

# Pipelined Processor Design

'22H2

송 인 식

# Outline

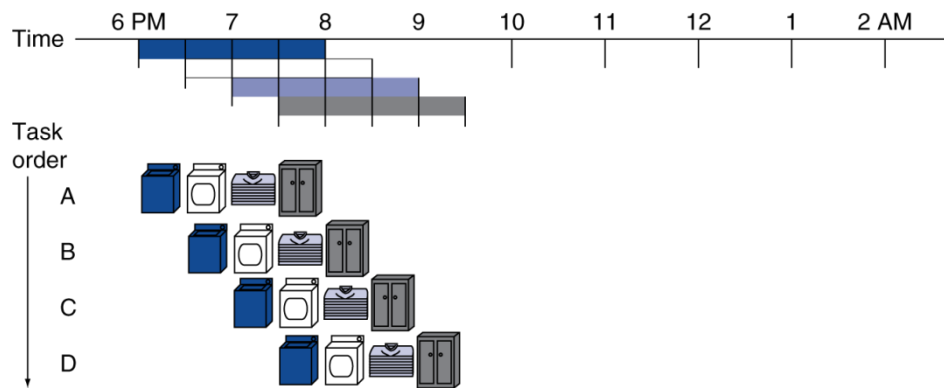
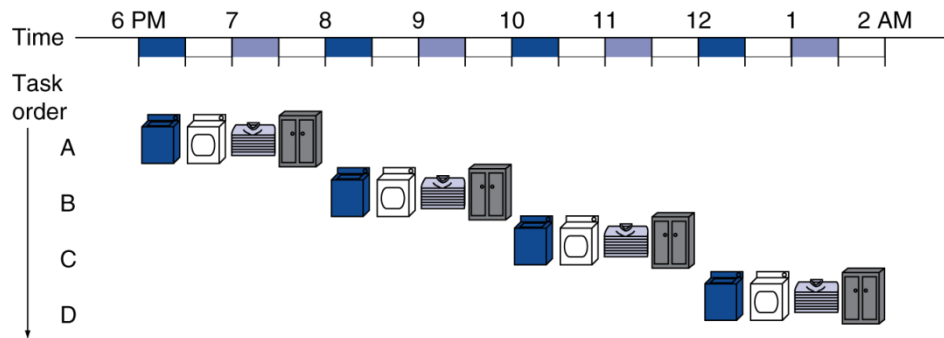
- An Overview of Pipelining
- Pipelined Datapath and Control
- Data Hazards: Forwarding vs. Stalling

# Introduction

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

# Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



## ■ Four loads:

- Speedup  
 $= 8 / 3.5 = 2.3$

## ■ Non-stop:

- Speedup  
 $= 2n / 0.5n + 1.5 \approx 4$   
 $= \text{number of stages}$

# 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

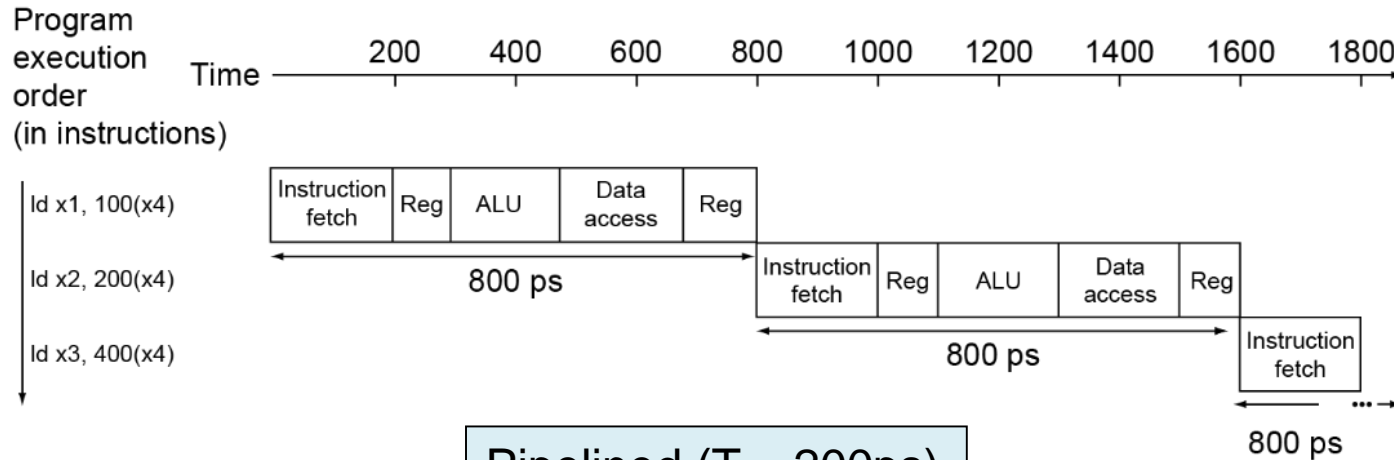
# Pipeline Performance

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

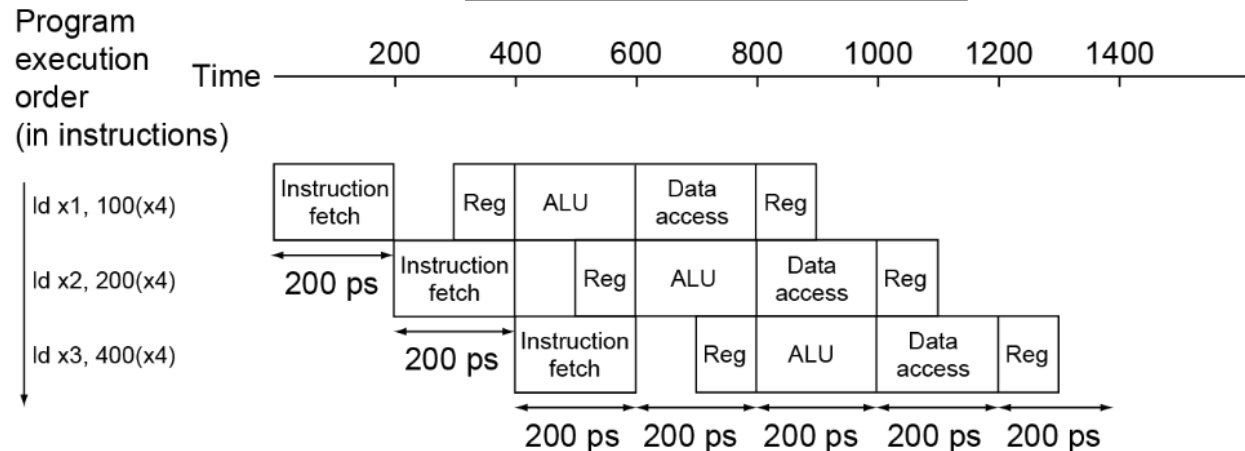
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

# Pipeline Performance

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>  
=  $\frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- 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

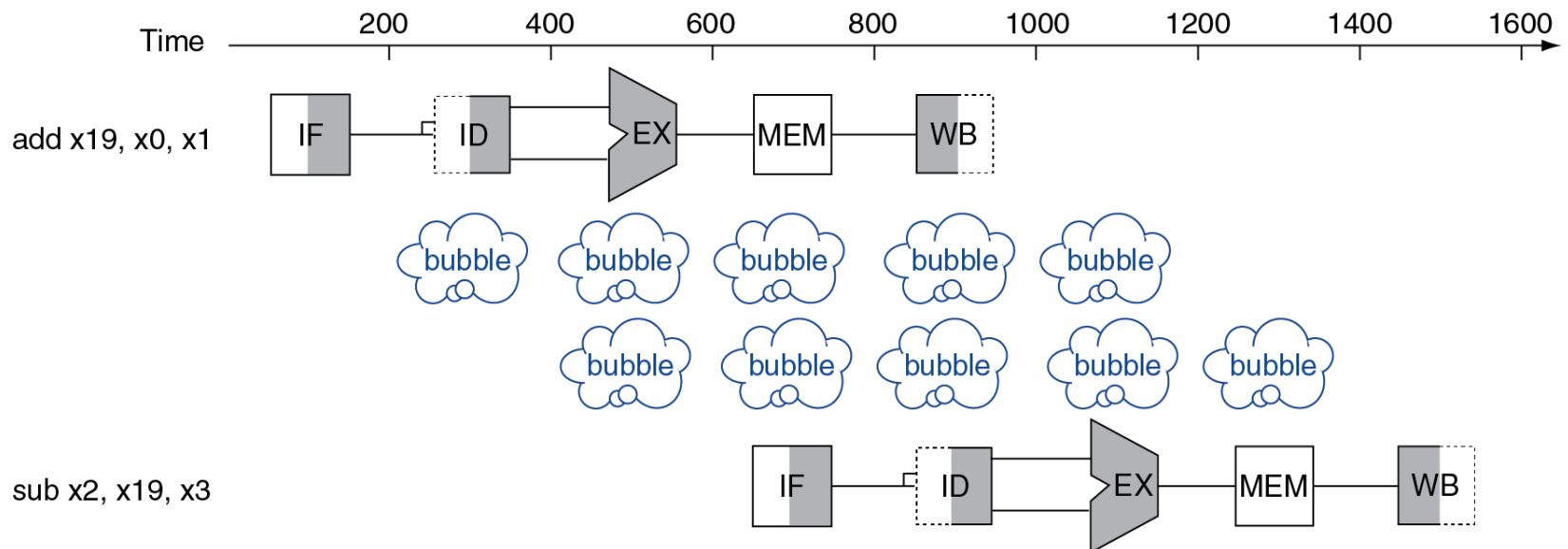
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

# 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

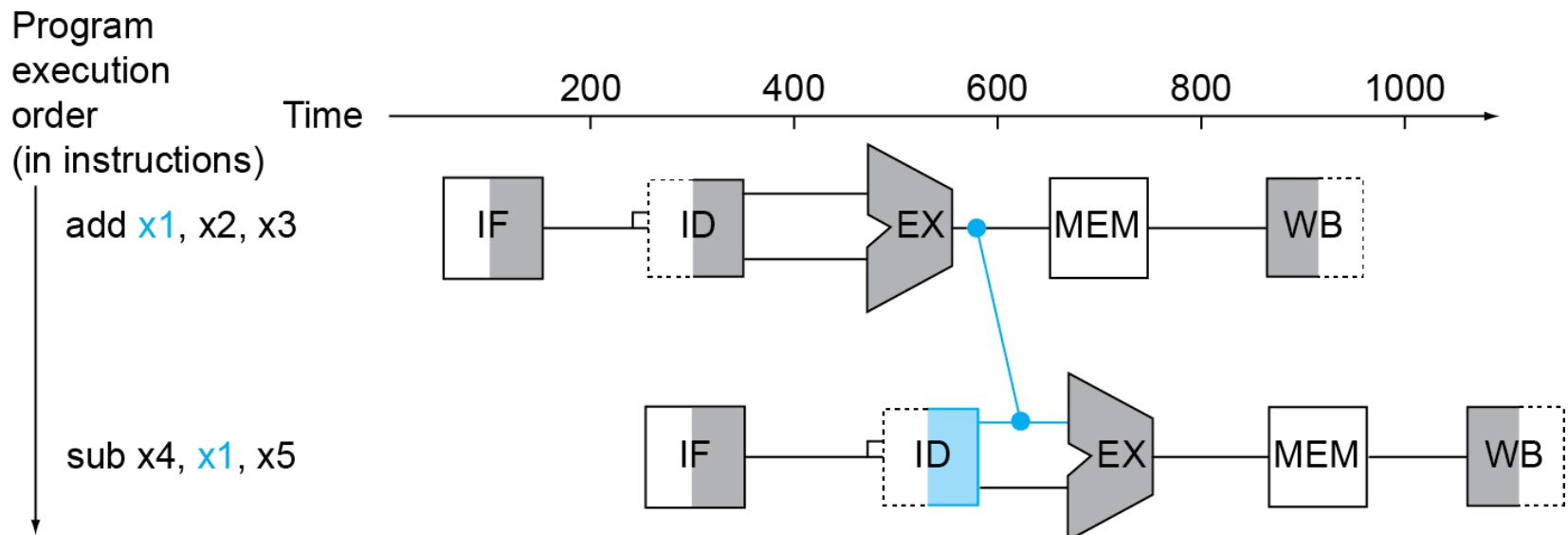
# Data Hazards

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



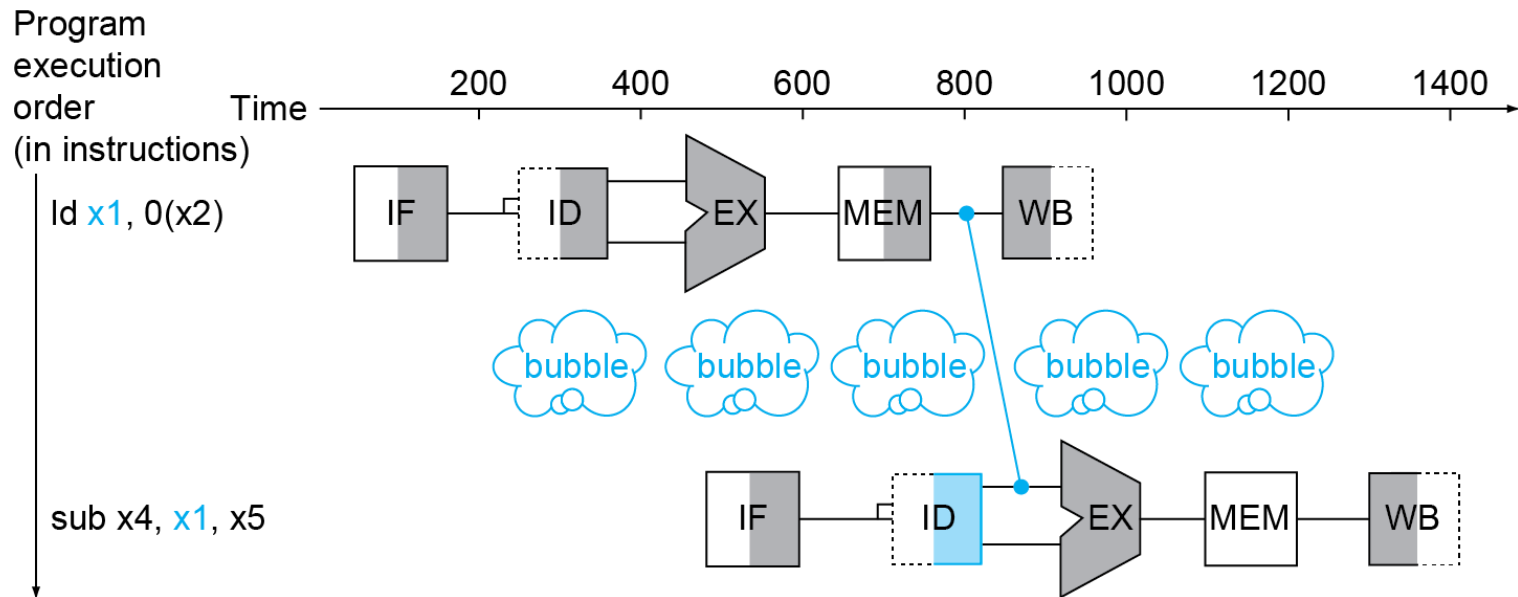
# Forwarding (aka Bypassing)

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



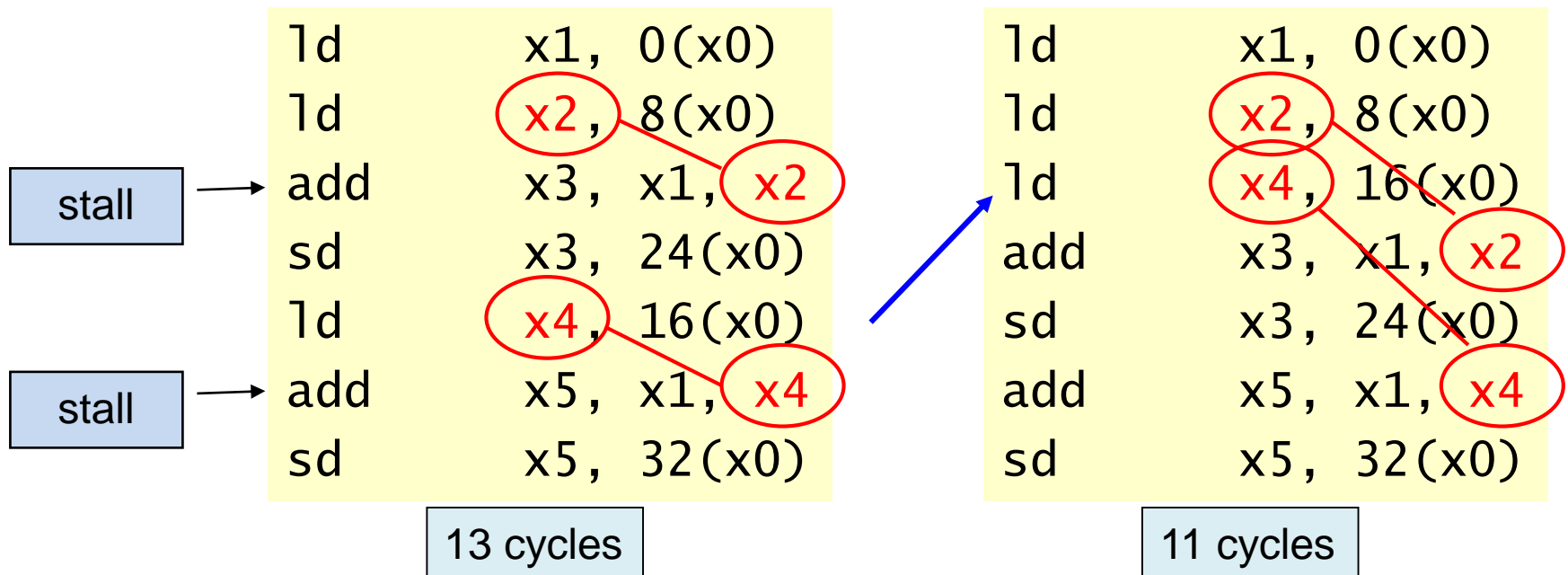
# Load-Use Data Hazard

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



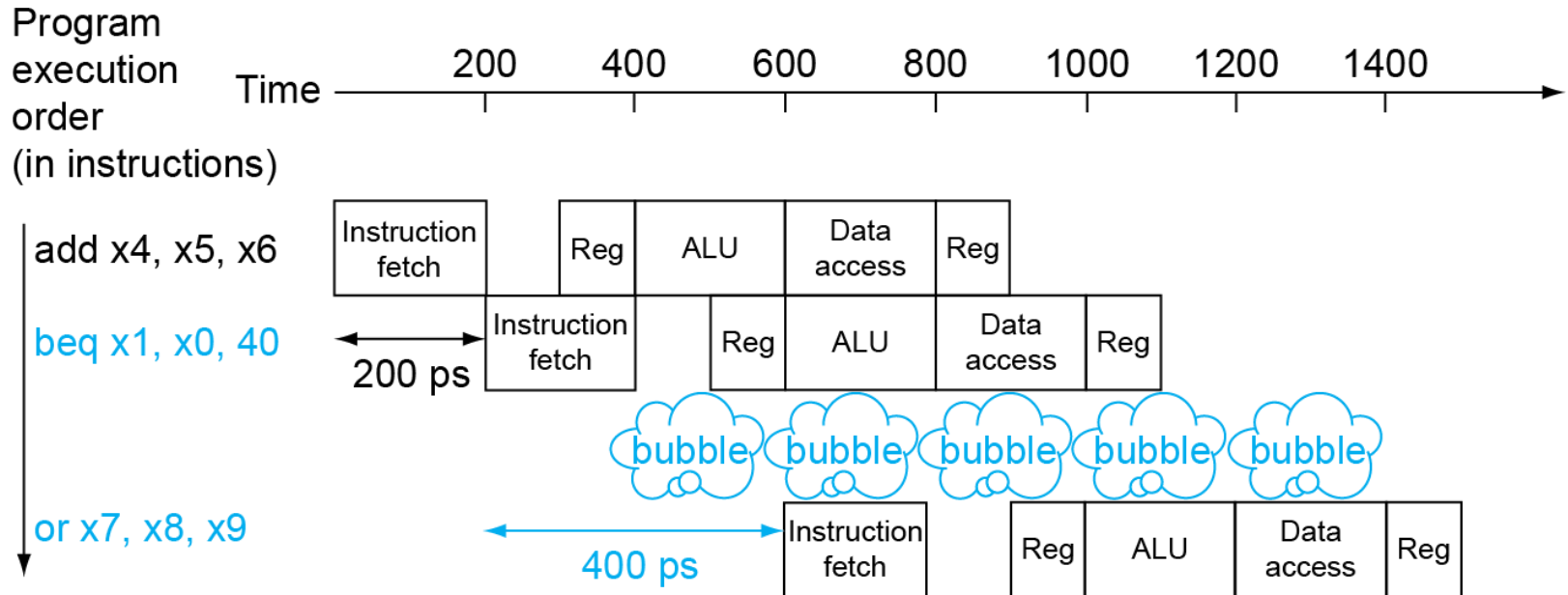
# Control Hazards

- 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

# 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

# Outline

- An Overview of Pipelining
- Pipelined Datapath and Control
- Data Hazards: Forwarding vs. Stalling

# RISC-V Pipelined Datapath

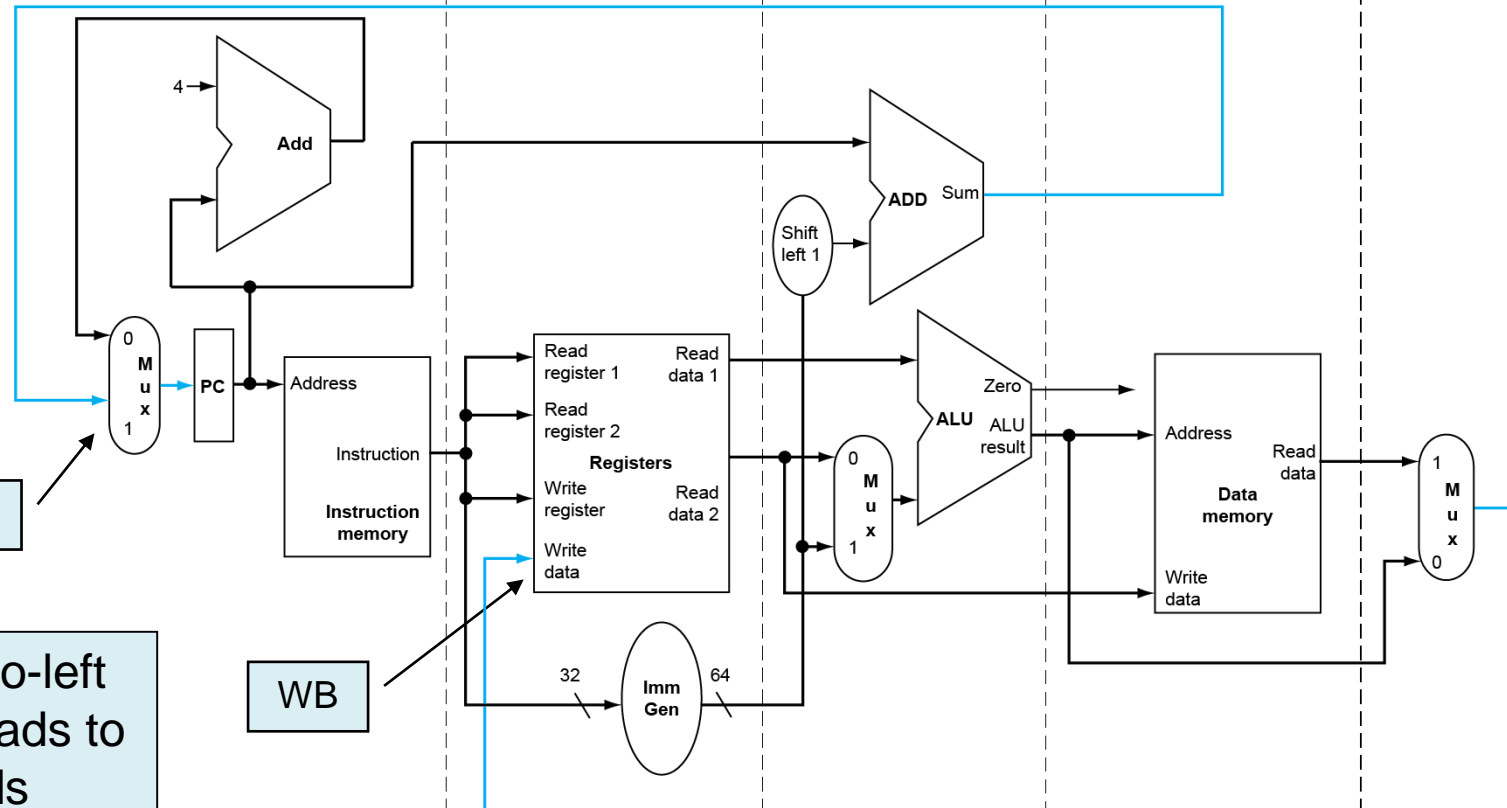
IF: Instruction fetch

ID: Instruction decode/  
register file read

EX: Execute/  
address calculation

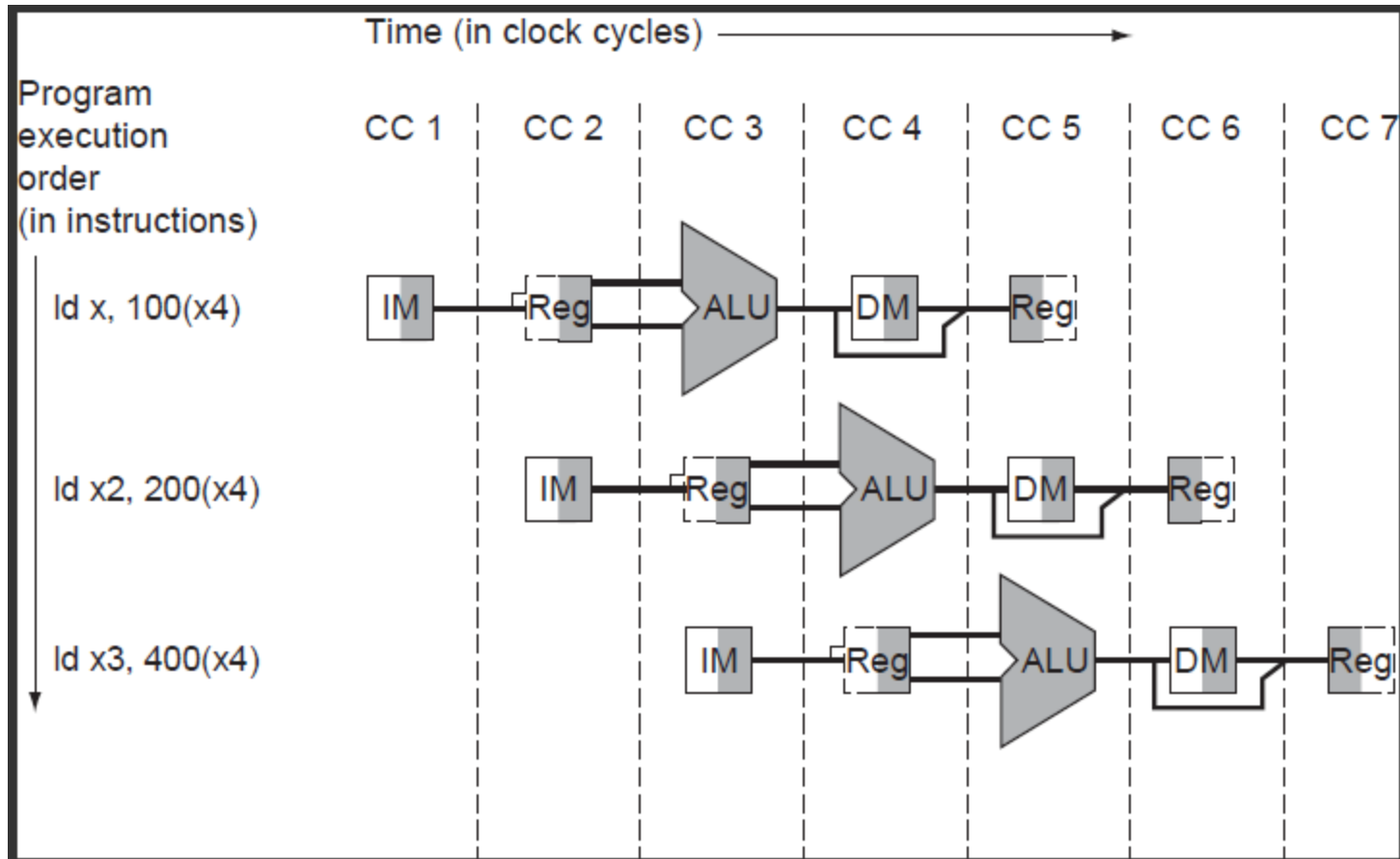
MEM: Memory access

WB: Write back



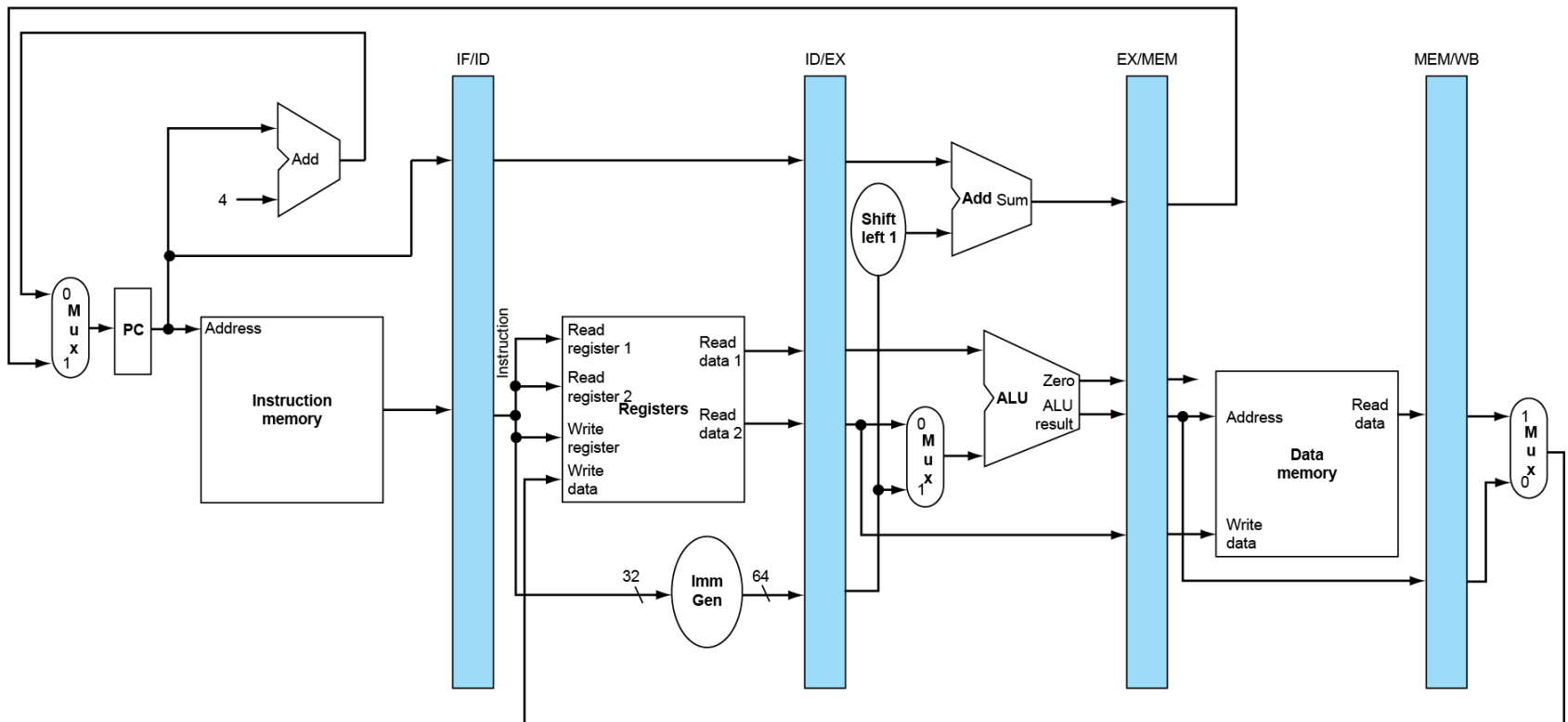
Right-to-left  
flow leads to  
hazards

# Instructions being executed in pipelined execution



# Pipeline registers

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

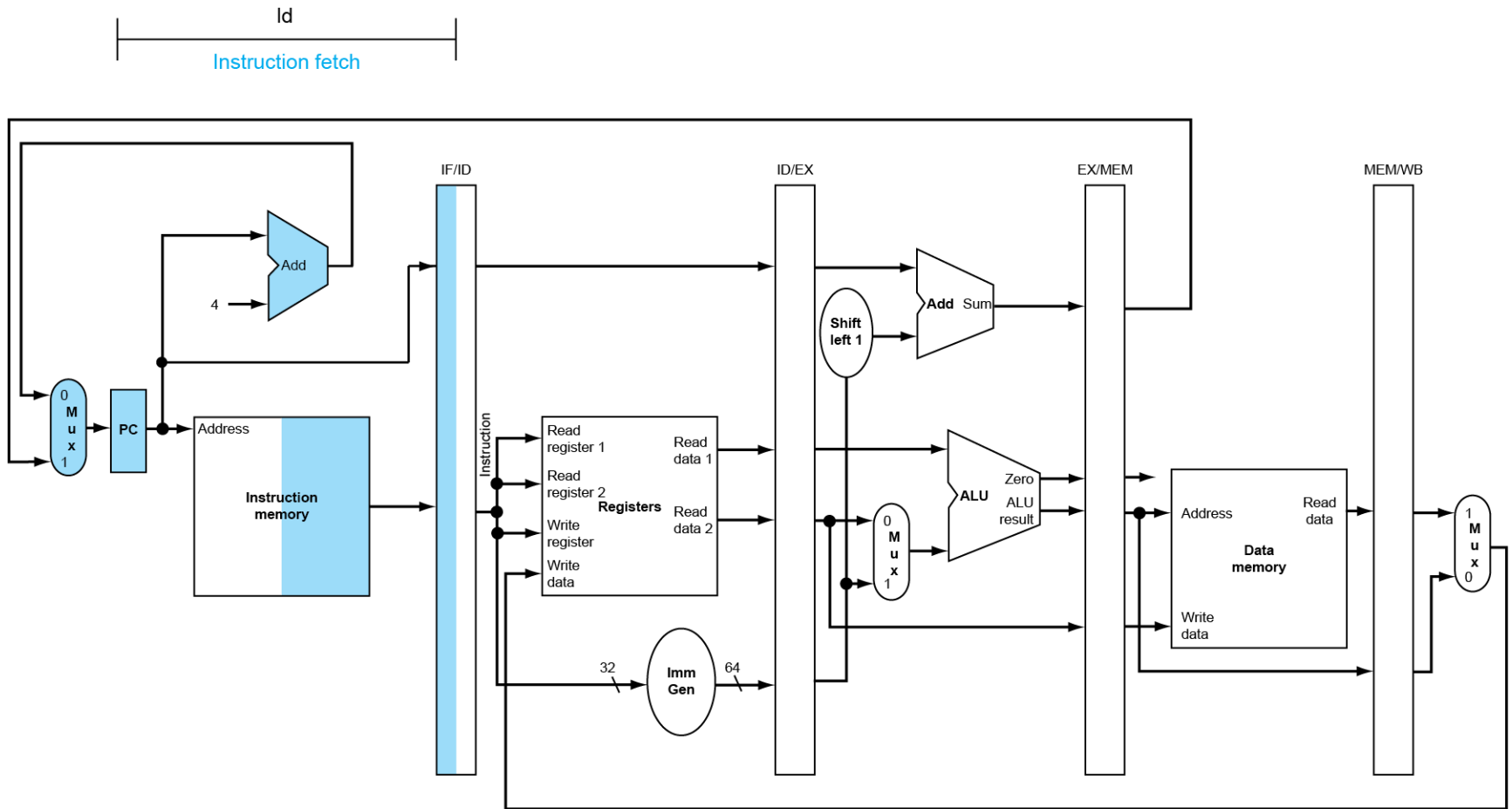




# Pipeline Operation

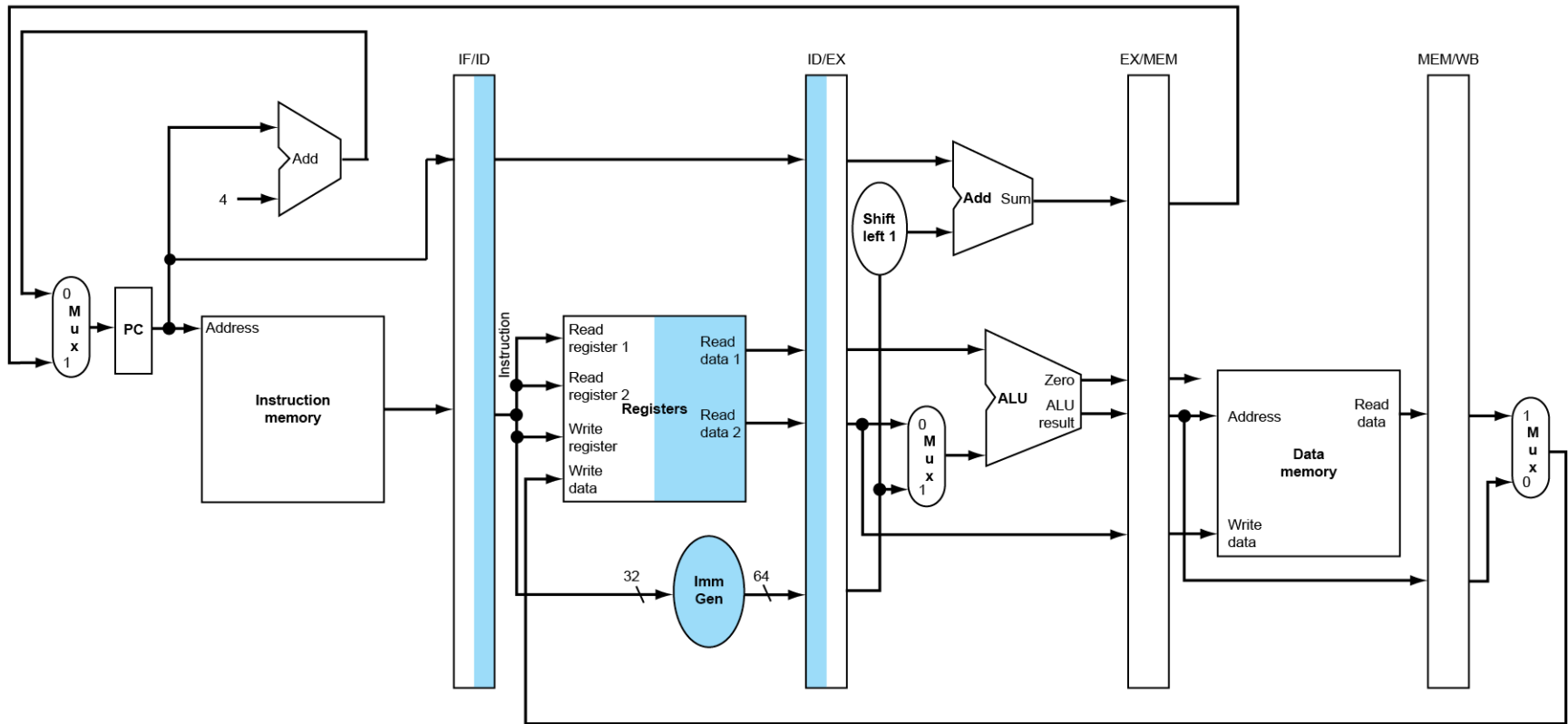
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams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

# IF for Load, Store, ...

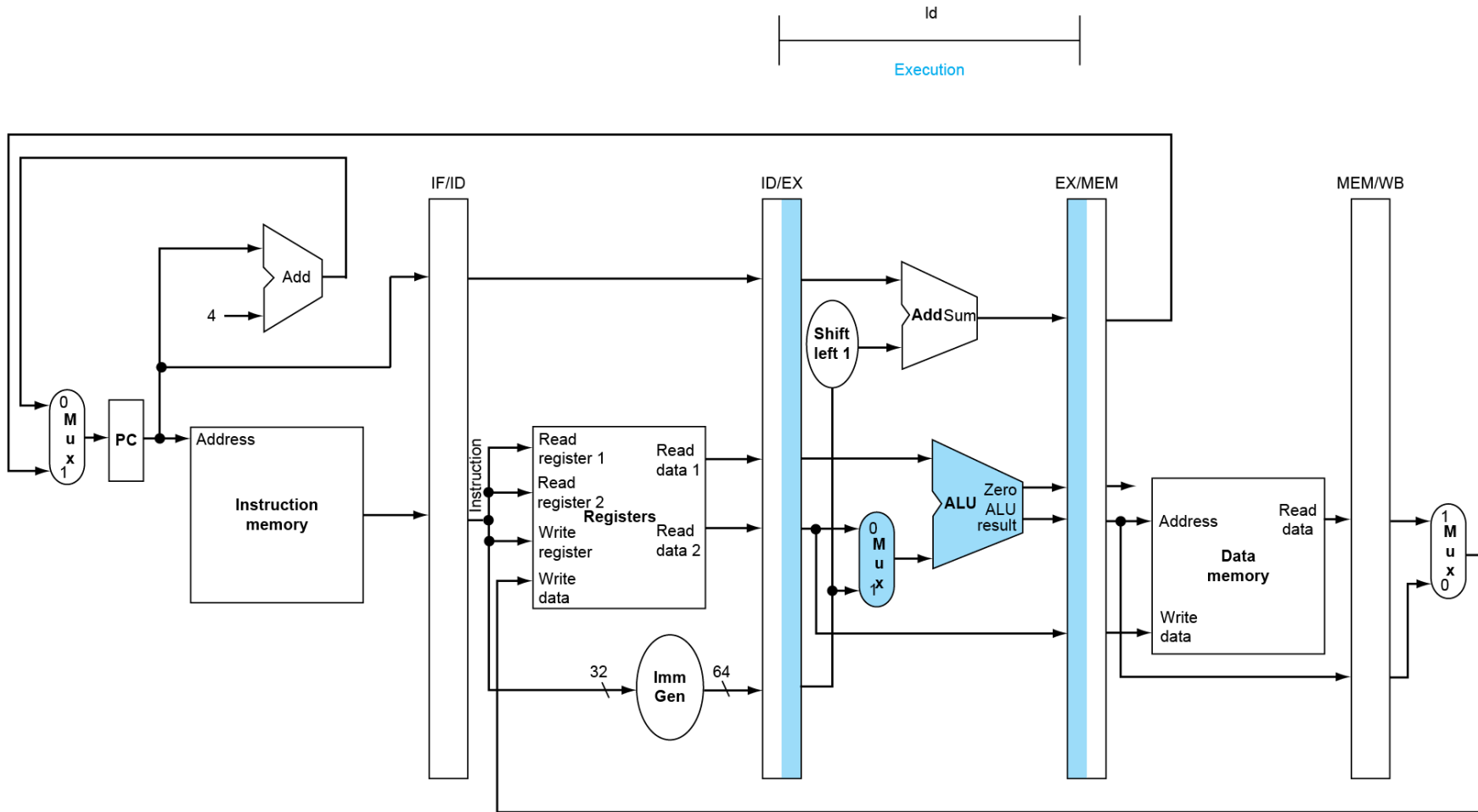


# ID for Load, Store, ...

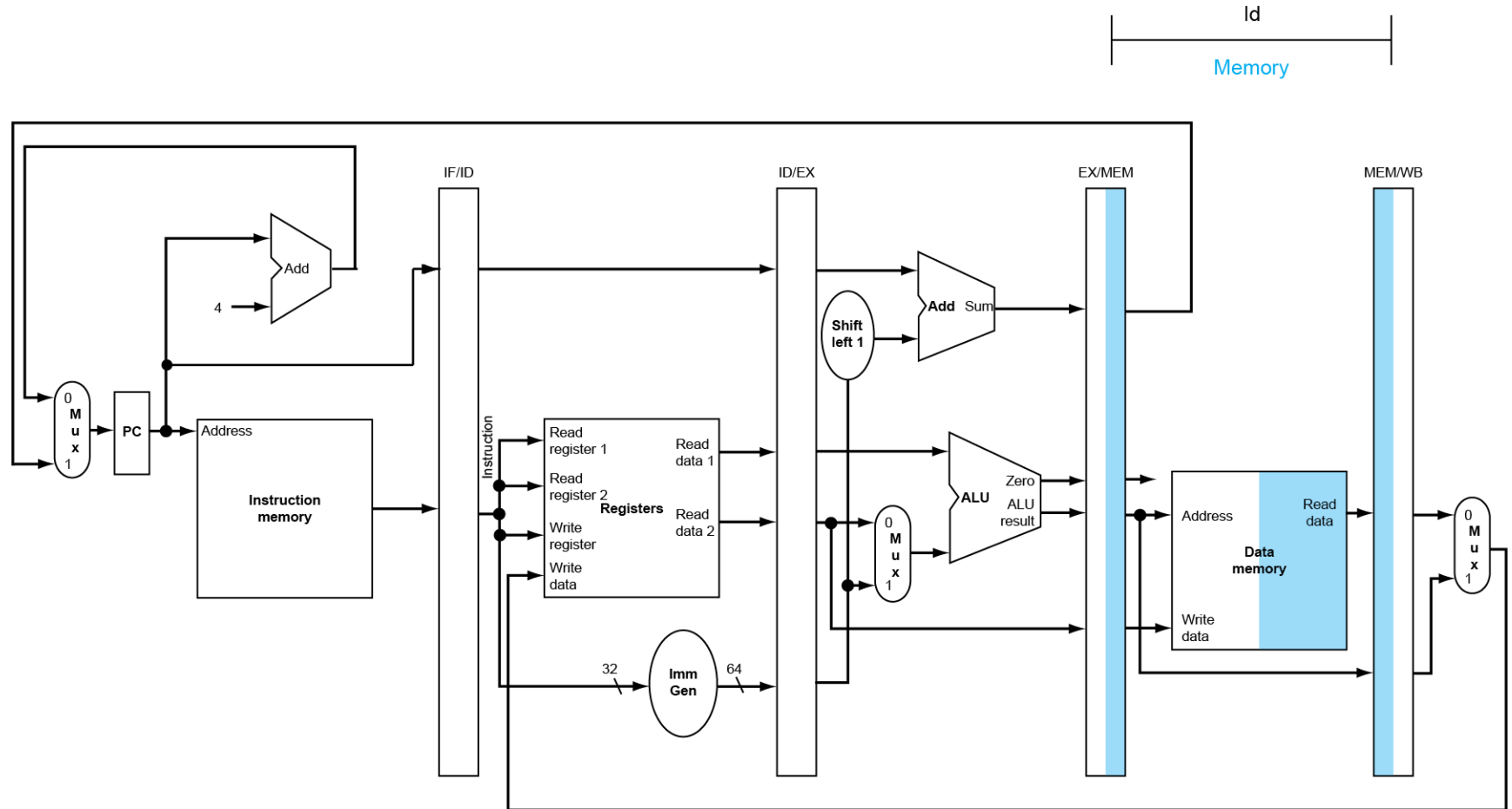
Id  
Instruction decode



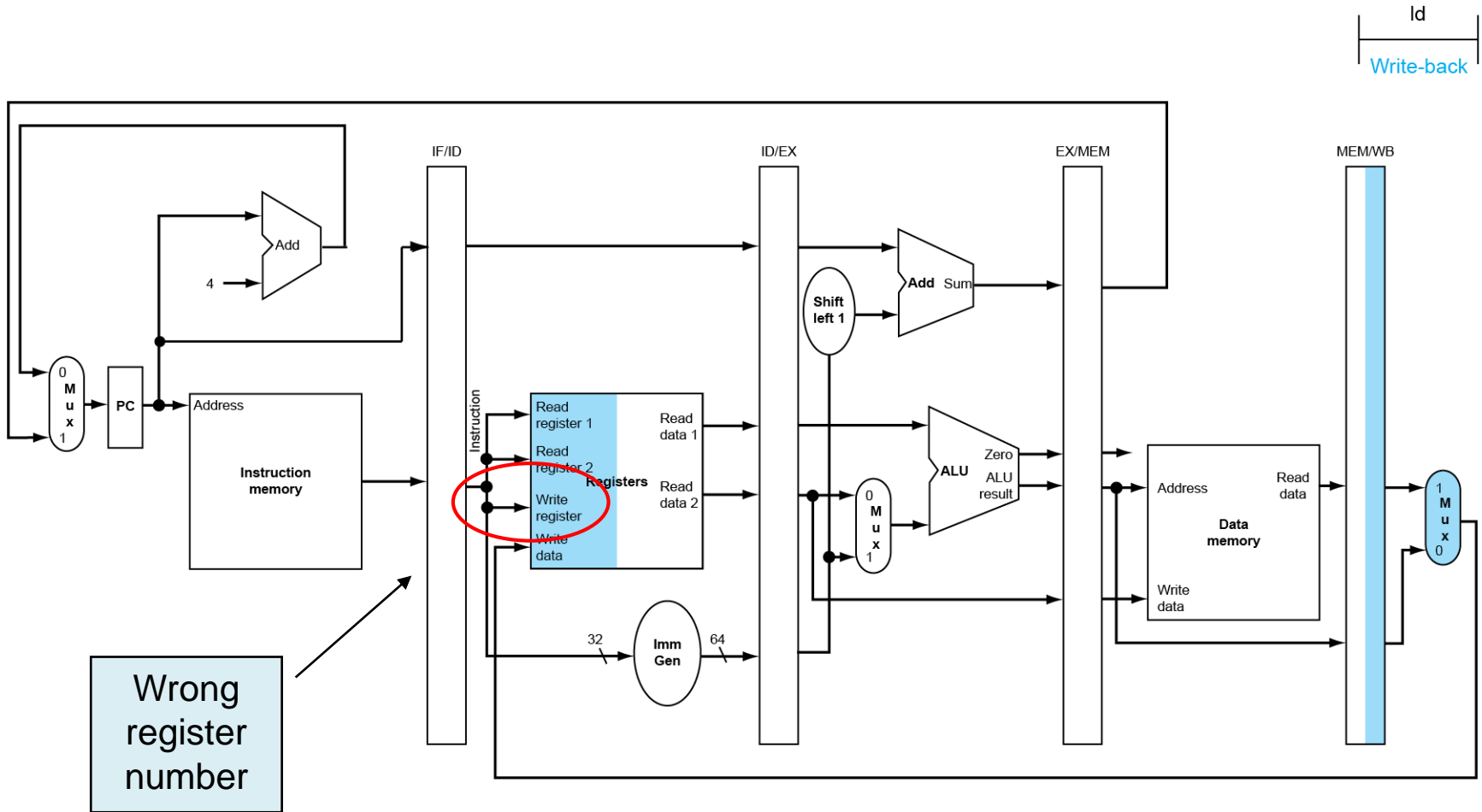
# EX for Load



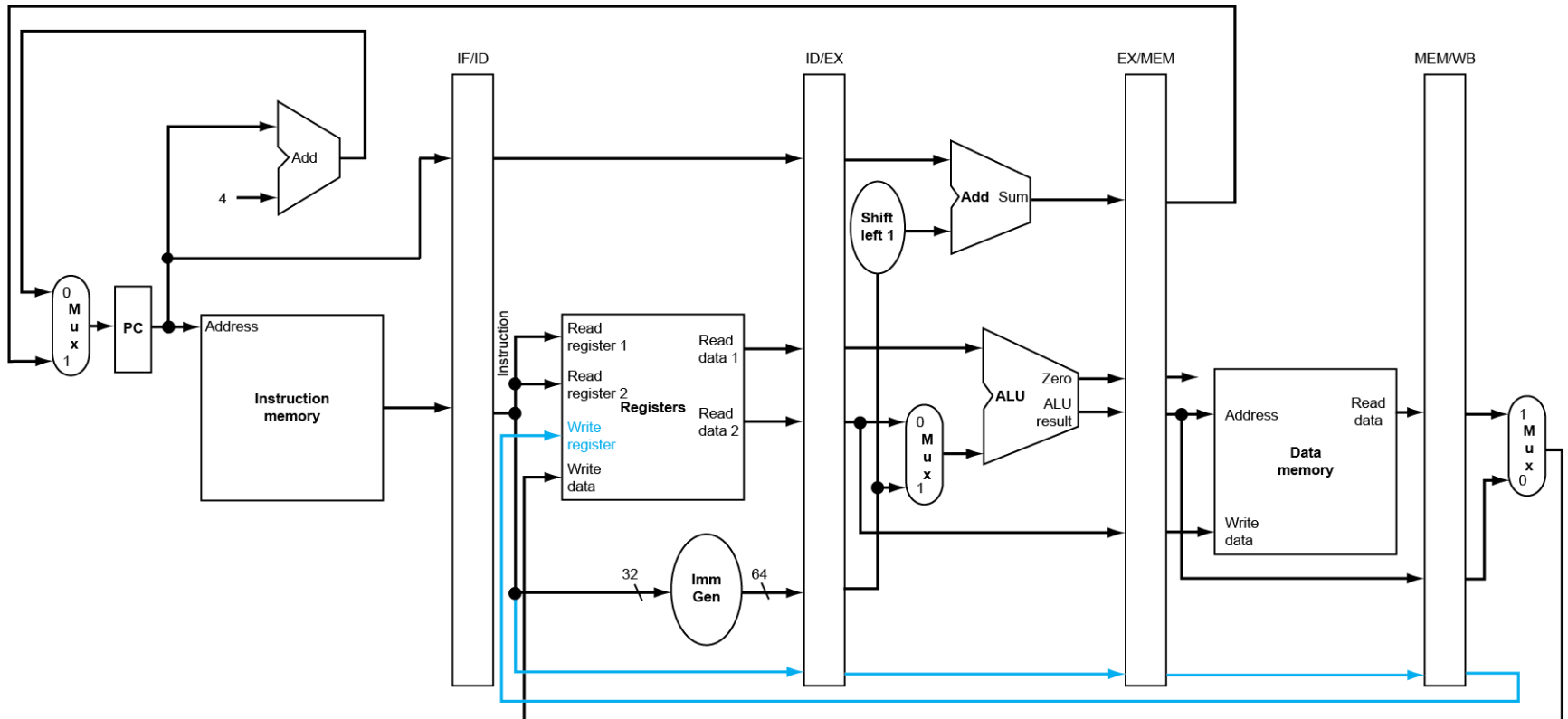
# MEM for Load



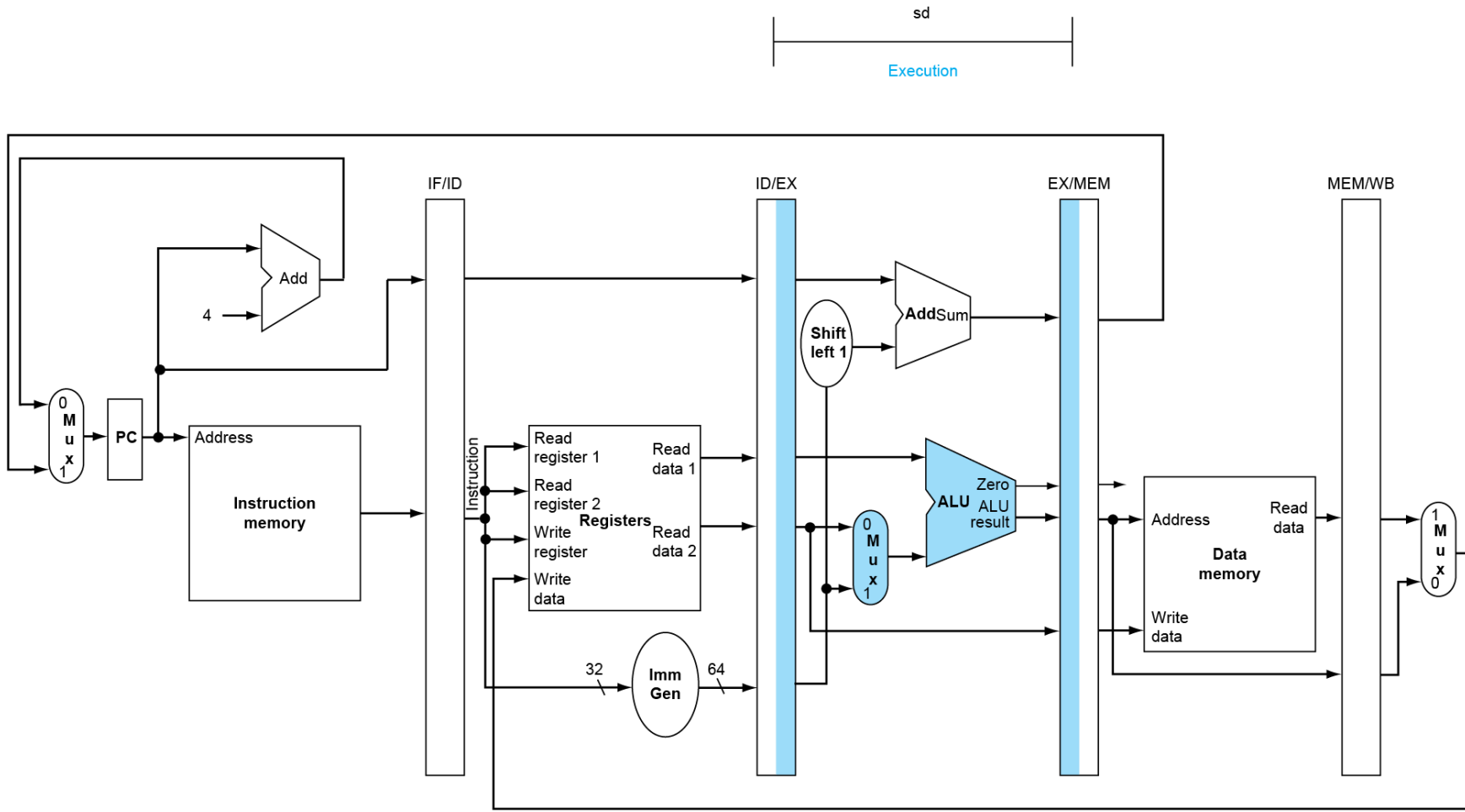
# WB for Load



# Corrected Datapath for Load

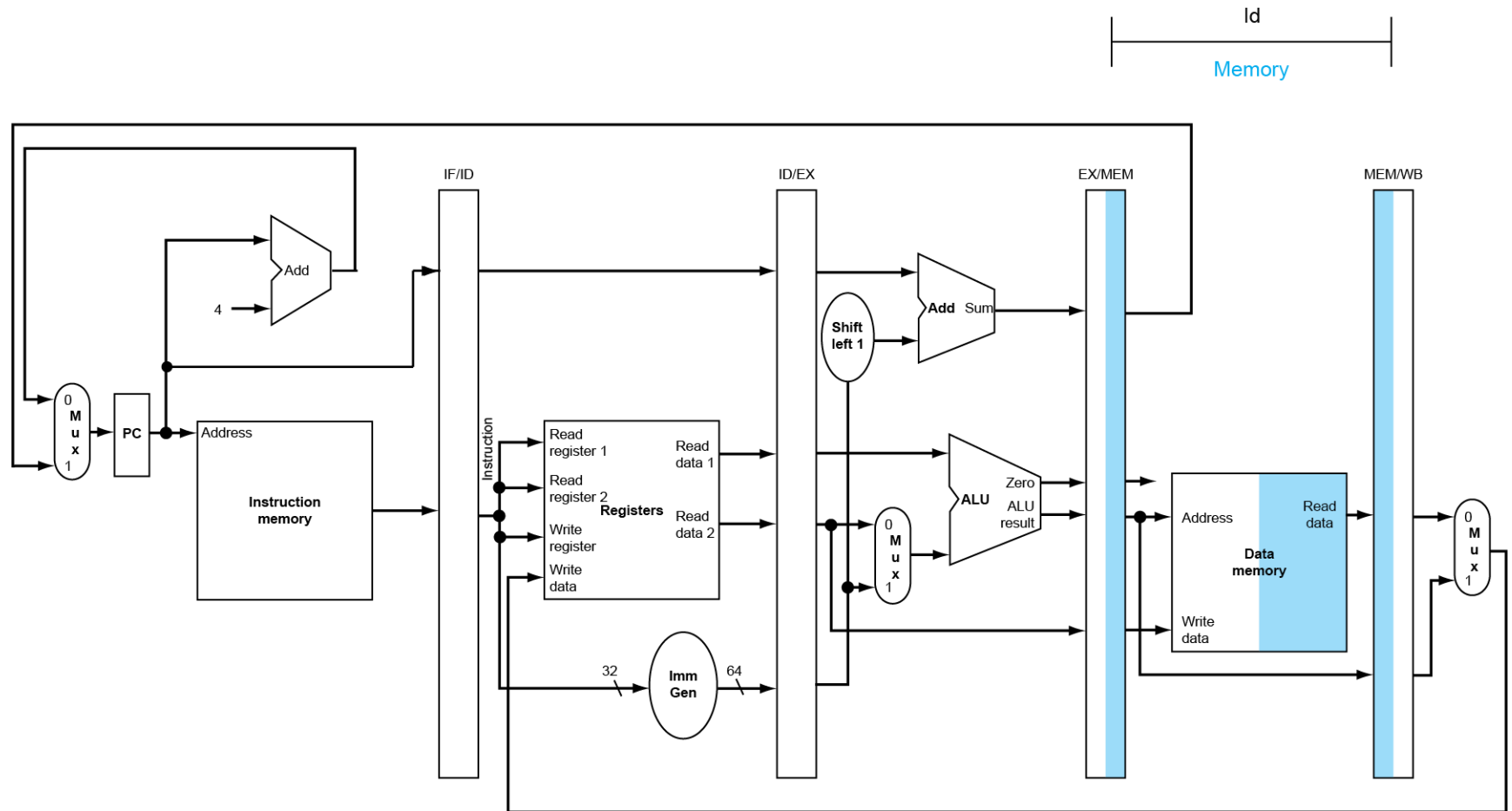


# EX for Store

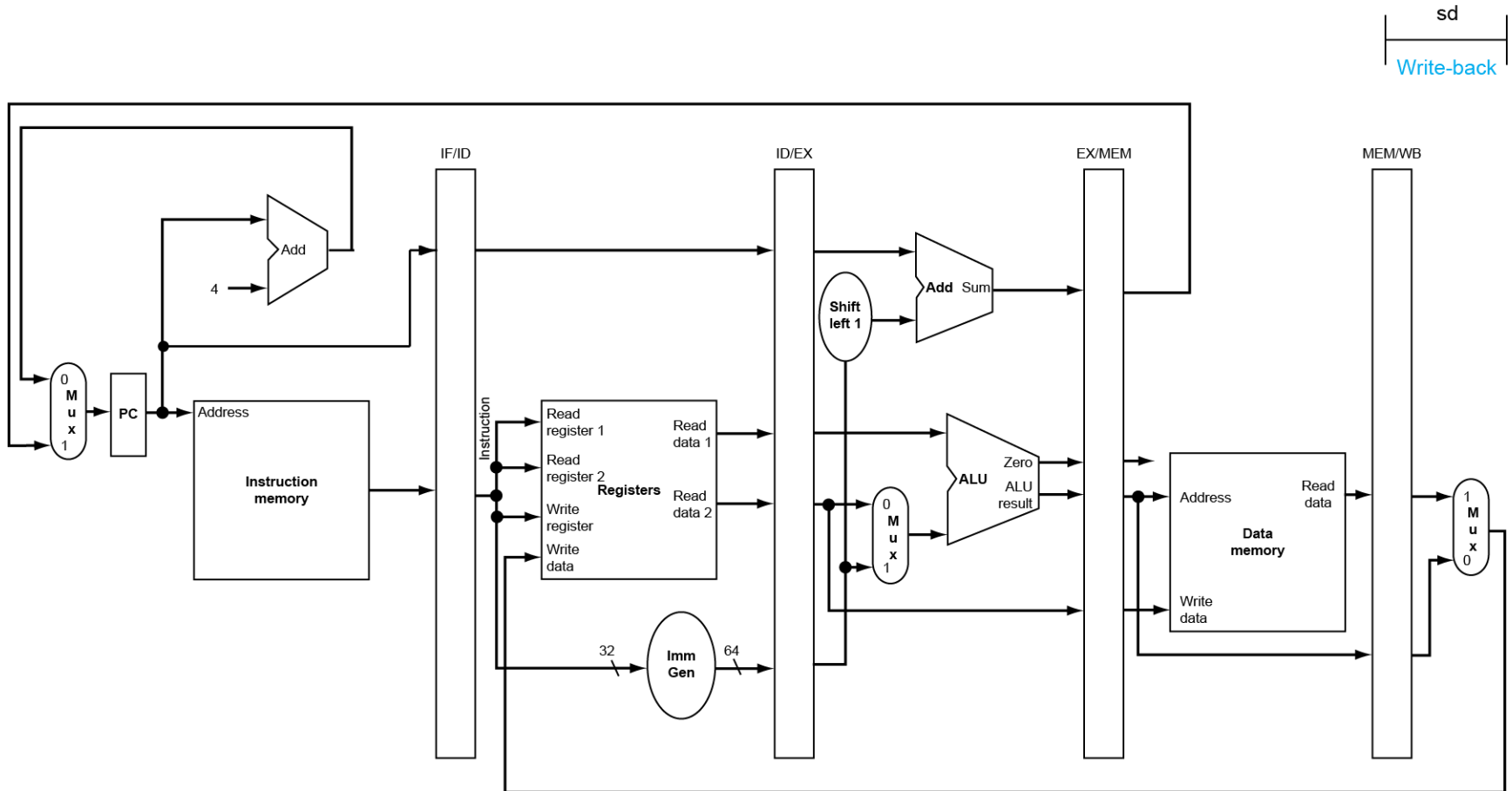




# MEM for Store

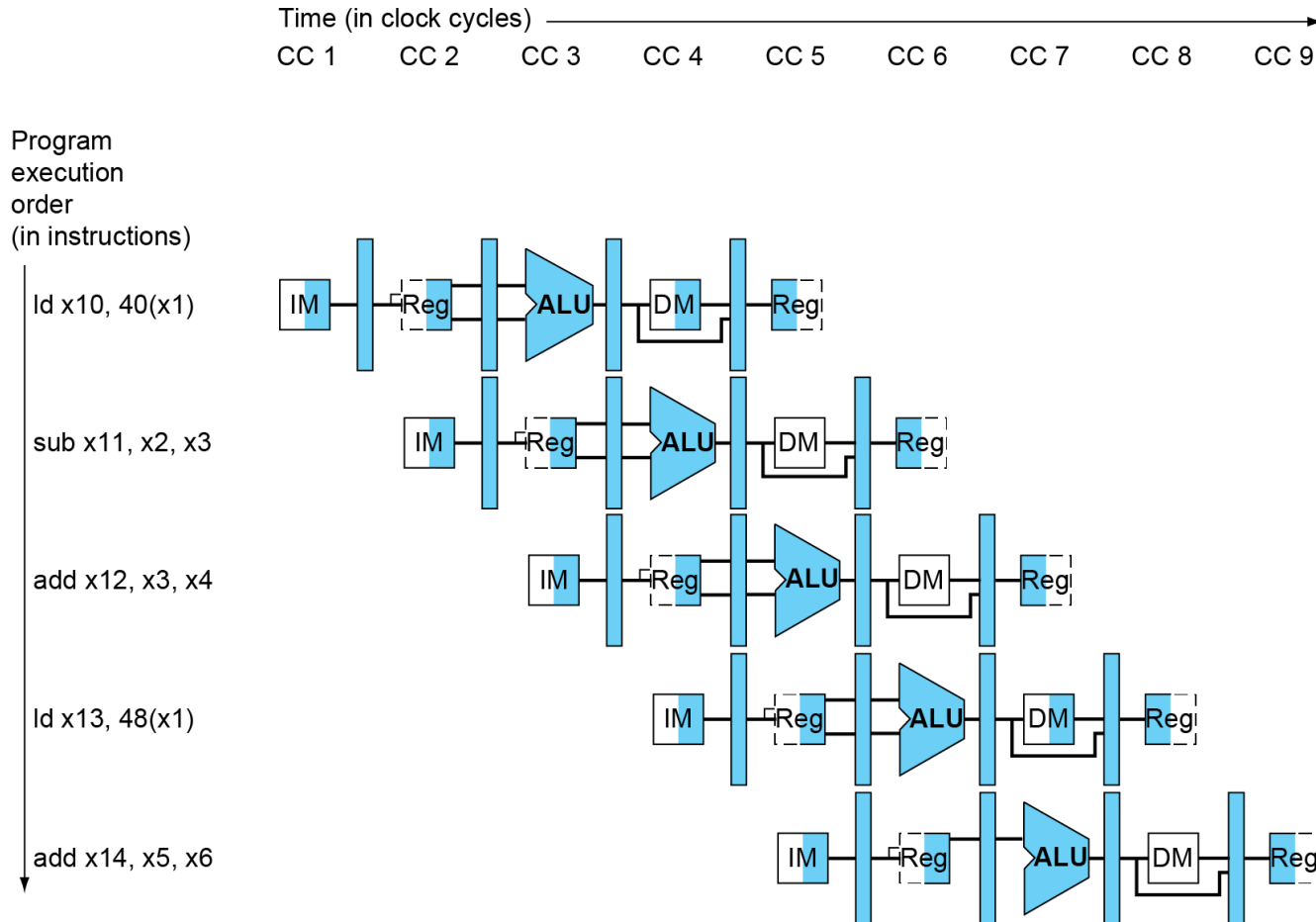


# WB for Store



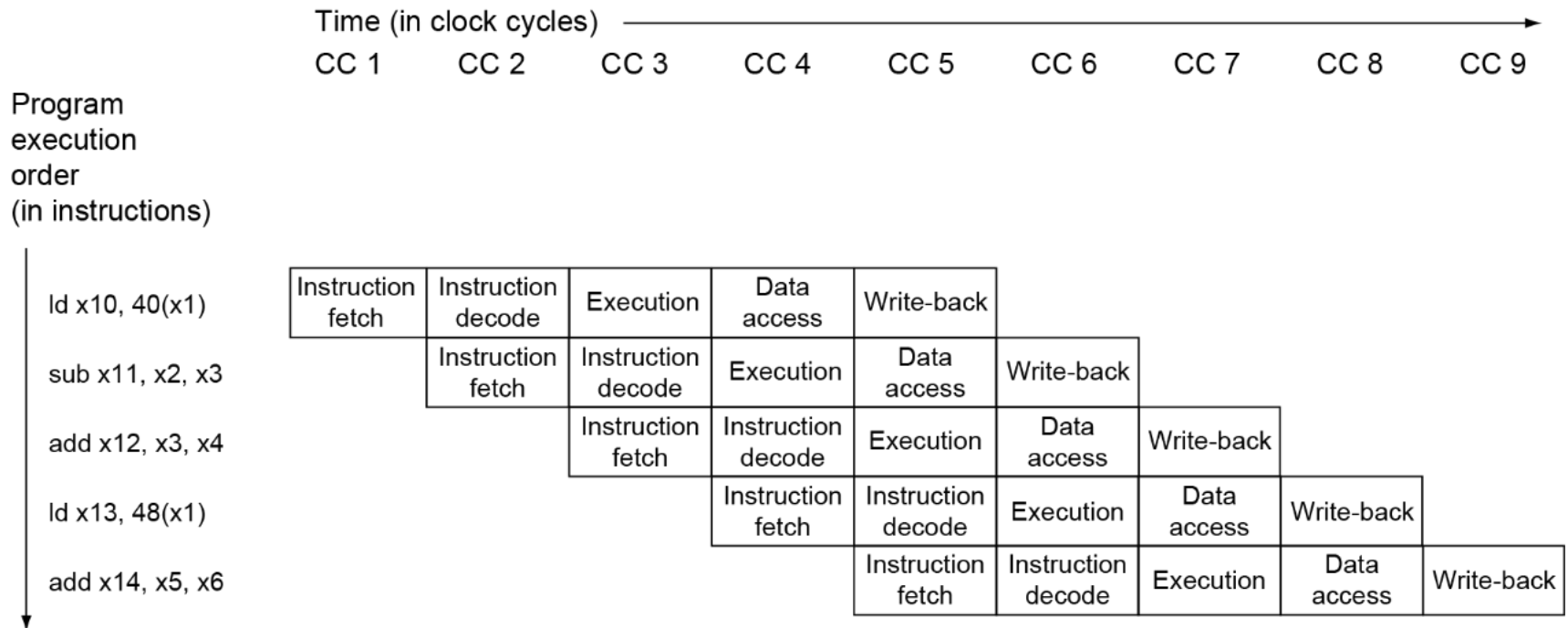
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



# Multi-Cycle Pipeline Diagram

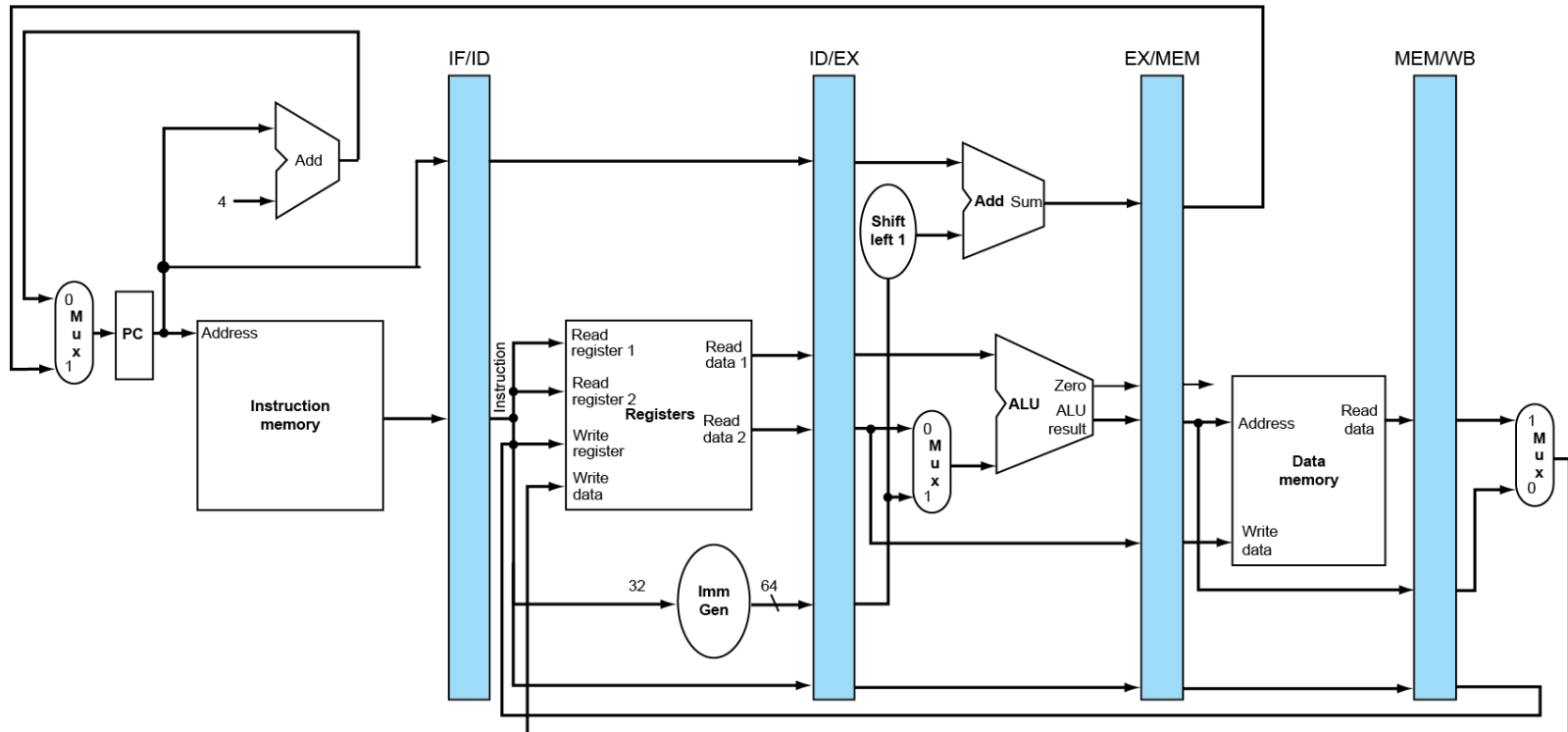
- Traditional form



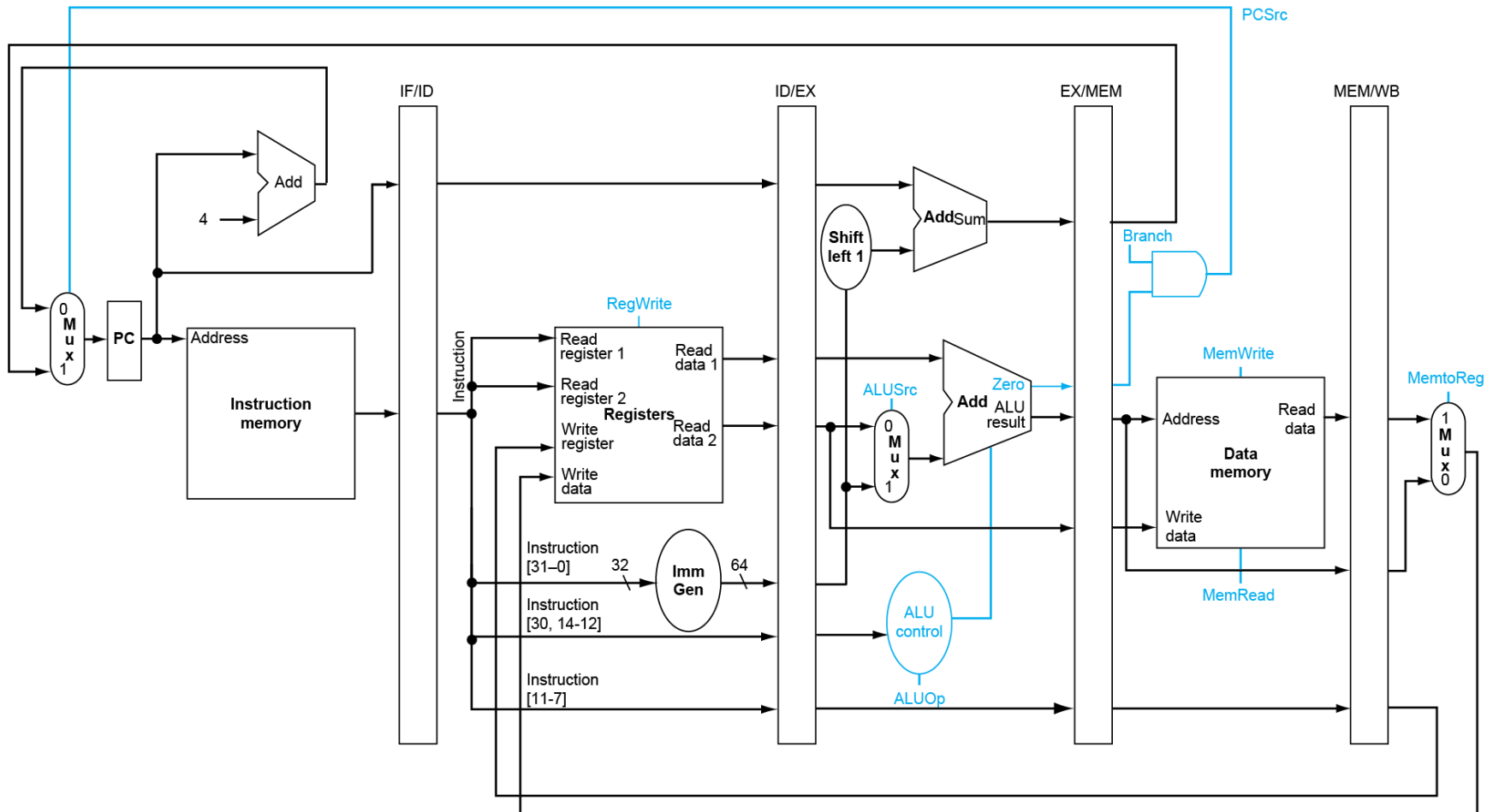
# Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

add x14, x5, x6	ld x13, 48(x1)	add x12, x3, x4	sub x11, x2, x3	ld x10, 40(x1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



# Pipelined Control (Simplified)



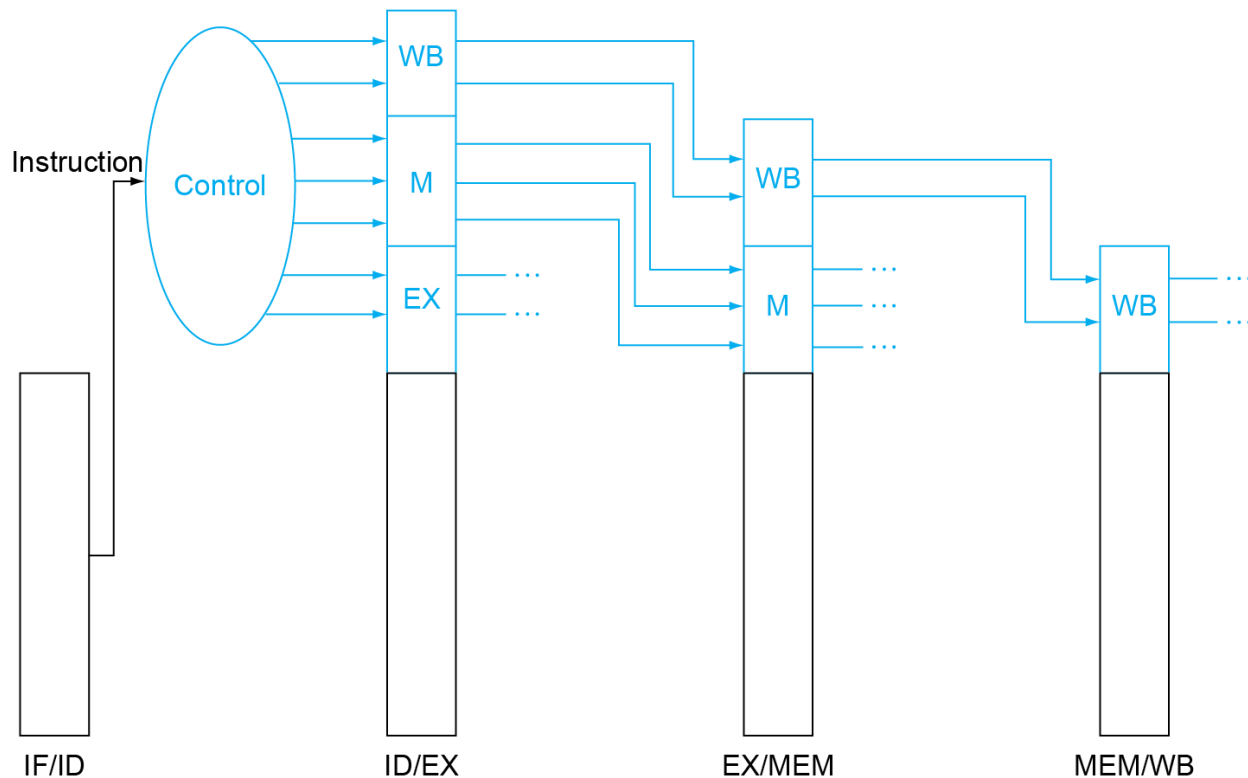
# Pipelined Control

- Control signals grouped by pipeline stages

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
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

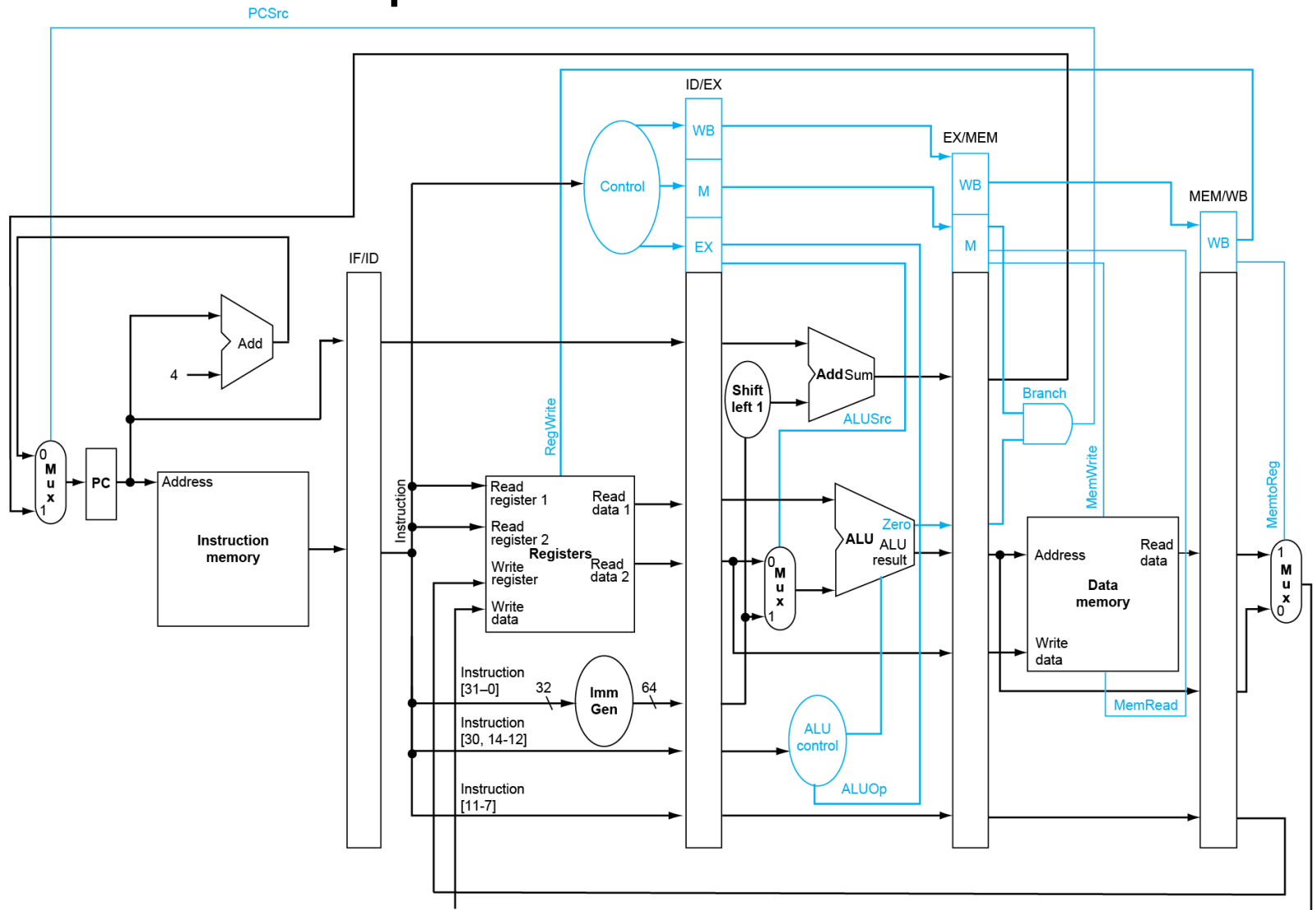
# Pipelined Control

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





# Pipelined Control



# Outline

- An Overview of Pipelining
- Pipelined Datapath and Control
- Data Hazards: Forwarding vs. Stalling

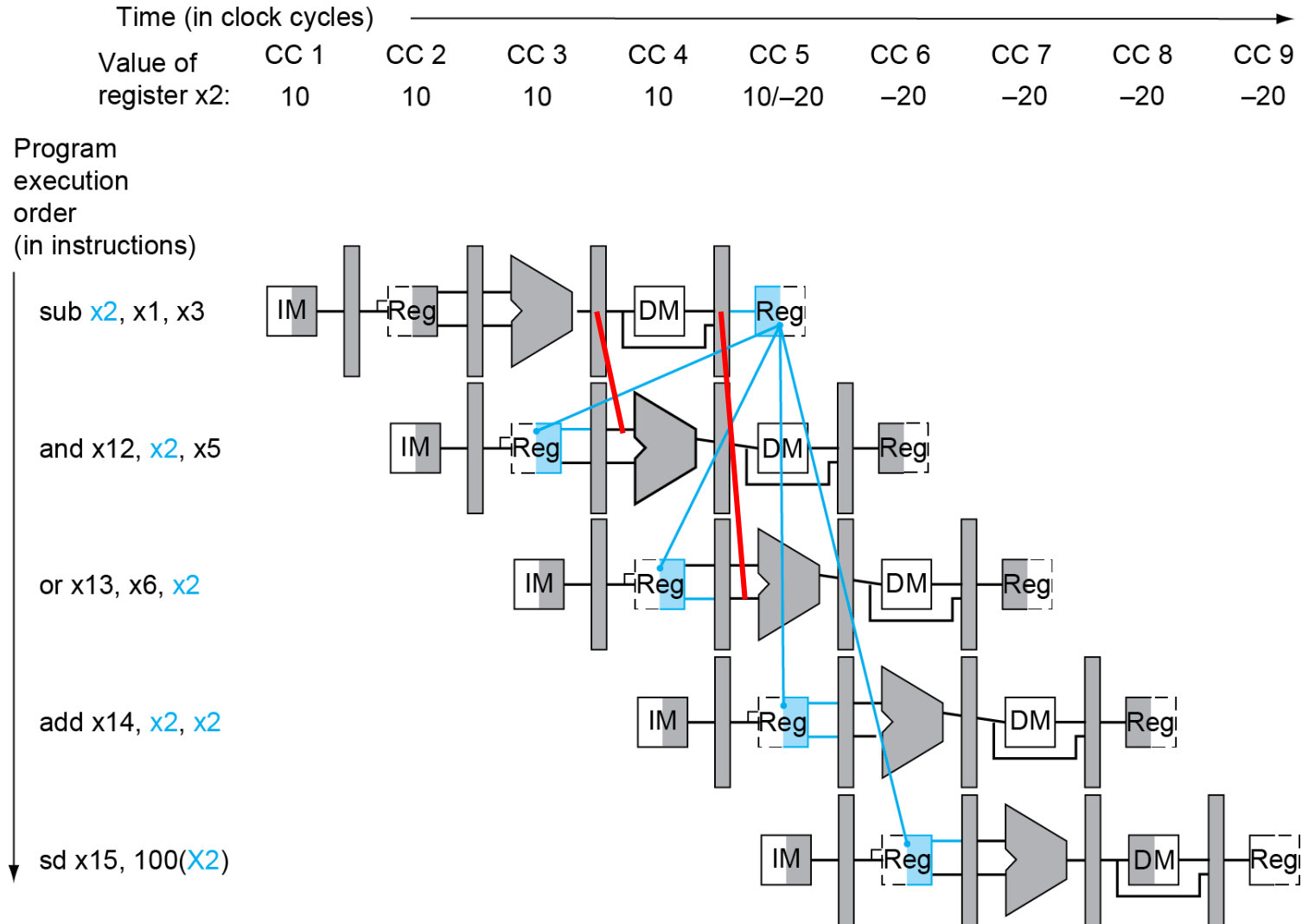
# Data Hazards in ALU Instructions

- Consider this sequence:

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

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

# Dependencies & Forwarding



# Detecting the Need to Forward

- 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

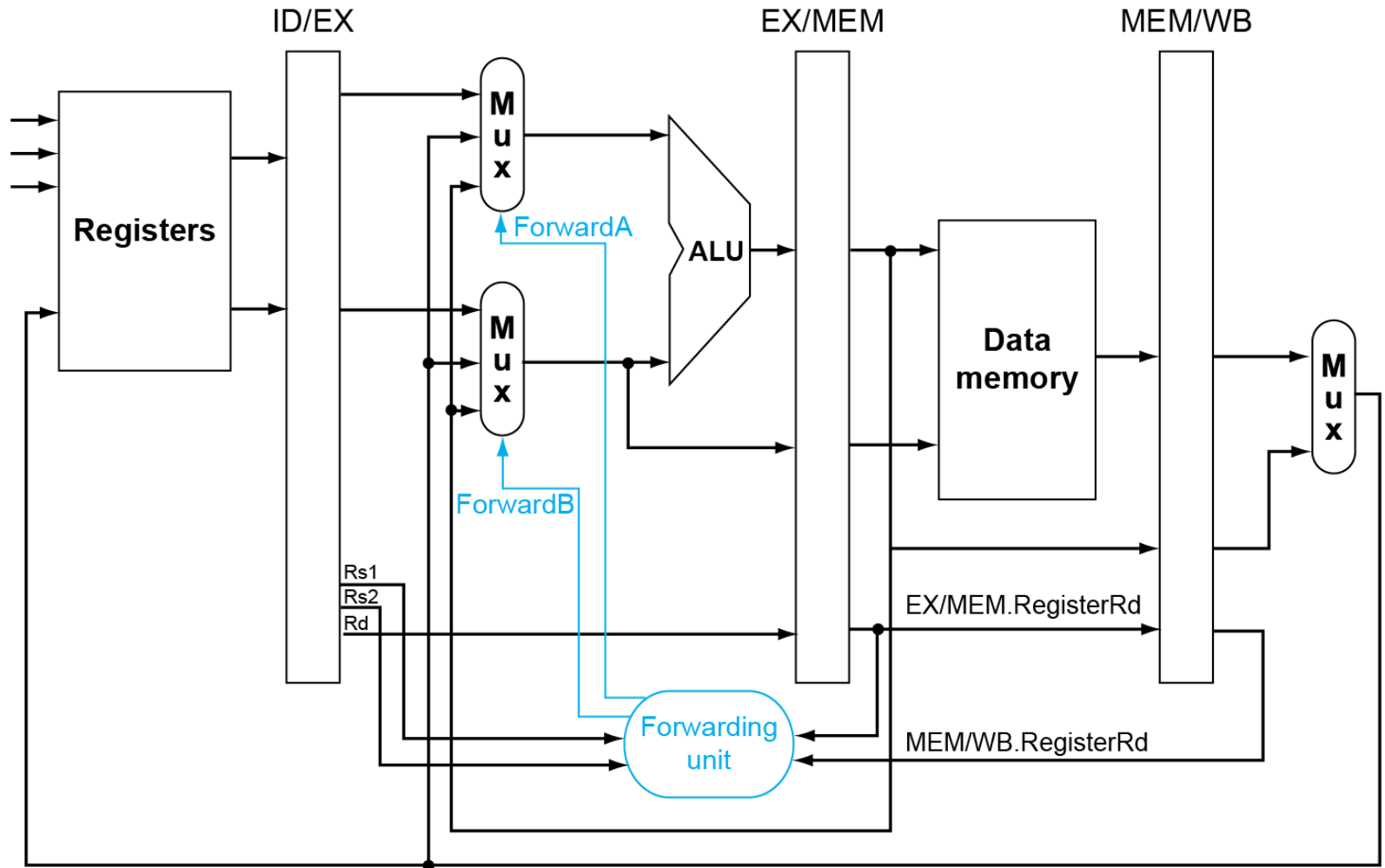
Fwd from  
EX/MEM  
pipeline reg

Fwd from  
MEM/WB  
pipeline reg

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

# Forwarding Paths



# 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.



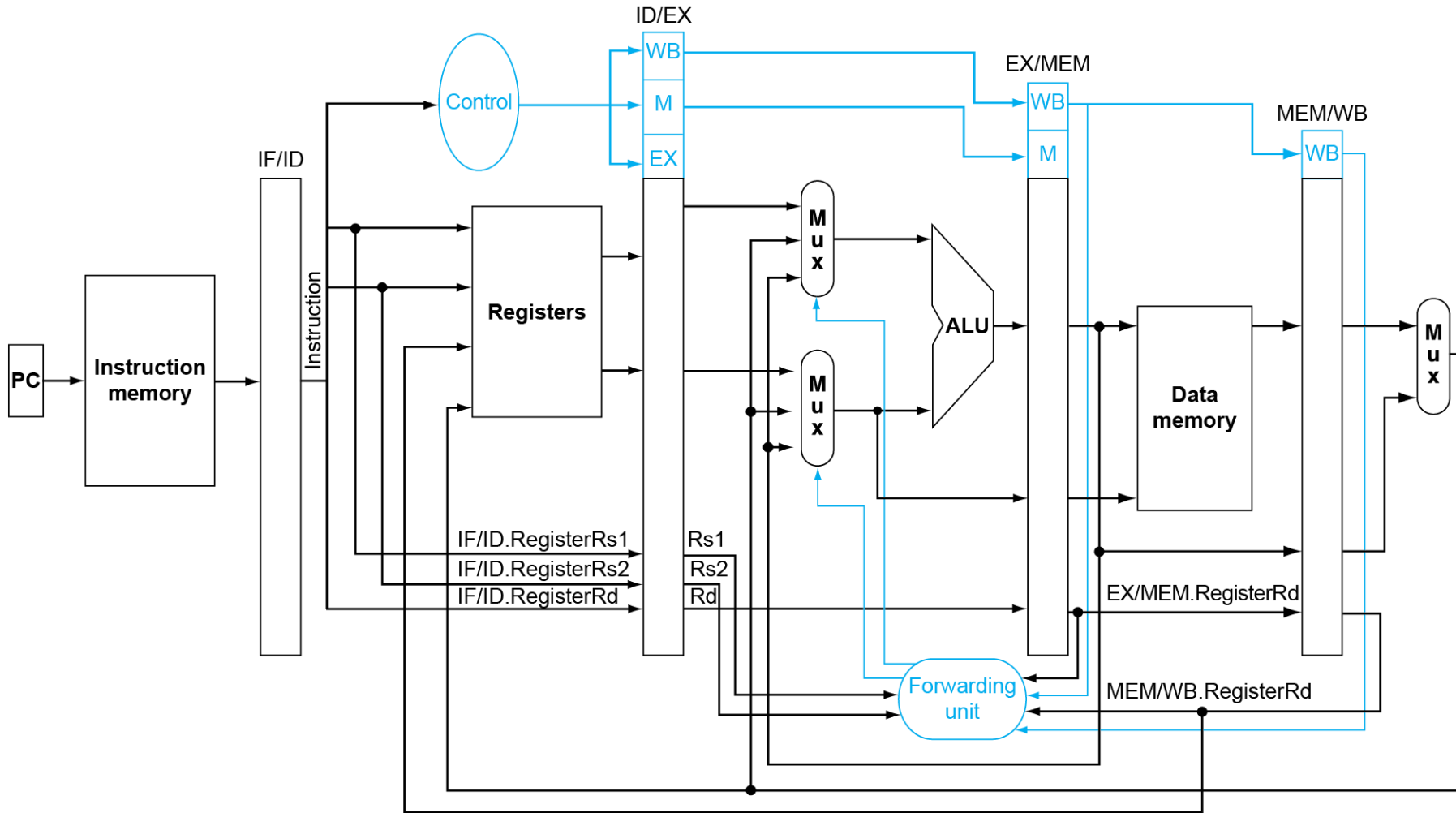
# Double Data Hazard

- Consider the sequence:
  - add x1, x1, x2
  - add x1, x1, x3
  - add x1, x1, x4
- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

# Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd  $\neq$  0)  
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  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  $\neq$  0)  
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

# Datapath with Forwarding



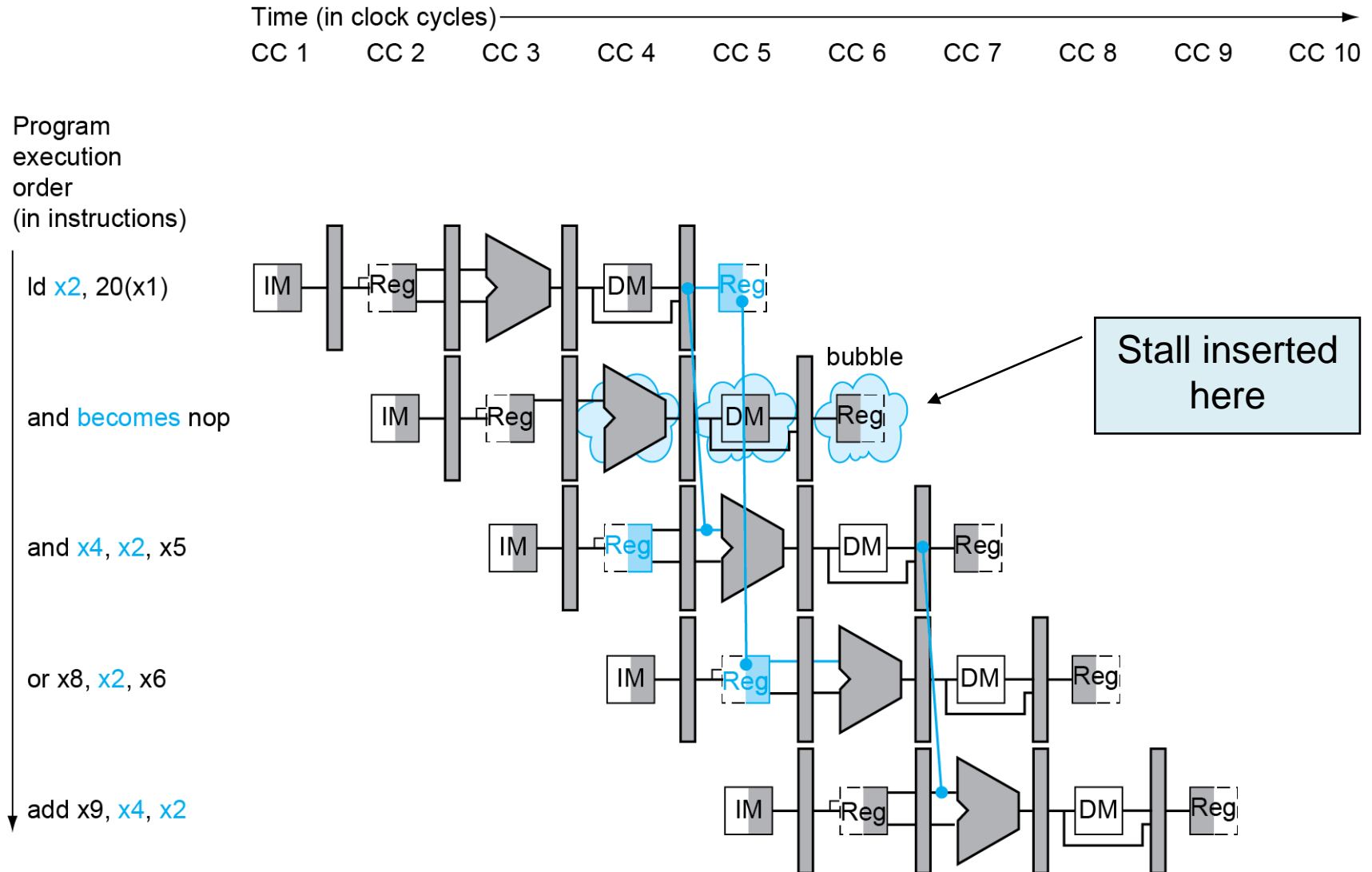
# Load-Use Hazard Detection

- 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) or  
(ID/EX.RegisterRd = IF/ID.RegisterRs1))
- If detected, stall and insert bubble

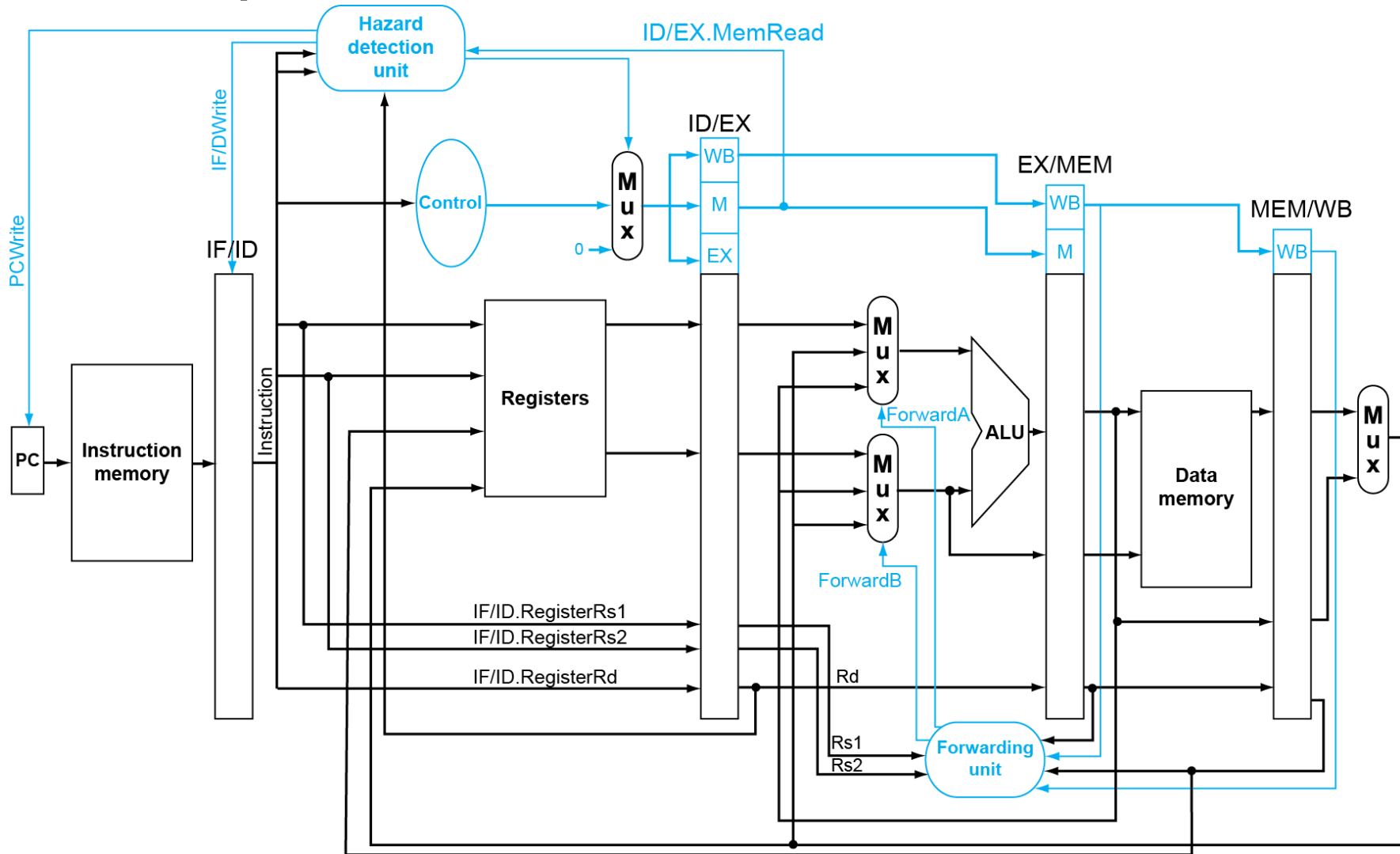
# How to Stall the Pipeline

- 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 1d
    - 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



# Questions?