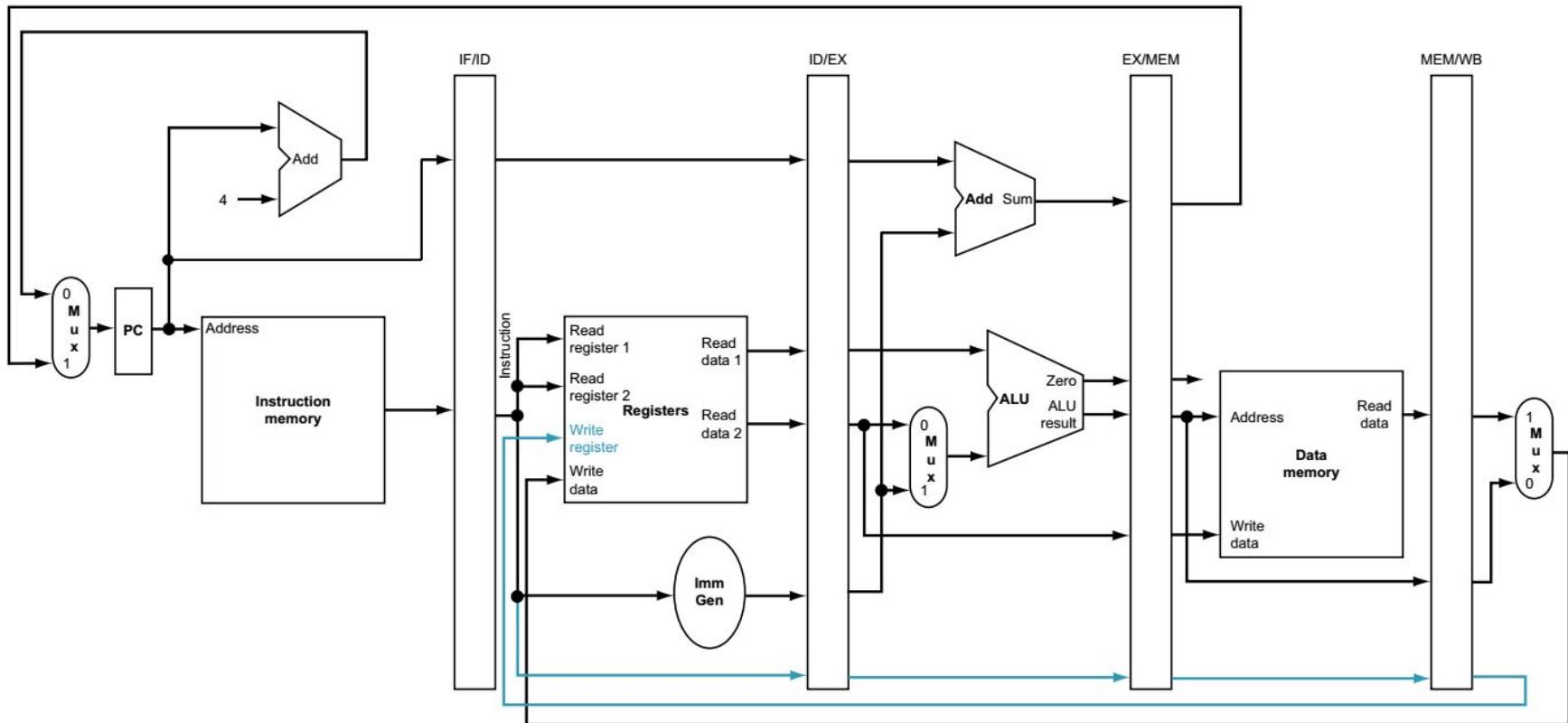
# MEM for Load



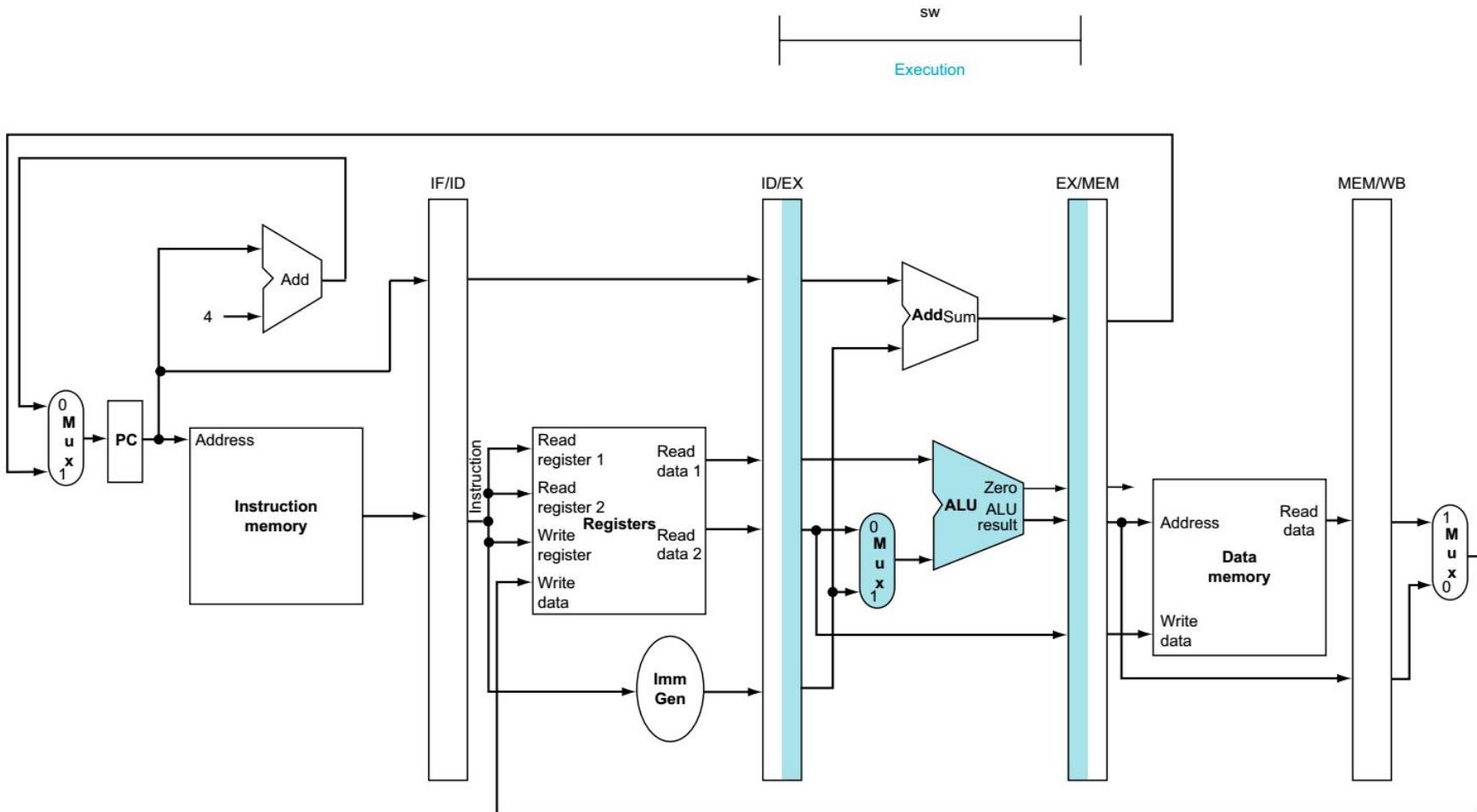
# WB for Load



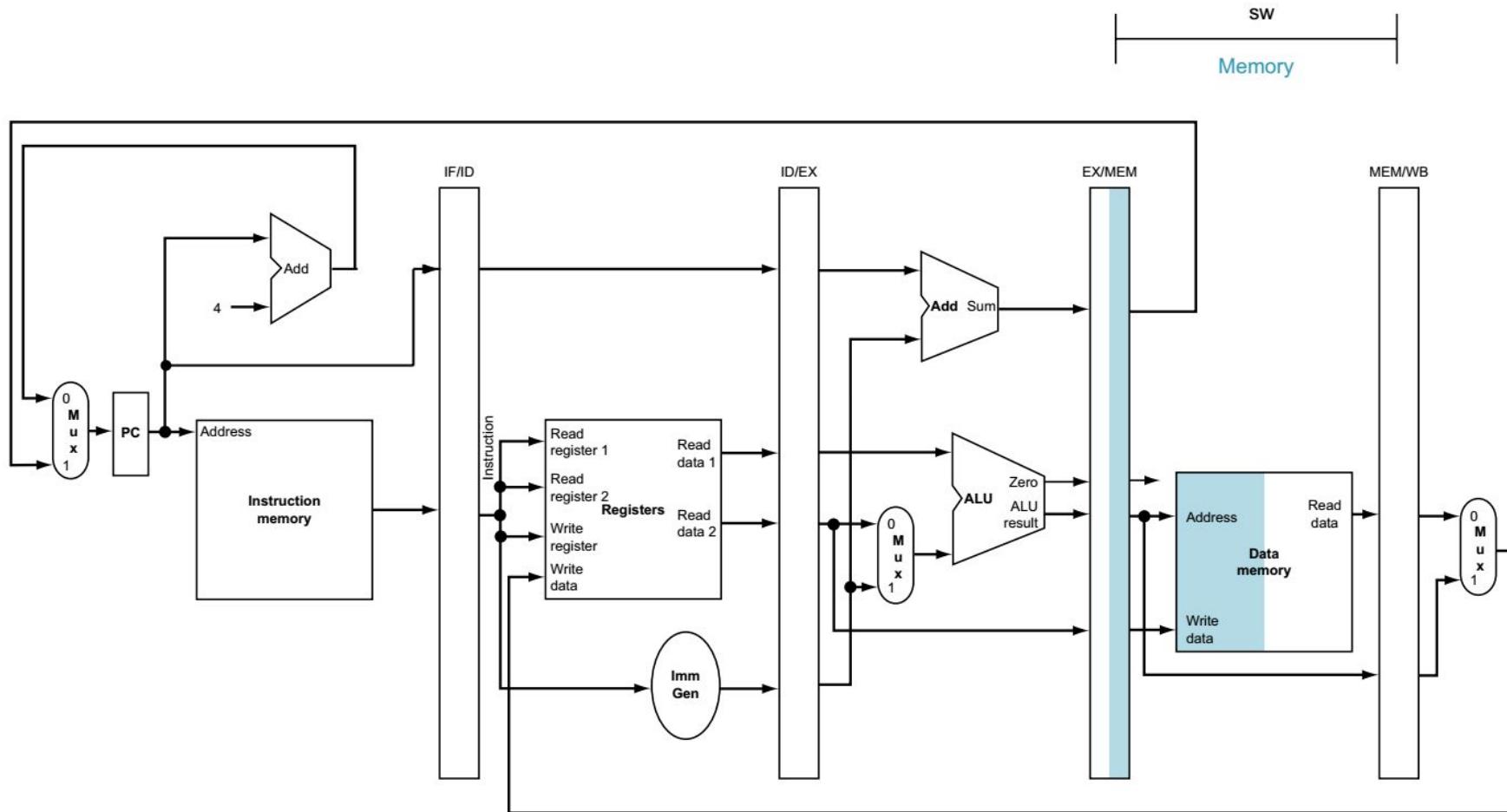
# Corrected Datapath for Load



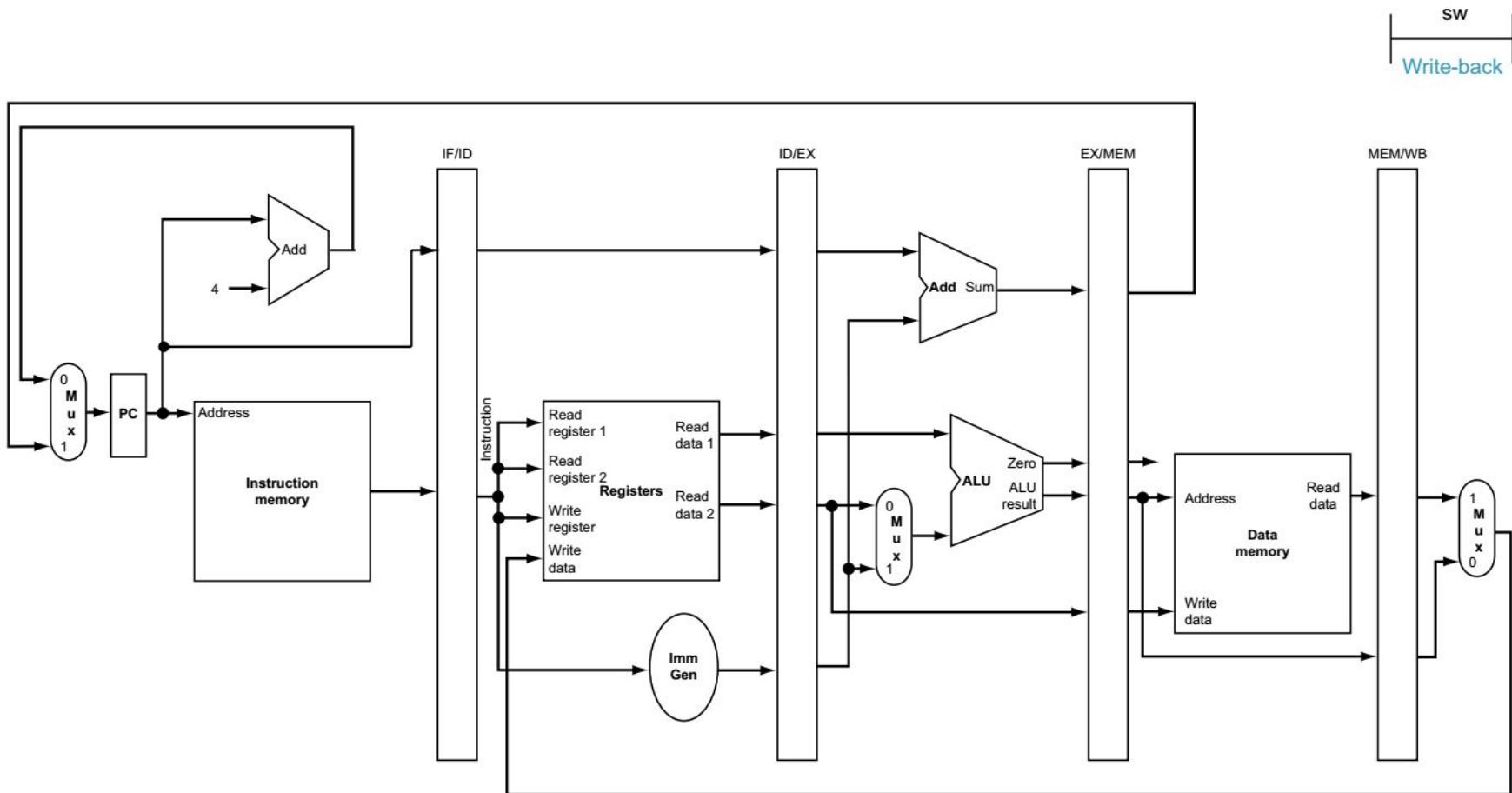
# EX for Store



# MEM for Store

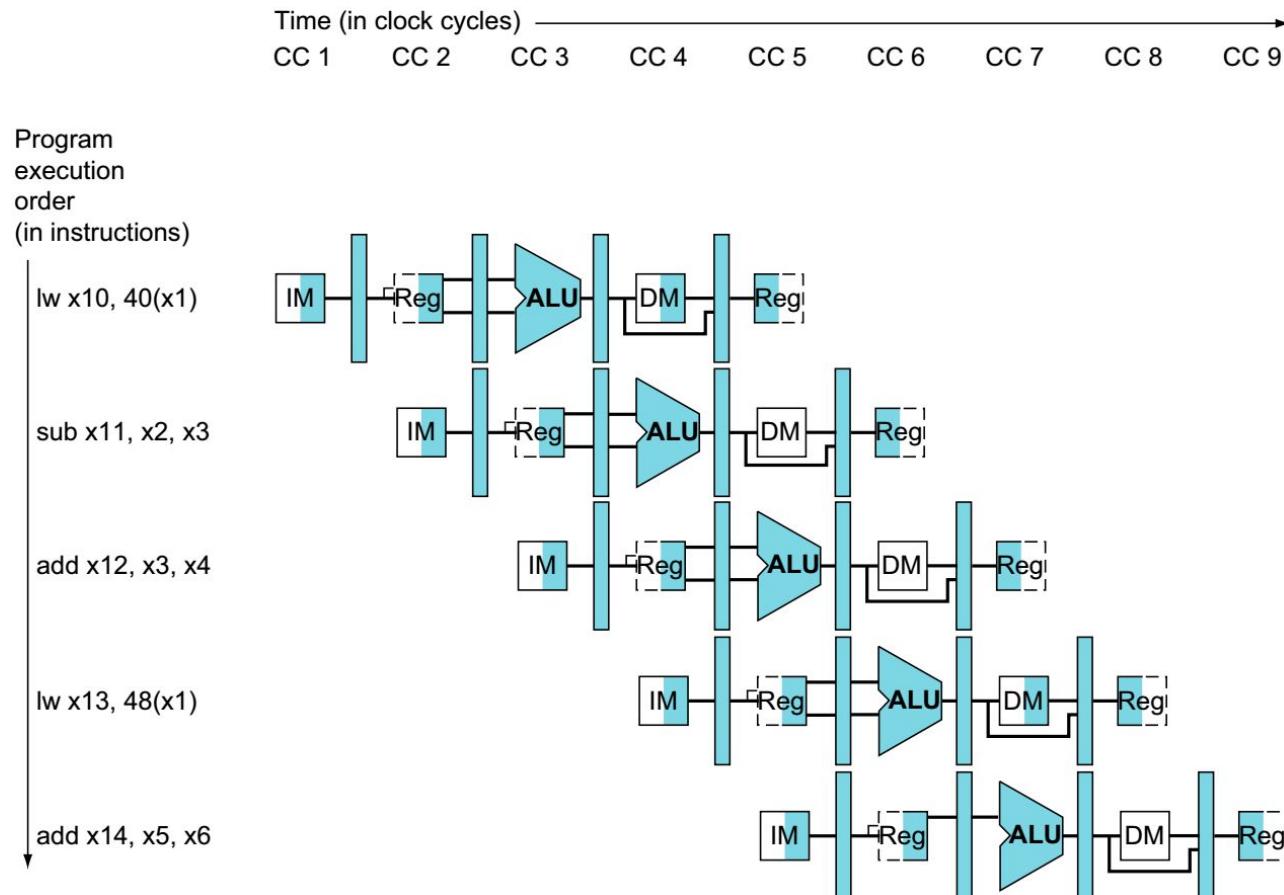


# WB for Store



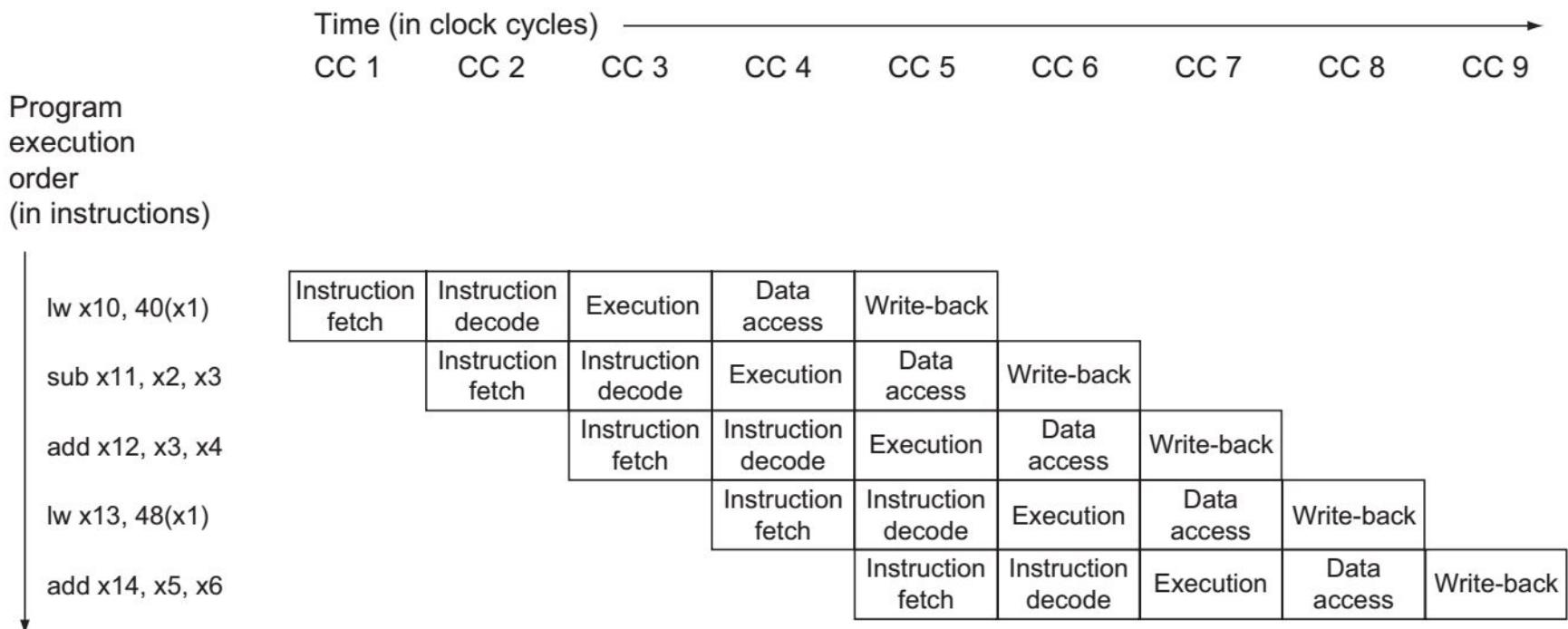
# Multi-Cycle Pipeline Diagram

## Form showing resource usage



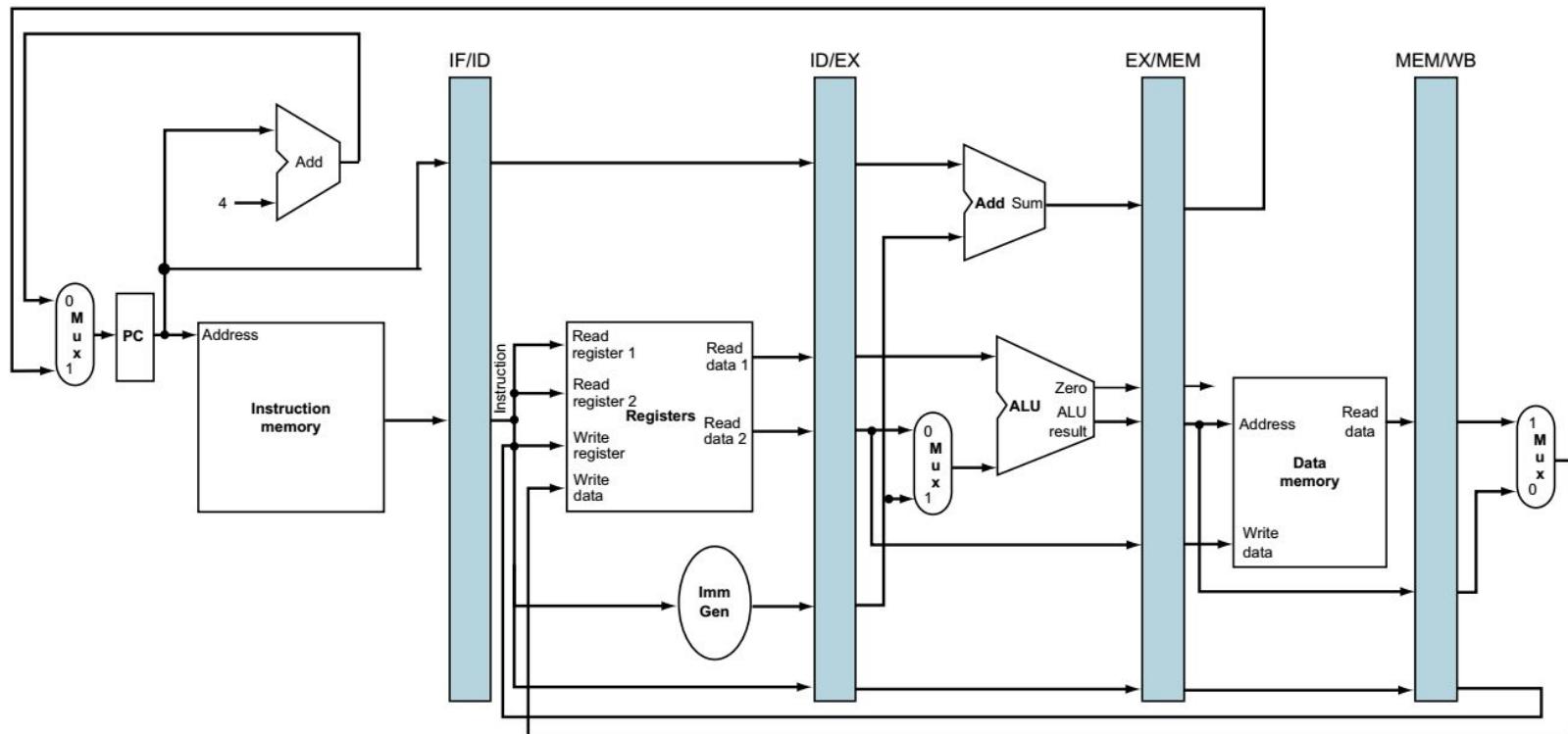
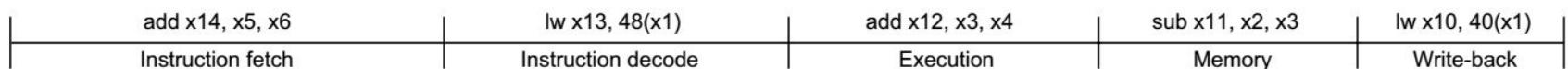
# Multi-Cycle Pipeline Diagram

## Traditional form

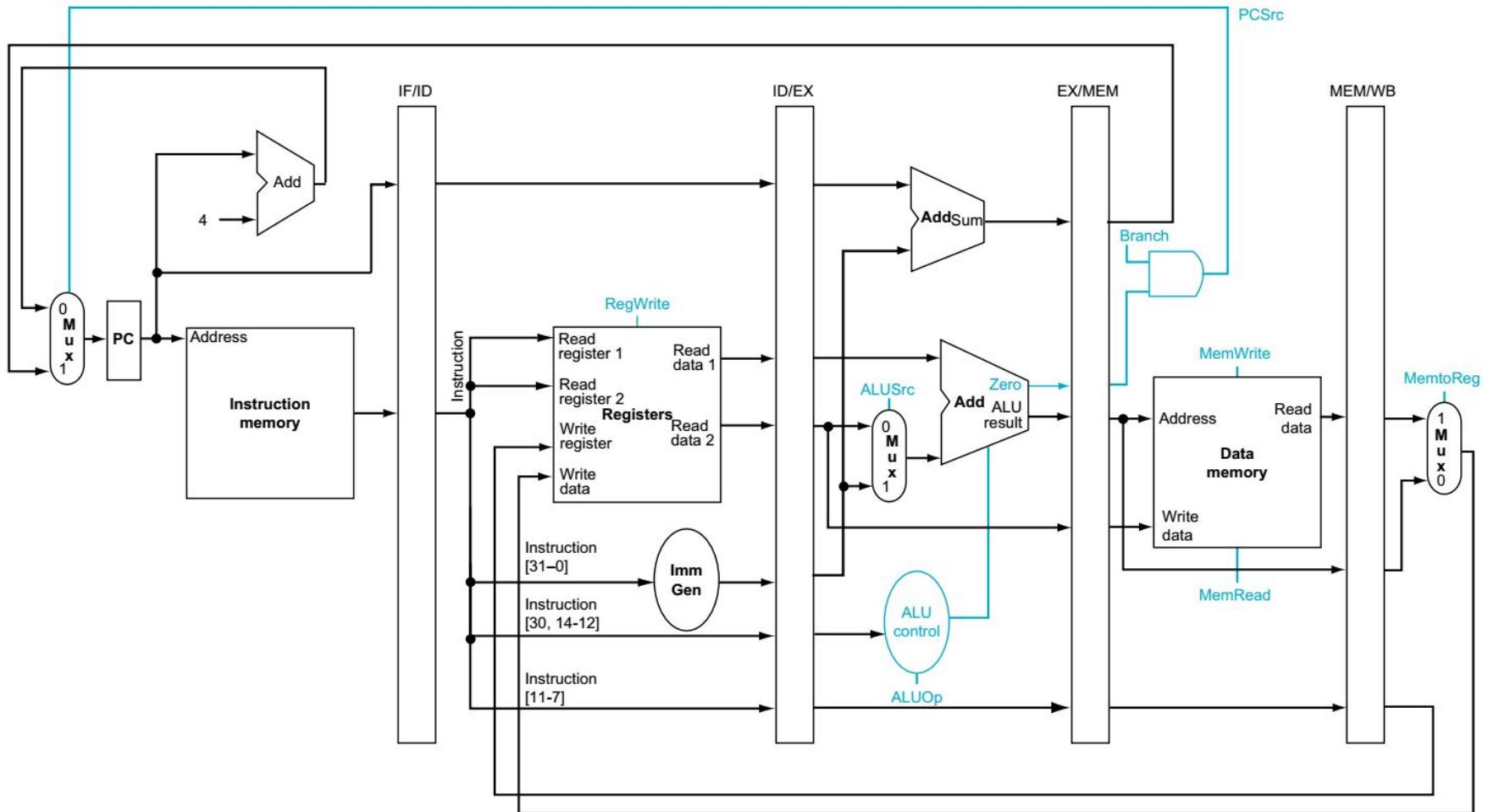


# Single-Cycle Pipeline Diagram

## State of pipeline in a given cycle



# Pipelined Control (Simplified)



# Pipelined Control<sub>1</sub>

<b>Instruction</b>	<b>ALUOp</b>	<b>operation</b>	<b>Funct7 field</b>	<b>Funct3 field</b>	<b>Desired ALU action</b>	<b>ALU control input</b>
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

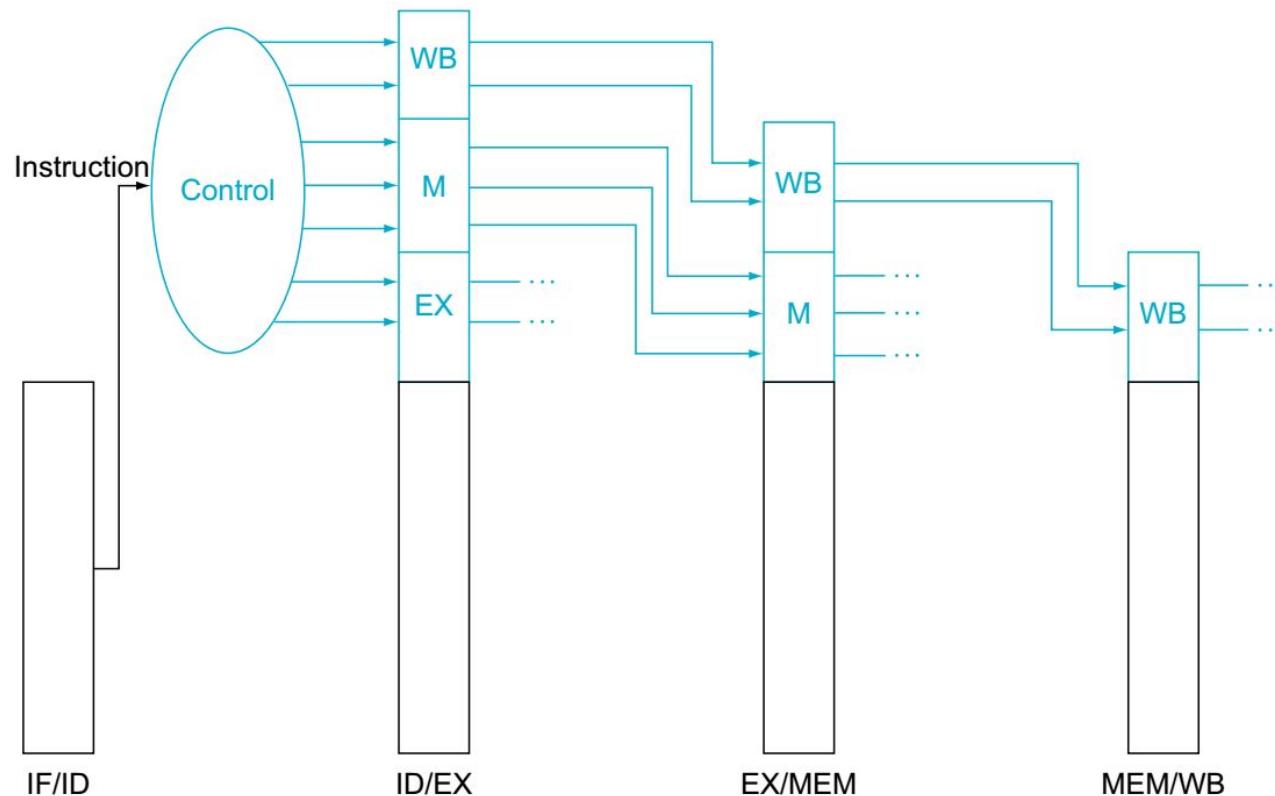
<b>Signal name</b>	<b>Effect when deasserted</b>	<b>Effect when asserted</b>
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 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.

# Pipelined Control<sub>2</sub>

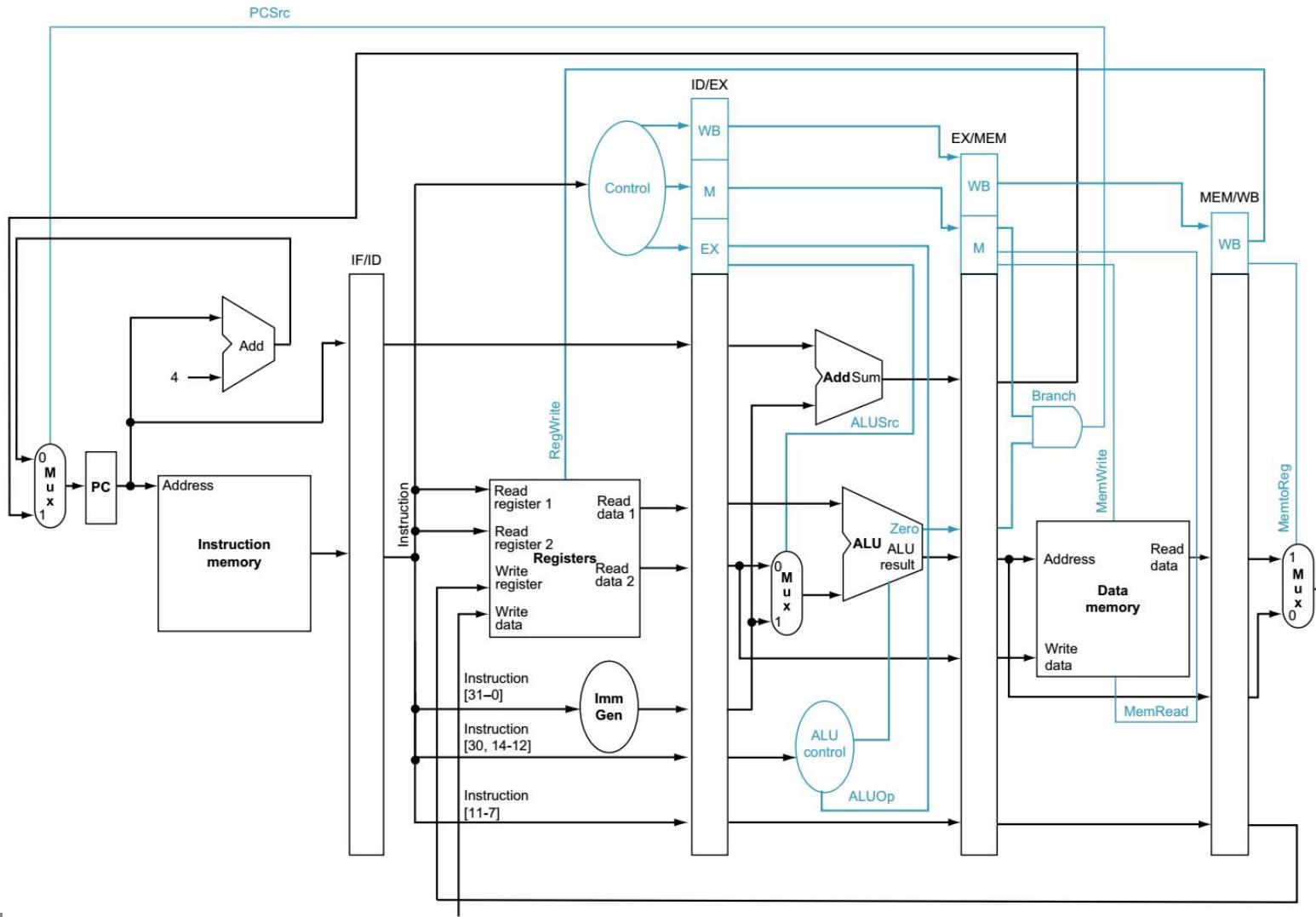
Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem- Read	Mem- Write	Reg- Write	Memto- Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

# Pipelined Control<sub>3</sub>

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



# Pipelined Control<sub>4</sub>



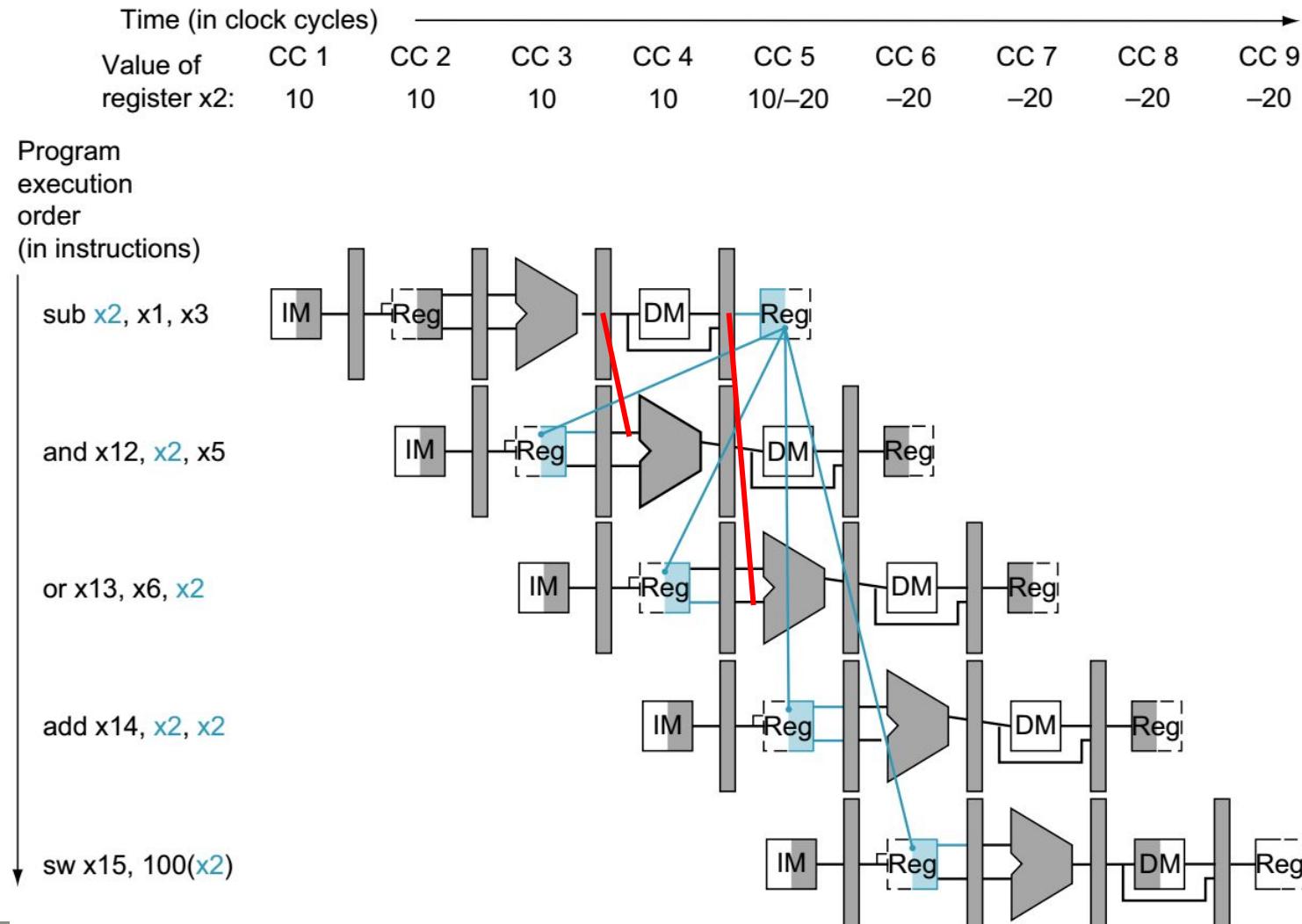
# Data Hazards in ALU Instructions

- Consider this sequence:

```
sub    x2, x1, x3  
and    x12, x2, x5  
or     x13, x6, x2  
add    x14, x2, x2  
sw     x15, 100(x2)
```

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

# Dependencies & Forwarding



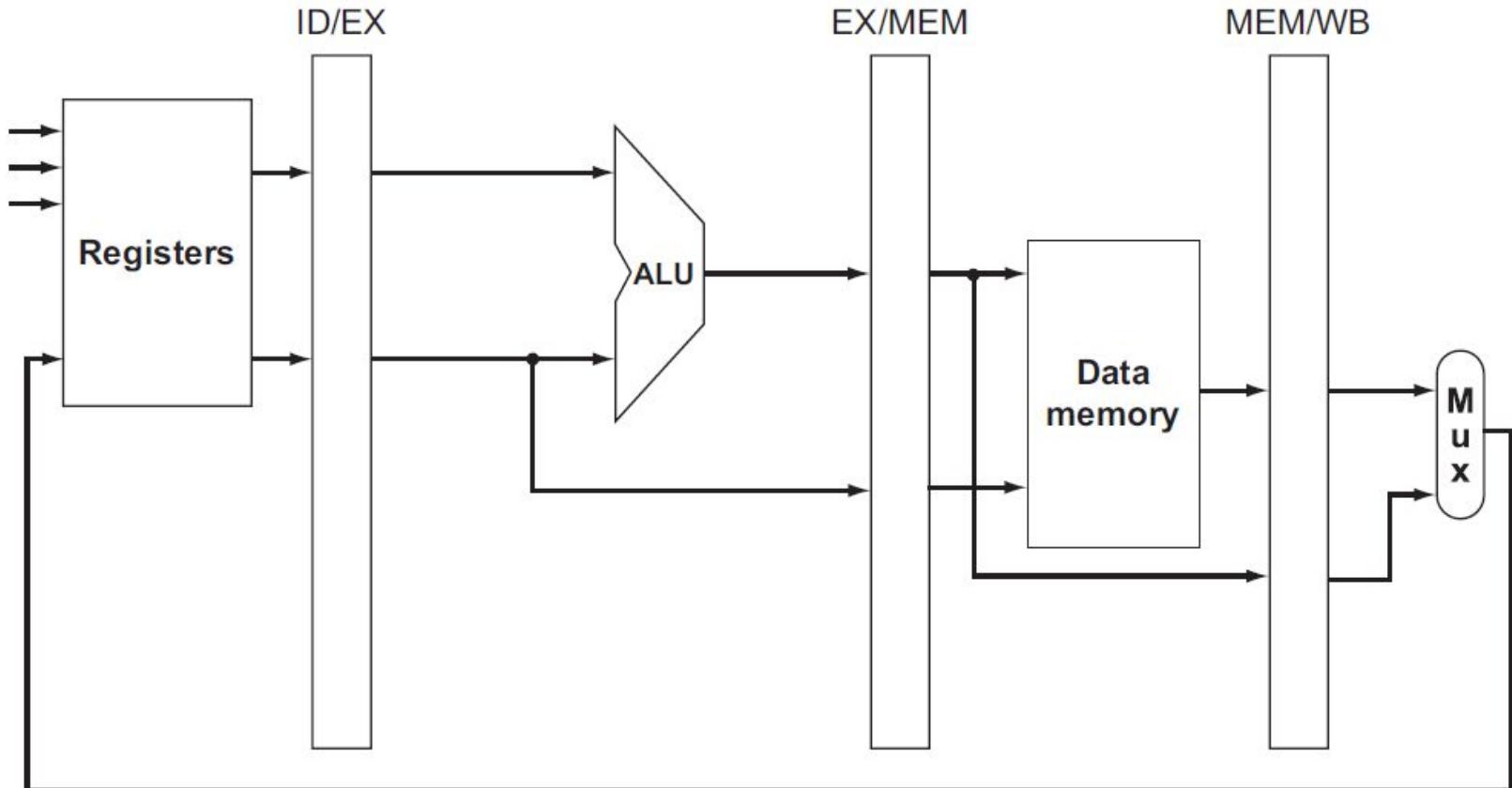
# Detecting the Need to Forward<sub>1</sub>

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

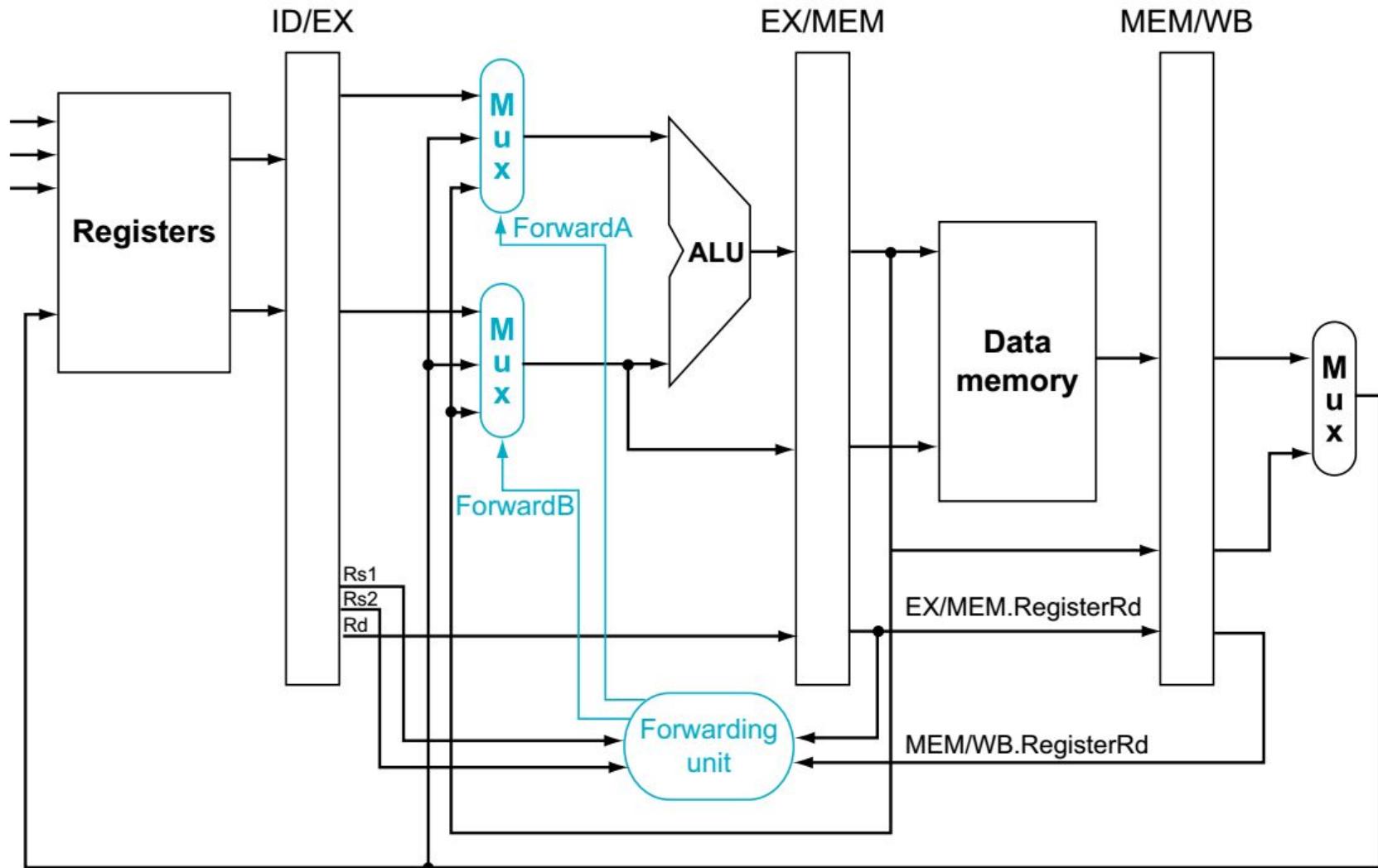
Fwd from  
EX/MEM  
pipeline reg

Fwd from  
MEM/WB  
pipeline reg

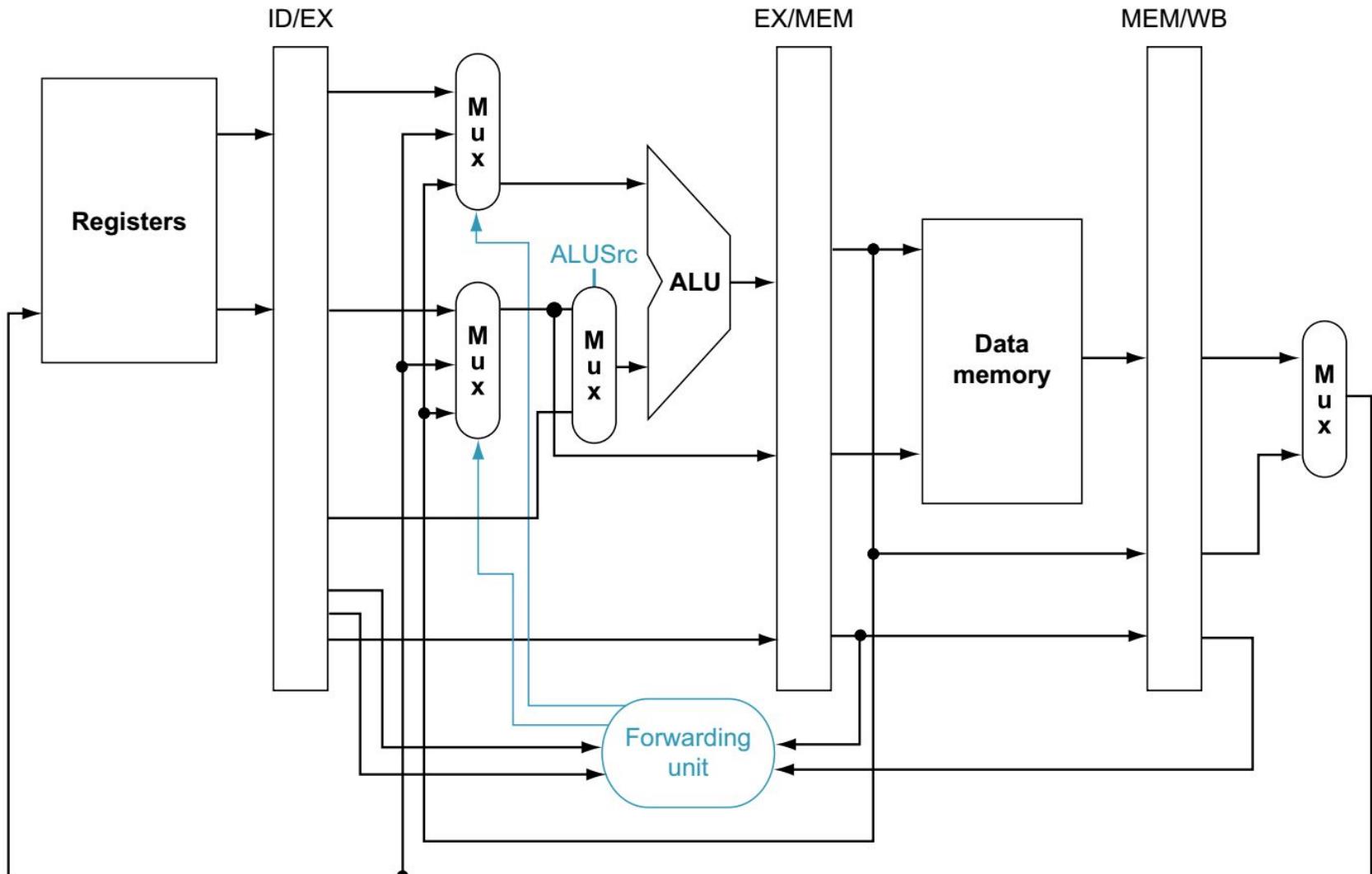
# No Forwarding



# Forwarding Paths<sub>1</sub>



# Forwarding Paths<sub>2</sub>



# Forwarding Conditions

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.

# Detecting the Need to Forward<sub>2</sub>

- 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 x0
  - EX/MEM.RegisterRd  $\neq$  0,  
MEM/WB.RegisterRd  $\neq$  0

# Forwarding Conditions<sub>1</sub>

## ■ EX hazard

- if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))  
ForwardA = 10
- If (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))  
ForwardB = 10

# Forwarding Conditions<sub>2</sub>

- MEM hazard
  - if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd ≠ 0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))  
ForwardA = 01
  - If (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd ≠ 0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))  
ForwardB = 01

# Double Data Hazard

- Consider the sequence:

add  $x_1, x_1, x_2$

add  $x_1, x_1, x_3$

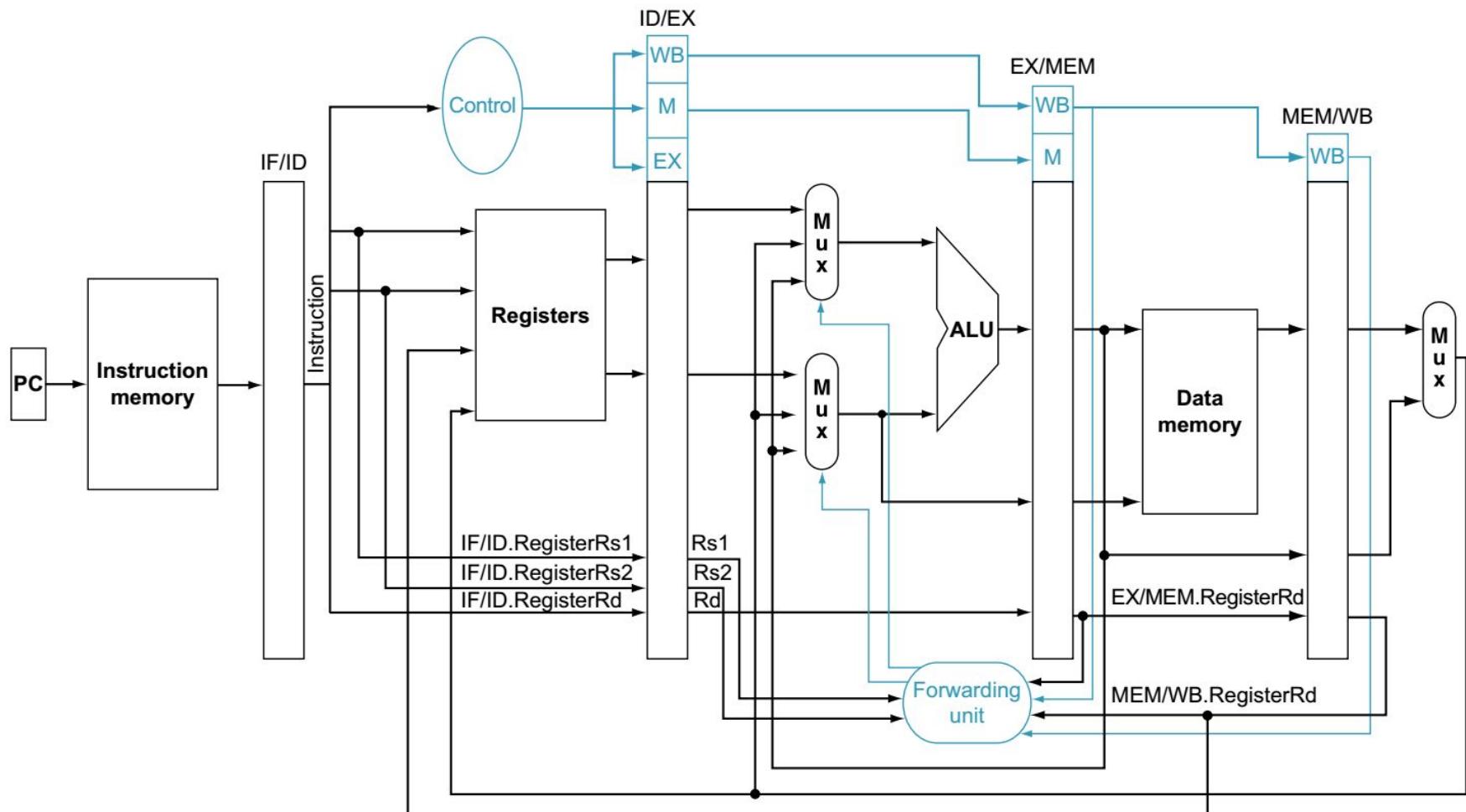
add  $x_1, x_1, x_4$

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

# 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.RegisterRs1))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) 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.RegisterRs2))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

# Datapath with Forwarding



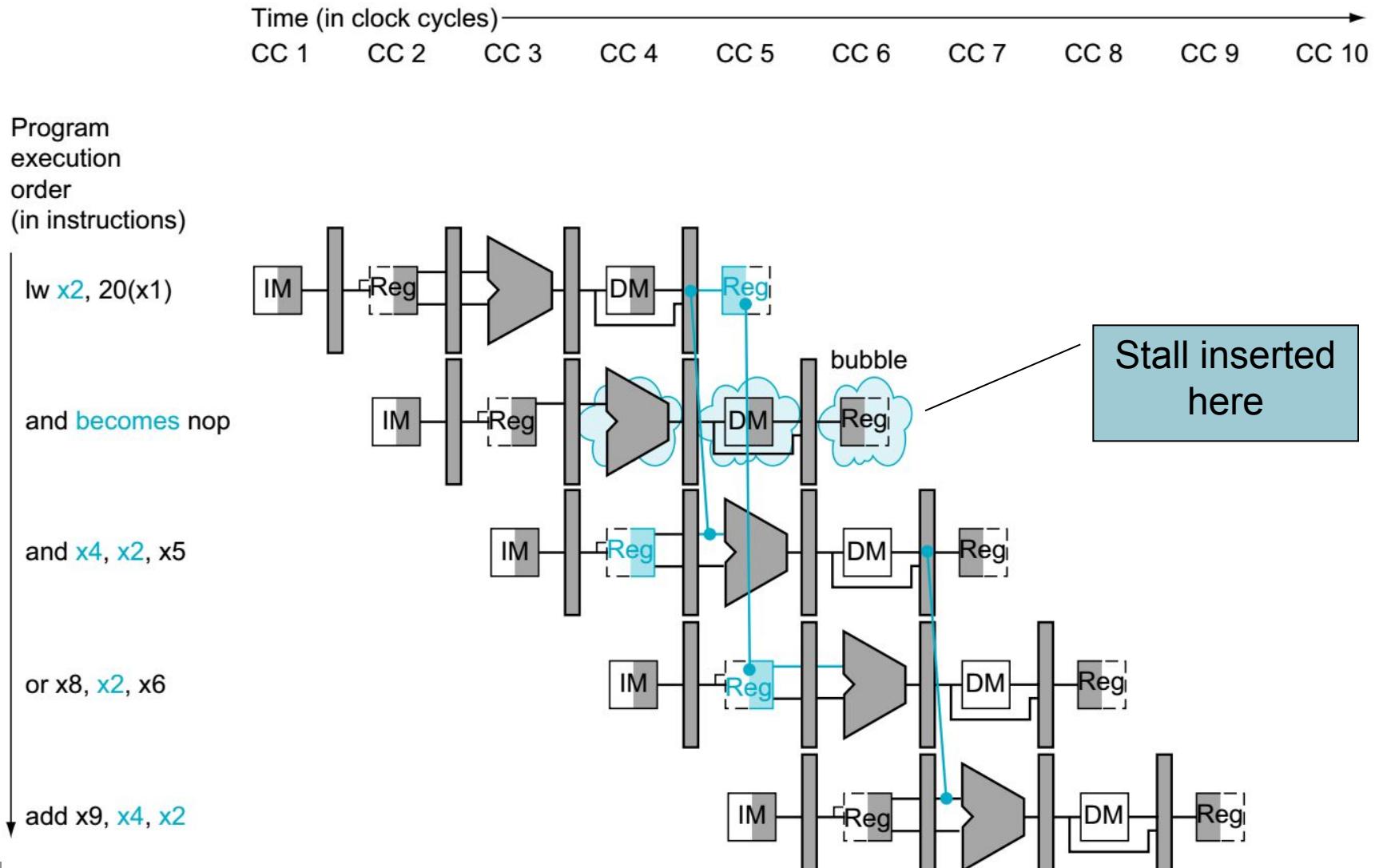
# Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and
$$((ID/EX.RegisterRd = IF/ID.RegisterRs1) \text{ or } (ID/EX.RegisterRd = IF/ID.RegisterRs1))$$
- If detected, stall and insert bubble

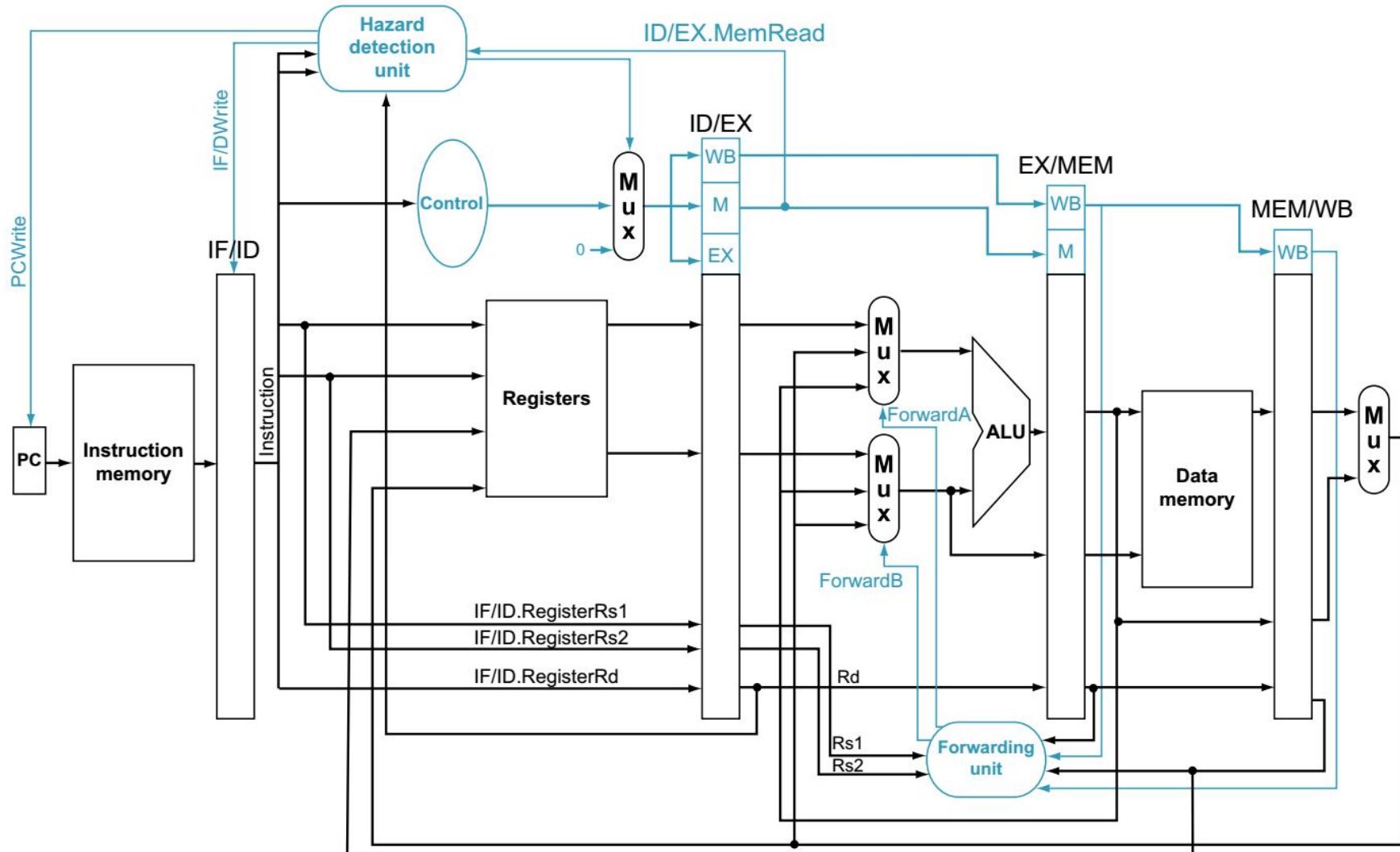
# How to Stall the Pipeline

- 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

# Load-Use Data Hazard



# Datapath with Hazard Detection



# Stalls and Performance

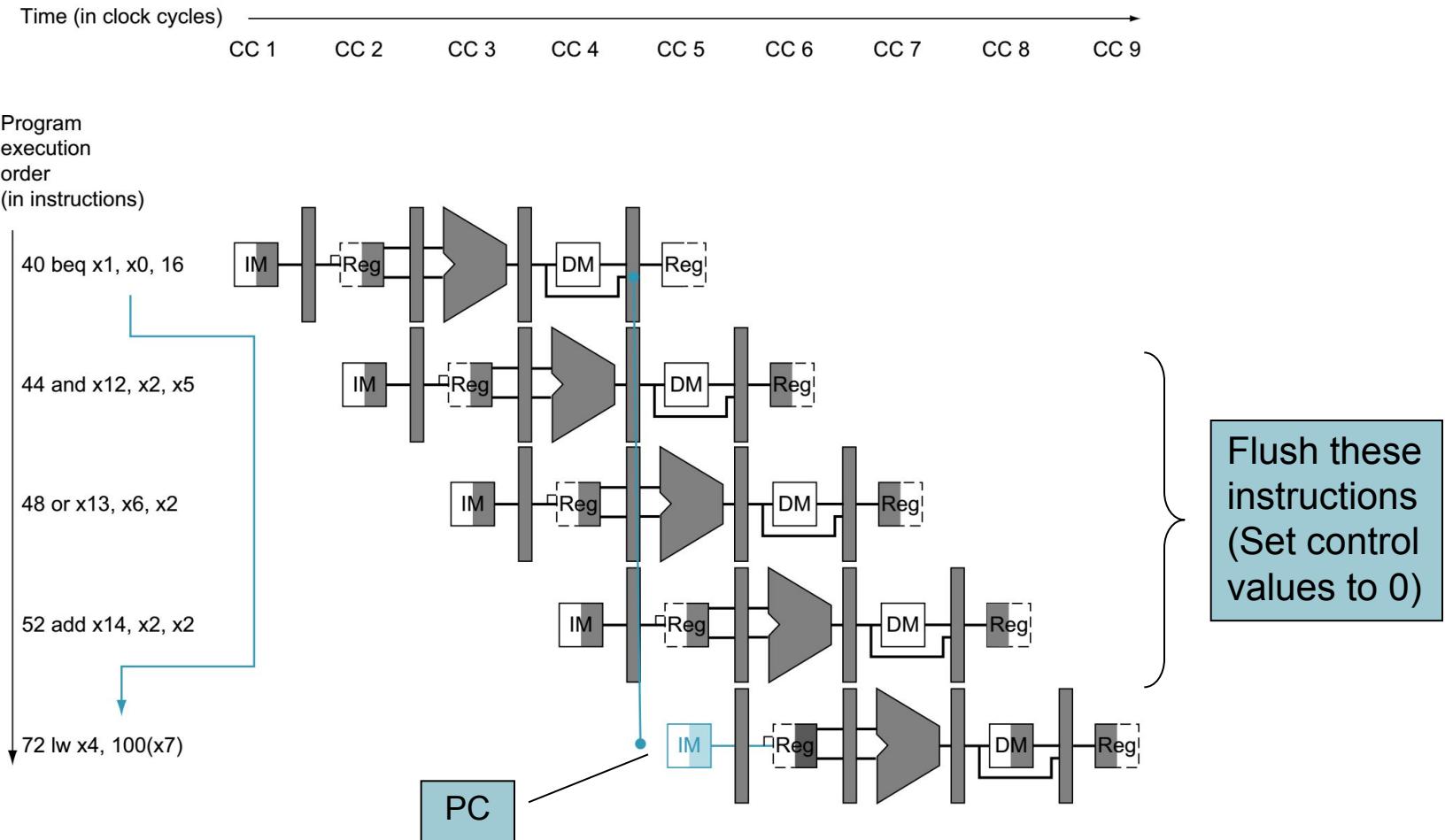
## The BIG Picture

- 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



# Branch Hazards

- If branch outcome determined in MEM

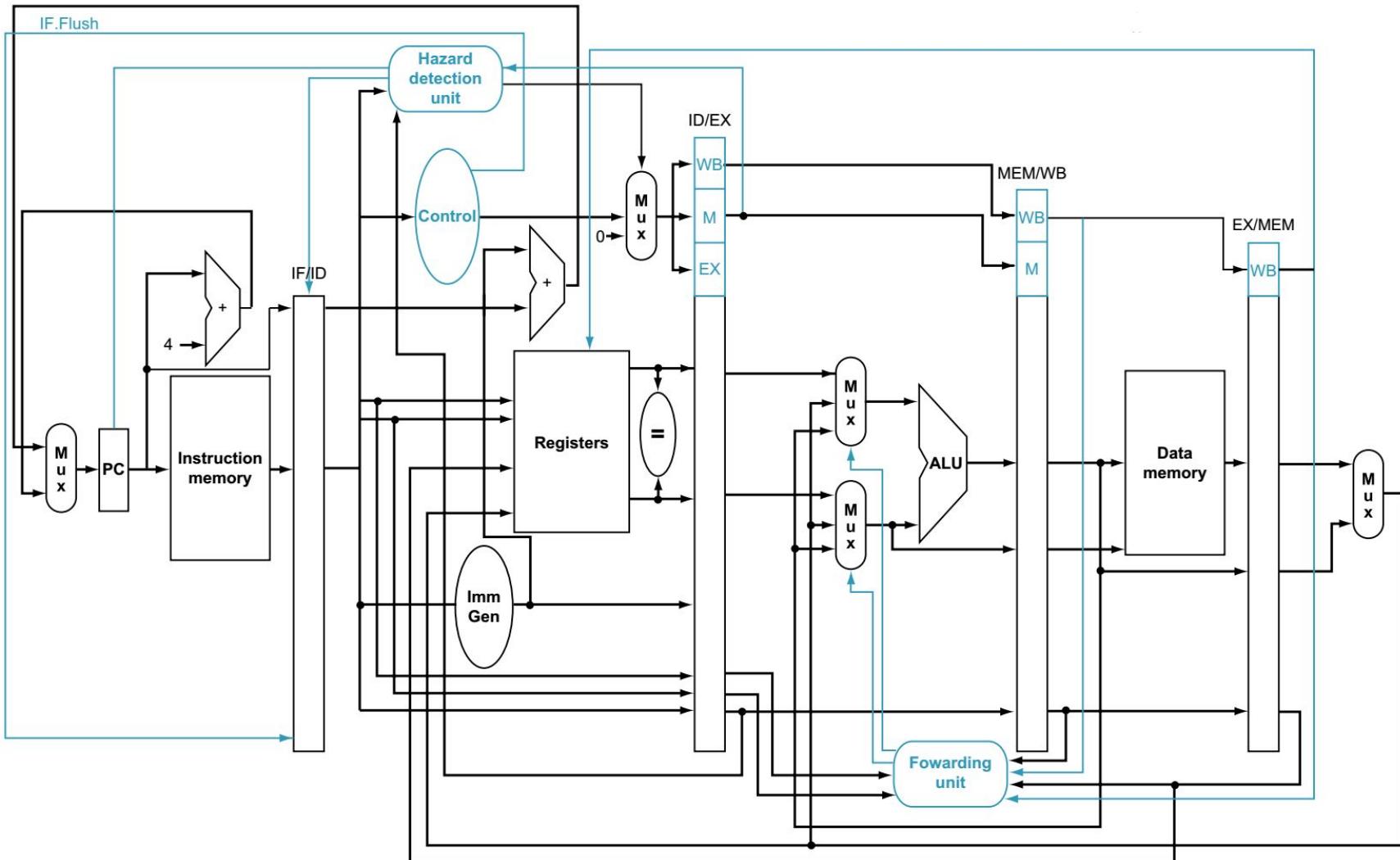


# Reducing Branch Delay<sub>1</sub>

---

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

# Reducing Branch Delay<sub>2</sub>

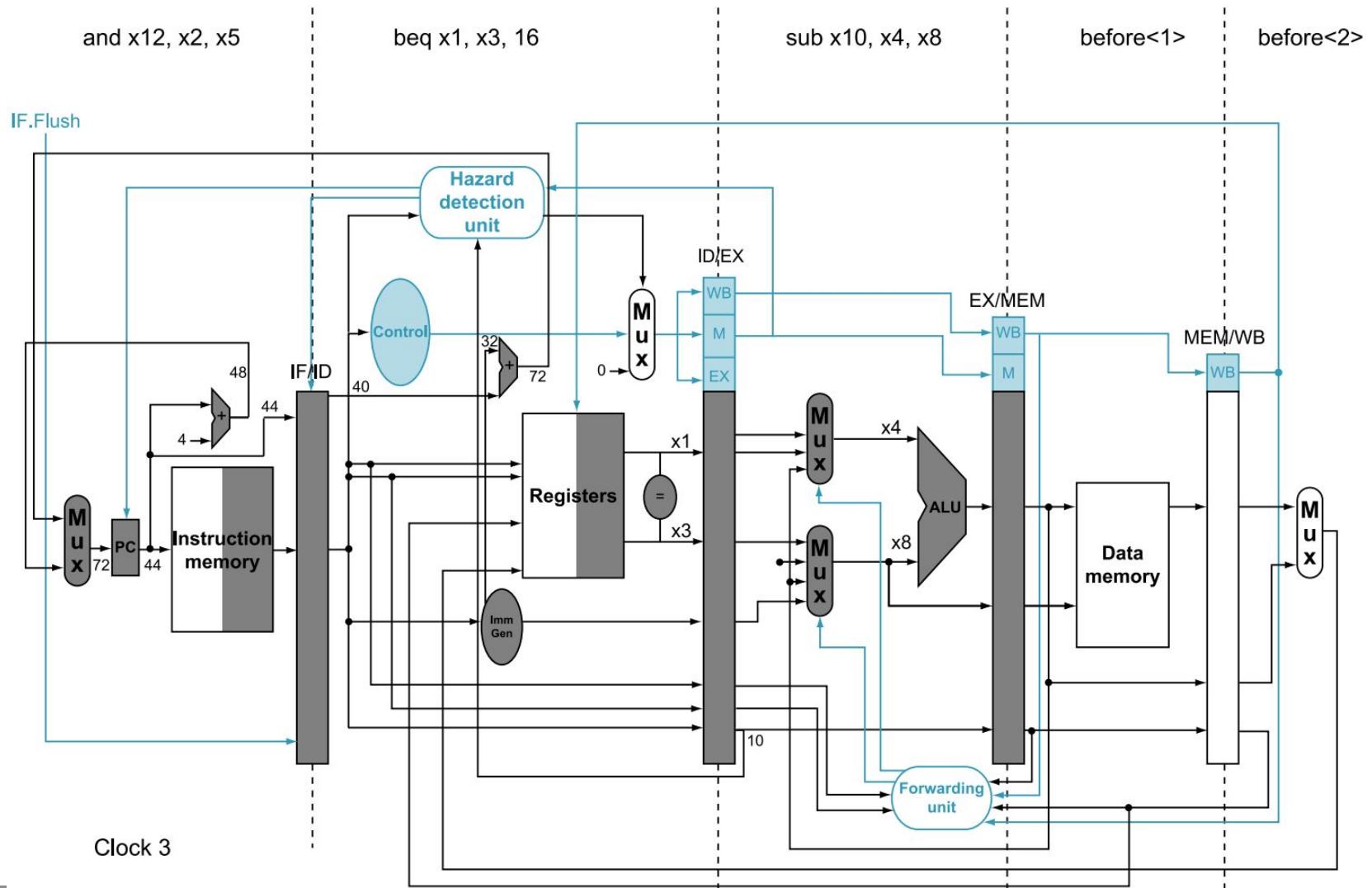


# Example: Branch Taken<sub>1</sub>

- Example: branch taken

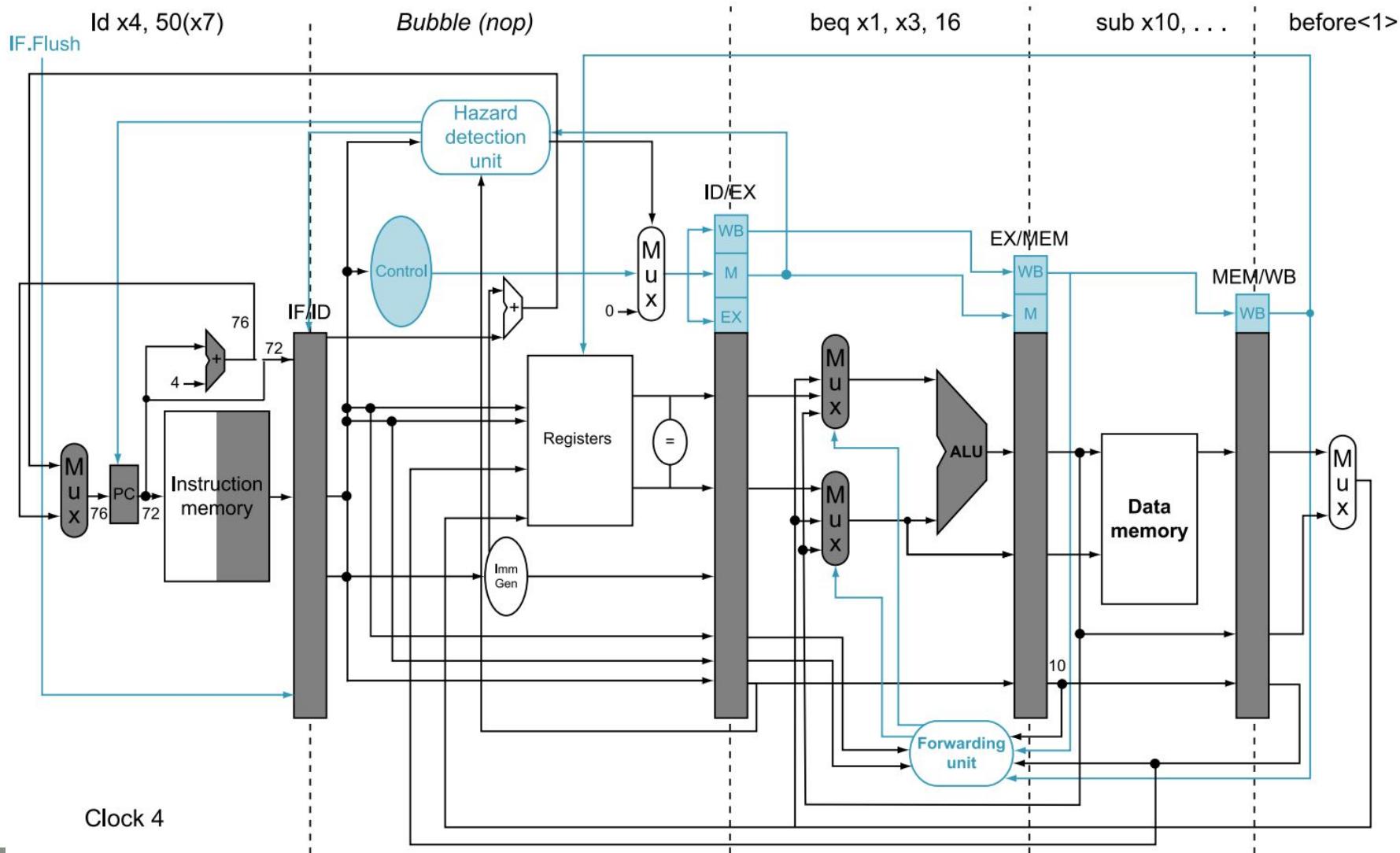
```
36: sub x10, x4, x8
40: beq x1, x3, 16 // PC-relative branch
               // to 40+16*2=72
44: and x12, x2, x5
48: orr x13, x2, x6
52: add x14, x4, x2
56: sub x15, x6, x7
...
72: ld x4, 50(x7)
```

# Example: Branch Taken<sub>2</sub>



Clock 3

# Example: Branch Taken<sub>3</sub>

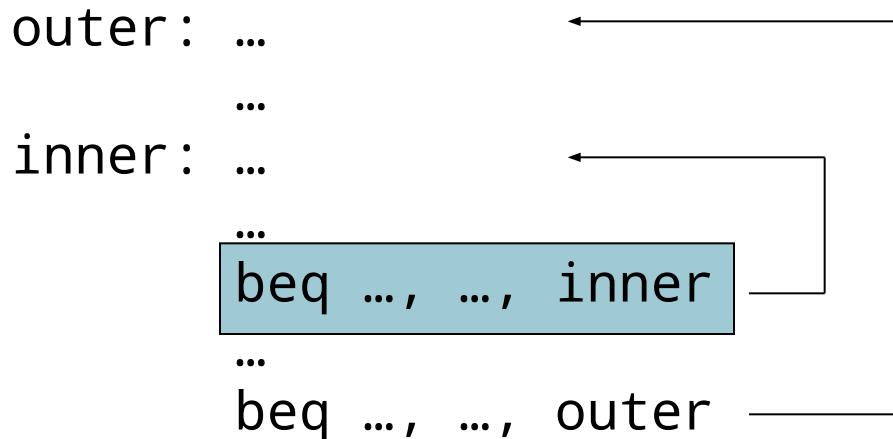


# Dynamic Branch Prediction

- 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

# 1-Bit Predictor: Shortcoming

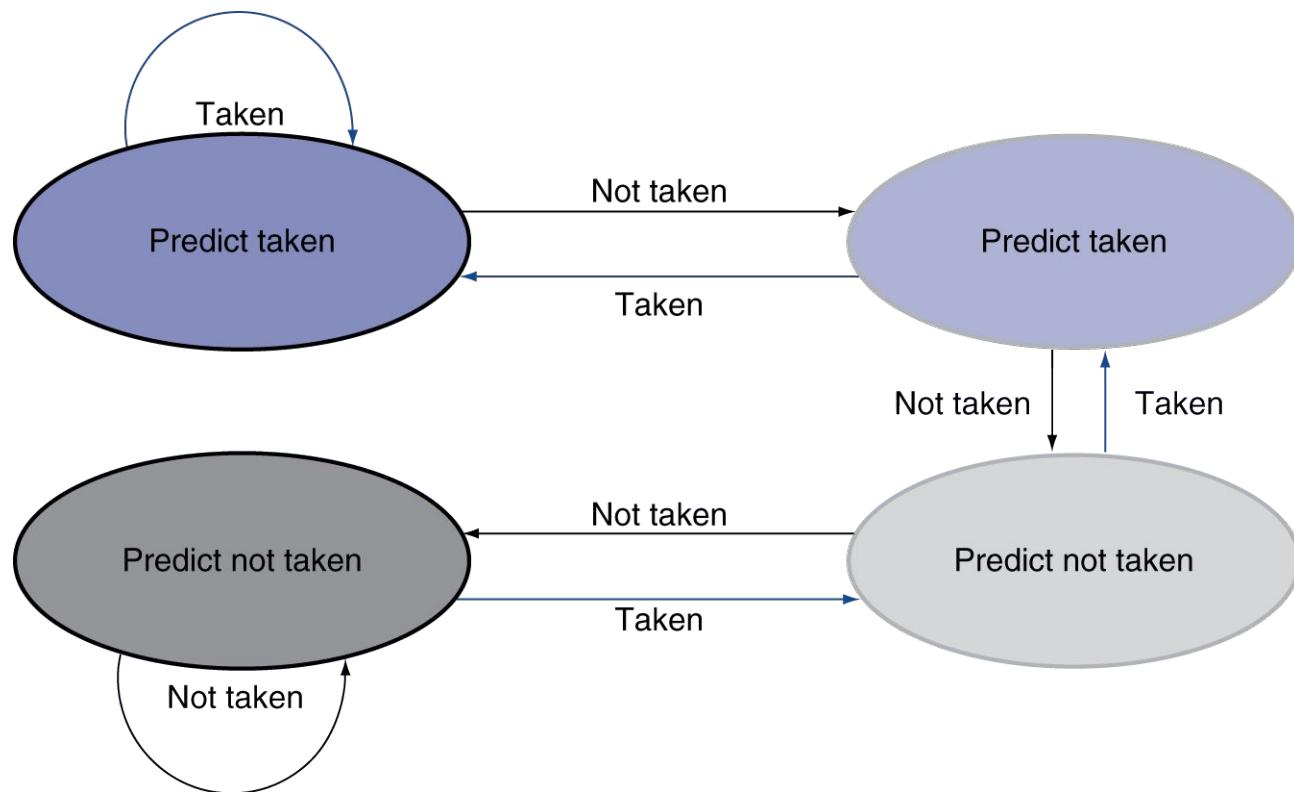
- Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

# 2-Bit Predictor

- Only change prediction on two successive mispredictions



# 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

# 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, syscall, ...
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

# Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, ...
- Jump to handler
  - Assume at 0000 0000 1C09 0000<sub>hex</sub>

# An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Exception vector address to be added to a vector table base register:
  - Undefined opcode    00 0100 0000<sub>two</sub>
  - Hardware malfunction: 01 1000 0000<sub>two</sub>
  - ...                        ...
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

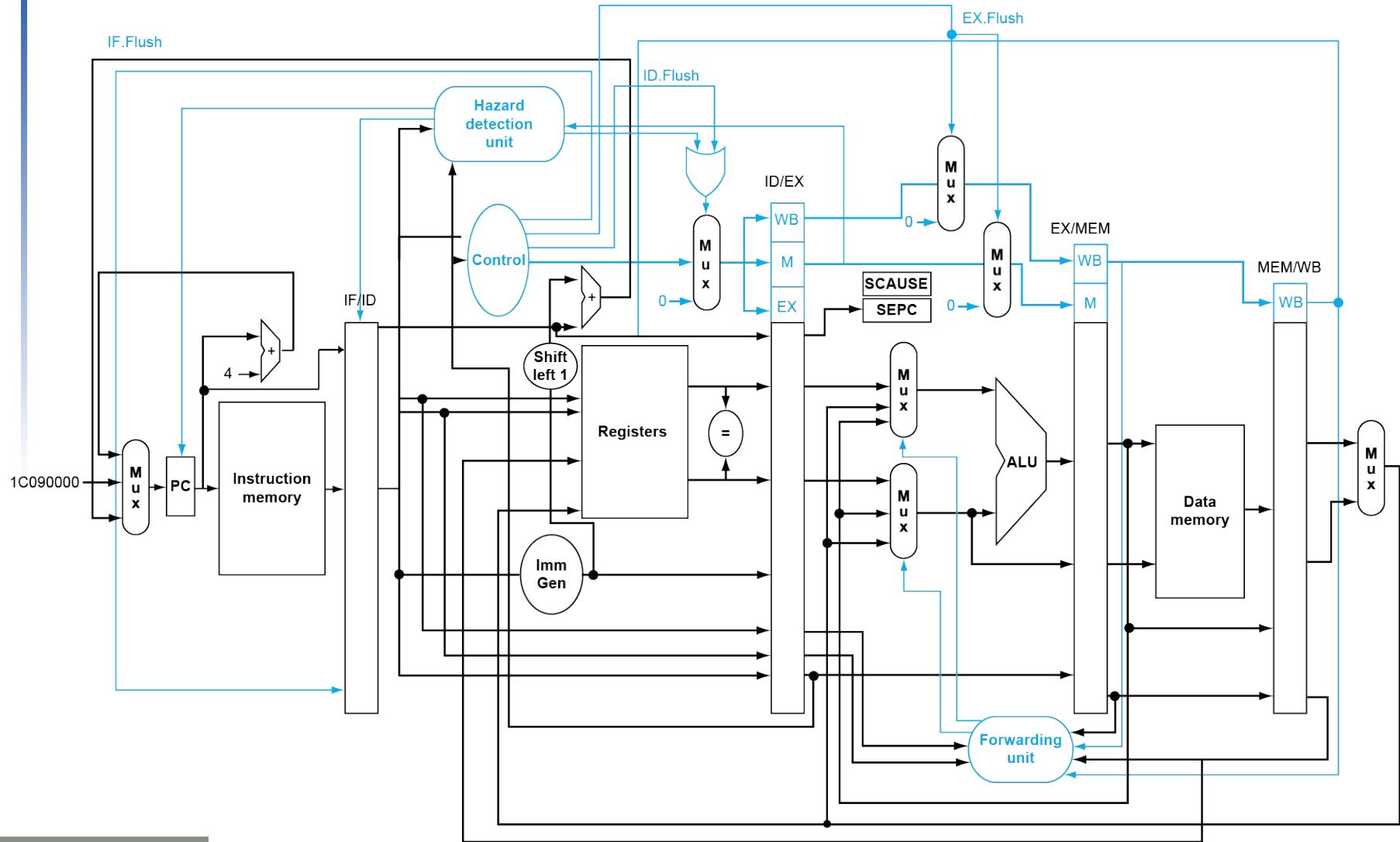
# Handler Actions

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

# Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage
  - add x1 , x2 , x1
    - Prevent x1 from being clobbered
    - Complete previous instructions
    - Flush add and subsequent instructions
    - Set SEPC and SCAUSE 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 next instruction
    - Refetched and executed from scratch
- PC saved in SEPC register
  - Identifies causing instruction

# Exception Example

- Exception on add in

```
40  sub   x11, x2, x4  
44  and   x12, x2, x5  
48  or    x13, x2, x6  
4c  add   x1,  x2, x1  
50  sub   x15, x6, x7  
54  lw    x16, 100(x7)
```

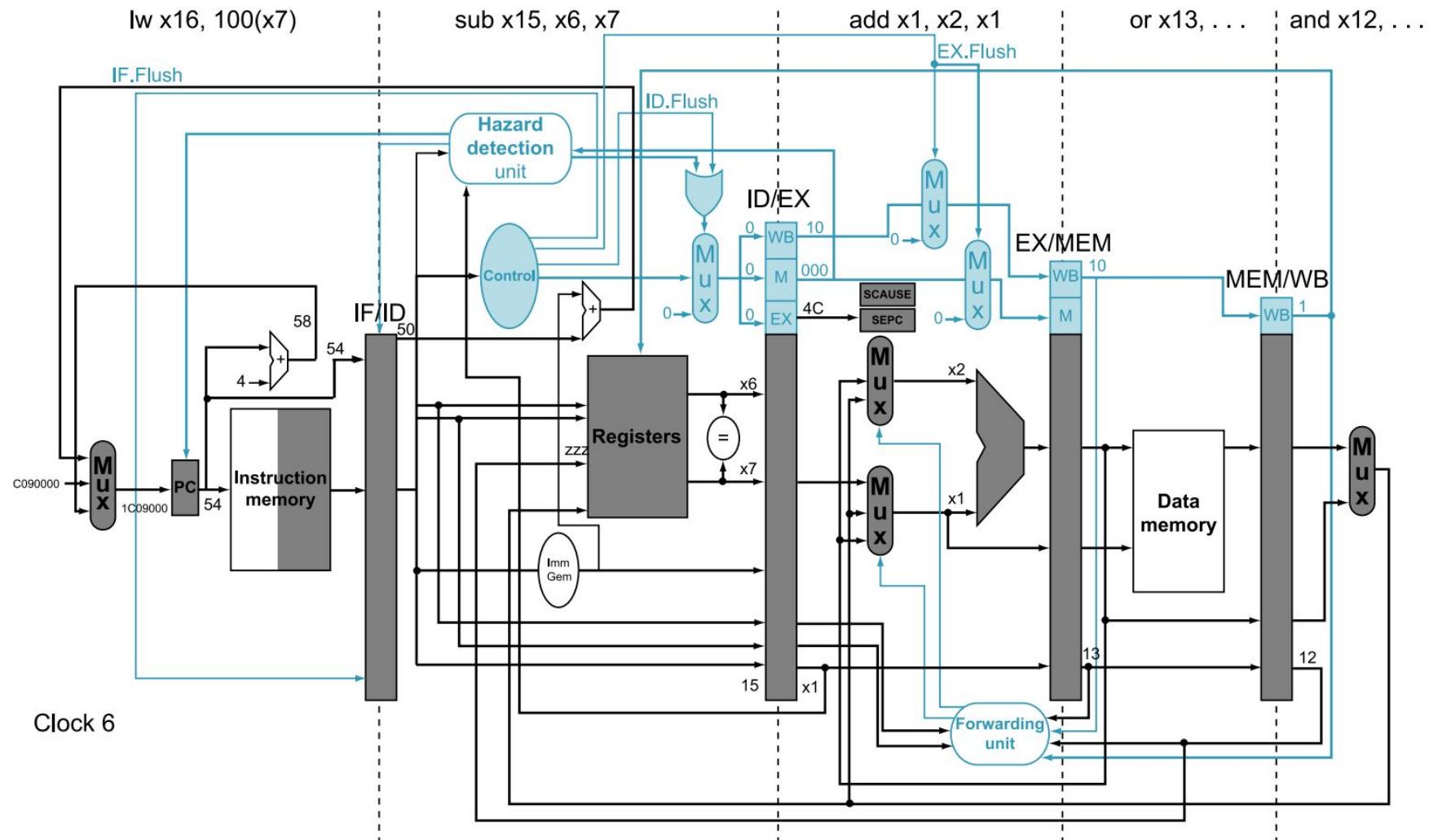
...

- Handler

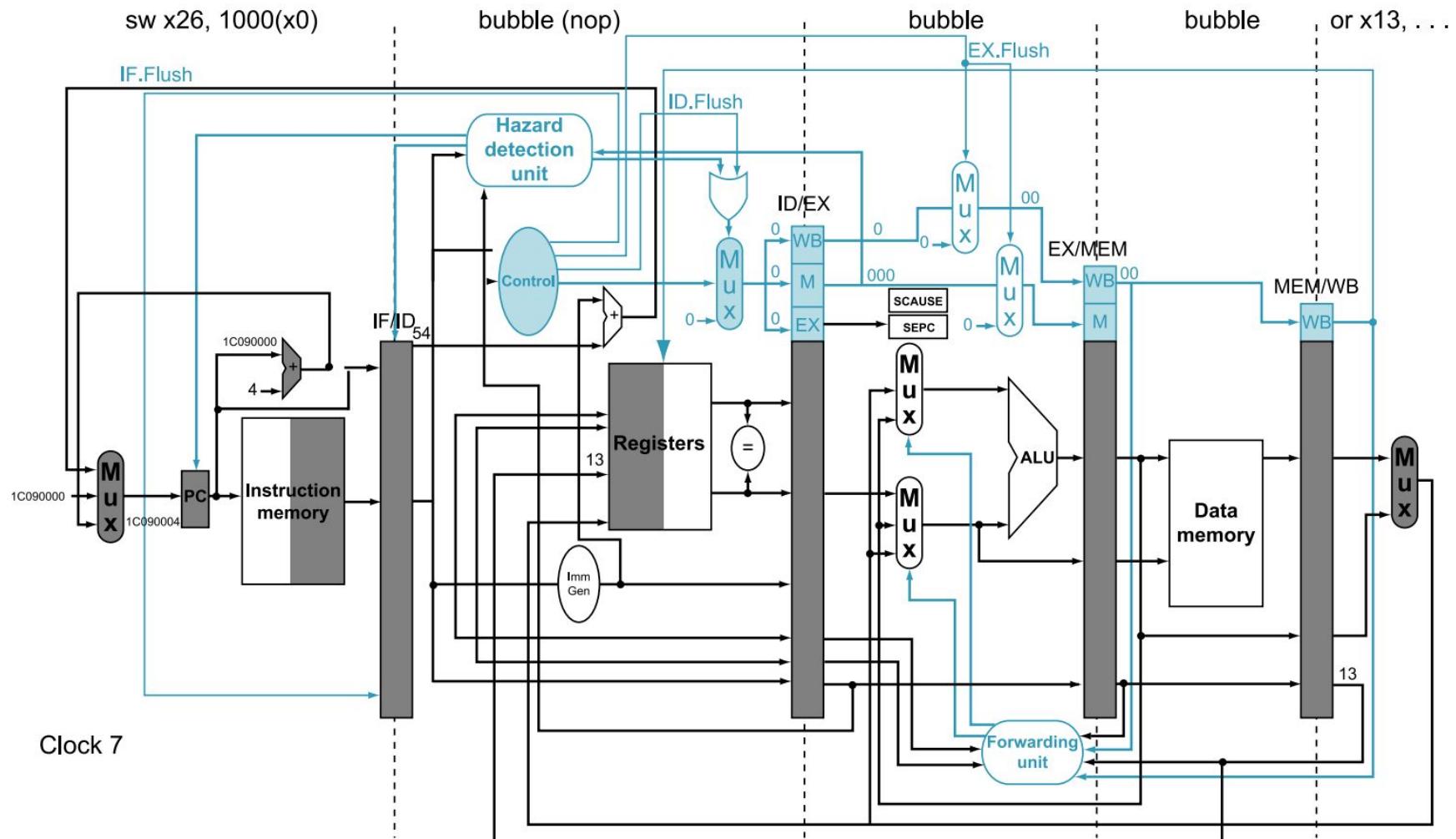
```
1C090000      sw   x26, 1000(x10)  
1c090004      sw   x27, 1008(x10)
```

...

# Exception Example



# Exception Example



# 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

# Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

# Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

# 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