

**COMP3710 (Class # 5176)**  
**Special Topics in Computer Science**  
**Computer Microarchitecture**

Convener: Shoaib Akram  
[shoaib.akram@anu.edu.au](mailto:shoaib.akram@anu.edu.au)



Australian  
National  
University



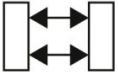
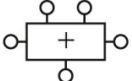

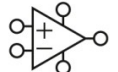


# Plan & Progress

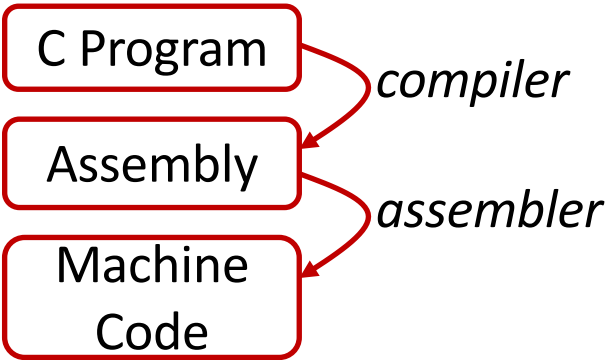
*Week 1: Motivation, overview, & ISA revision*

*Week 2: Digital logic & Single-Cycle MIPS/ARM CPU*

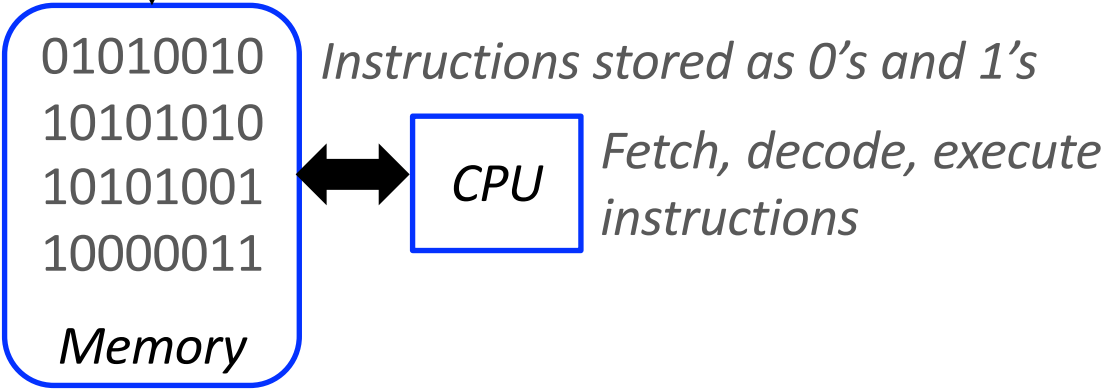
*This Week: Performance & In-Order Pipelined CPU*

# Big Picture

Application Software	>"hello world!"	Programs
Operating Systems		Device Drivers
Architecture		Instructions Registers
Micro-architecture		Datapaths Controllers
Logic		Adders Memories
Digital Circuits		AND Gates NOT Gates
Analog Circuits		Amplifiers Filters
Devices		Transistors Diodes
Physics		Electrons



*Software*  
*Hardware*  
*ISA is the boundary (Contract)*

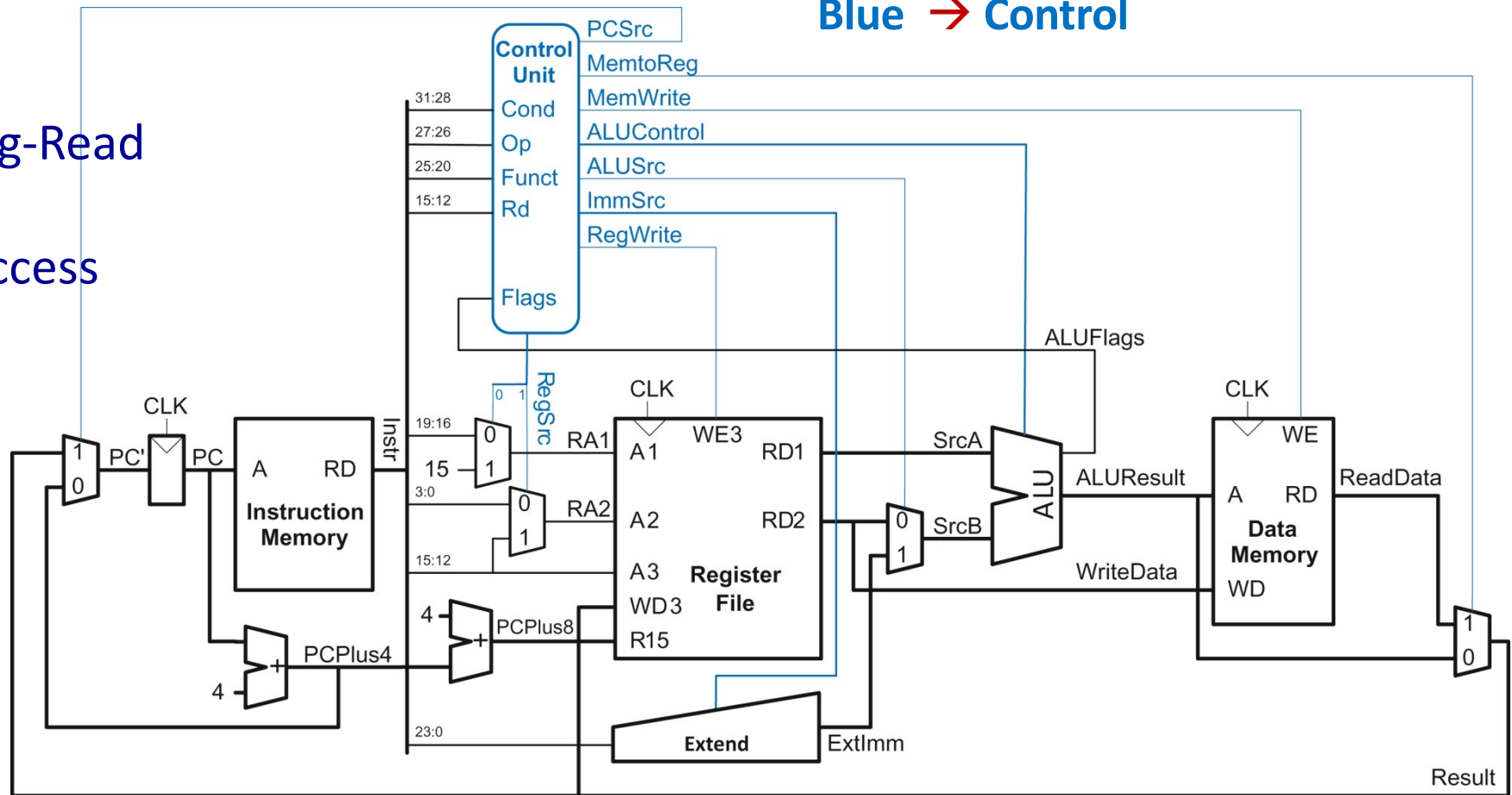


# Single-Cycle CPU Summary

## ARM

- Fetch
- Decode/Reg-Read
- Execute
- Memory Access
- Writeback

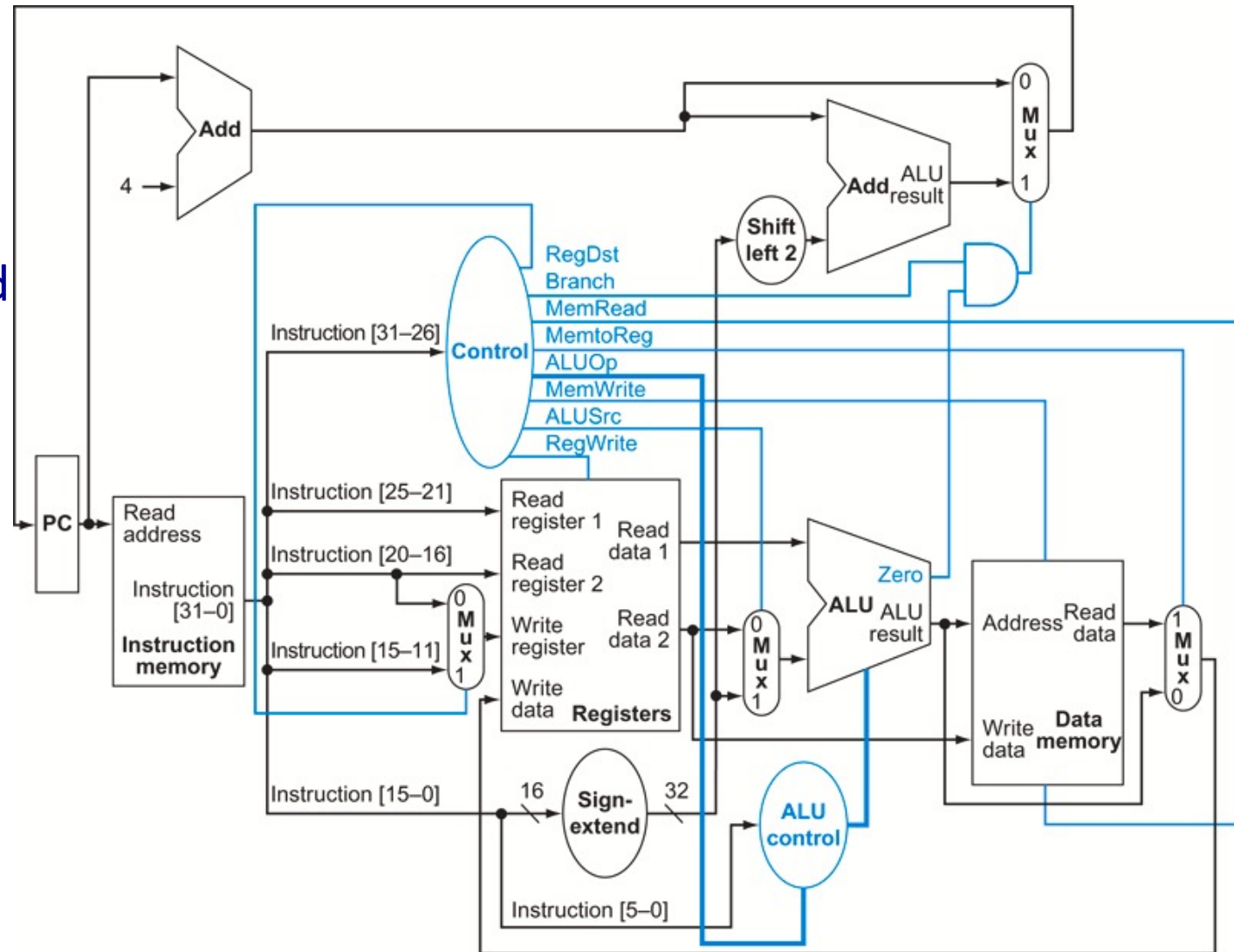
Black → Datapath  
 Blue → Control



# Single-Cycle CPU Summary

## MIPS

- Fetch
- Decode/Reg-Read
- Execute
- Memory Access
- Writeback



# Performance Analysis

- We want the fastest (**best performing**) computer for a task
  - How should we measure and report performance?
- Which metric is **fair** for comparing two computers?
  - Number of instructions
  - Clock frequency
  - Cycles per Instruction (**CPI**)
  - Number of cores

Important to understand the **true**, **gimmick-free** measure of computer performance



# # Instructions

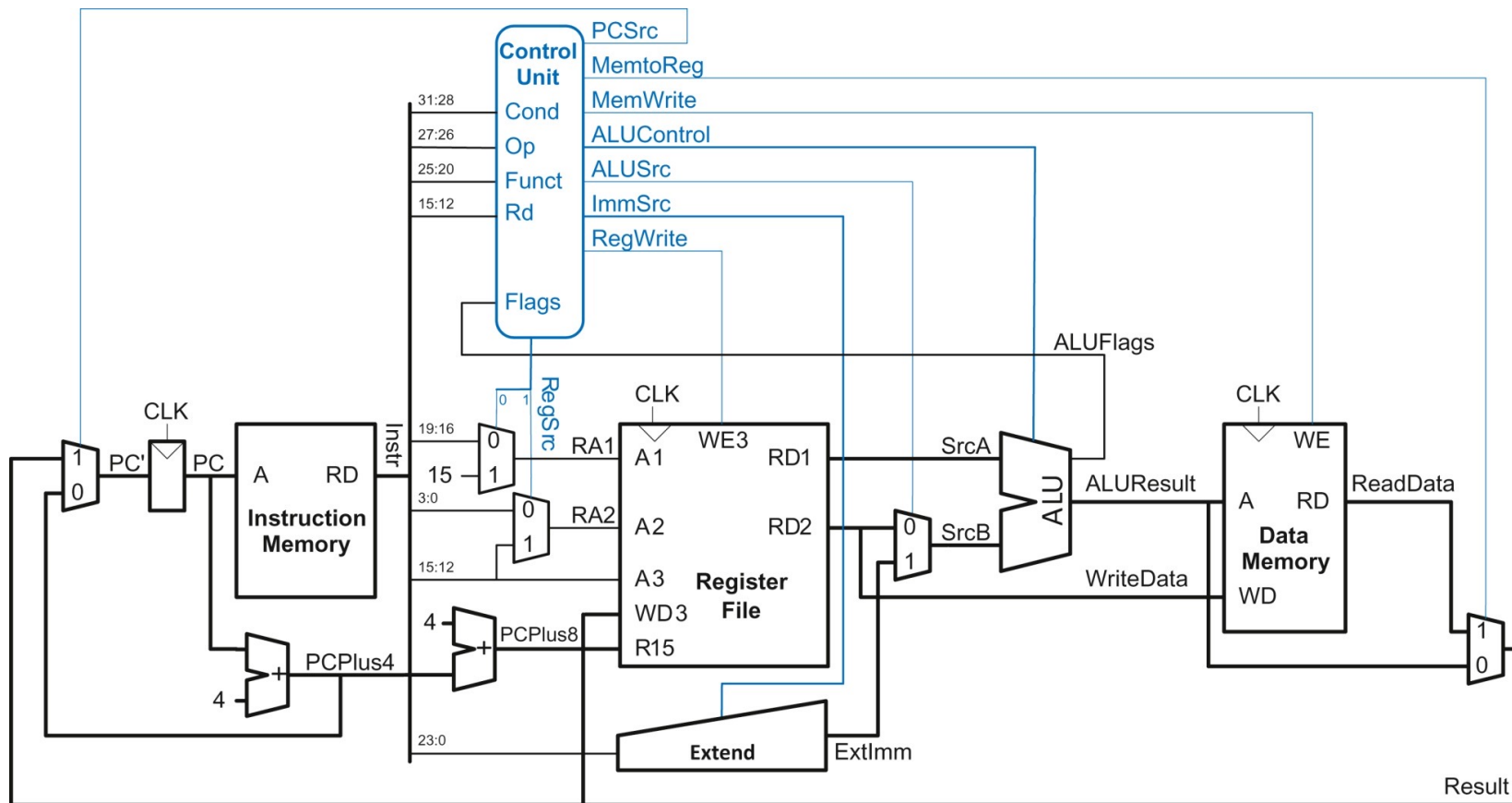
- RISC computer
  - Many more simple instructions
    - But simple hardware means shorter clock cycle
- CISC computer
  - Small number of instructions
    - But complex hardware means longer clock cycle

**Bottomline:** Number of instructions alone is not a good metric for quantifying the performance of applications

# CISC Instructions vs. Critical Path

`sum $dst-reg, $src1-reg, #offset($src2-reg)`

Changes to CPU datapath/ctrl to support `sum`?





# Clock Frequency

- Consider the two scenarios
  - Computer **A** has a faster clock rate than computer **B**
    - But **A** executes more instructions than **B**
  - Computer **A** has a faster clock rate than computer **B**
    - But **A** takes multiple cycles to execute an instruction
- Is **A** faster than **B**?

**Bottomline:** Clock frequency alone is not a fair metric for comparing the performance of **A** and **B**

# Cycles per Instruction (CPI)

- Cycles per instruction
  - Ratio of # cycles to # instructions
- A program has 10 instructions. Each instruction takes one cycle
  - Instructions = 10, Cycles = 10, CPI = 1
- A program has 10 instructions. Two out of 10 instructions take two cycles
  - Instructions = 10, Cycles = 12, CPI = 1.2
- The inverse of CPI is called IPC (Instructions per Cycle)
  - Instructions per cycle
  - For the above examples, IPC is 1 and 0.83

# CPI

- Reasons why each instruction can take multiple cycles
  - Multi-cycle CPU
  - Memory accesses take multiple cycles
  - Interrupts and exceptions
  - Stalls due to branches, jumps, page faults
- Memory accesses (loads) take variable # cycles because
  - Data may be present in faster (SRAM) cache
  - It may be present in slower main memory (DRAM)
  - Real DRAM has variable latency
    - There is a small SRAM cache called a row buffer for storing recently accessed data

# CPI

- Two computers A and B have the same CPI for a specific program.
  - Is there performance equivalent?
    - We need to know the cycle time
    - We need to know the # instructions
- **Question:** List two conditions which if satisfied make IPC a fair metric for comparing two microarchitectures.

**Bottomline:** *CPI alone is not a good metric for quantifying the performance of application A*

# Cycles per Instructions (CPI)

The # cycles a processor takes to execute a program divided by the total # instructions (*inverse is instructions per cycle or IPC*)

IPC is used in microarchitectural studies (*It reveals insight about the instruction throughput, bottleneck identification*)

It can be estimated using an architectural simulator or performance counter hardware on modern processors

# Execution Time

- The time it takes for a program to execute from start to finish is the only **true/golden** measure of performance

$$\text{Execution time} = (\text{\#instructions}) \left( \frac{\text{cycles}}{\text{instruction}} \right) \left( \frac{\text{seconds}}{\text{cycle}} \right)$$

- seconds per cycle = cycle time
- Execution time is measured in seconds
- Golden metric for quantifying computer performance!

# Execution Time

$$\text{Execution time} = (\text{\#instructions}) \left( \frac{\text{cycles}}{\text{instruction}} \right) \left( \frac{\text{seconds}}{\text{cycle}} \right)$$

## # instructions

- Depends on the ISA, skill of programmer, compiler, algorithm, programming language

## cycles per instruction

- Depends on the microarchitecture esp. memory system, compiler, ISA, programming language, algorithm

## seconds per cycle (a.k.a. clock cycle time, clock rate)

- critical path, circuit technology, type of adders, gate-level details, ISA

# Exercise: Perf Analysis

- Find the time it takes to execute a program with 100 billion instructions on a single-cycle CPU in 16 nm CMOS manufacturing process. See the table for delays of logic elements.

Parameter	Delay (ps)
$t_{pcq\_PC}$	40
$t_{mem}$	200
$t_{dec}$	70
$t_{mux}$	25
$t_{RFread}$	100
$t_{ALU}$	120
$t_{RFsetup}$	60

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$



# A Performance Quiz Series

Pick all fair metrics to compare two alternative microarchitectures (same ISA/program/frequency)

- #instructions
- cycle time
- cycles per instruction (CPI)
- execution time

# A Performance Quiz Series

Pick all fair metrics to compare two alternative compilers  
(same ISA/program/microarchitecture)

- #instructions
- cycle time
- cycles per instruction (CPI)
- execution time

# Exercise

Alternative compiled code sequences using instructions in classes A, B, and C

Class	A	B	C
CPI for class	1	2	3
# instructions sequence 1	2	1	2
# instructions sequence 2	4	1	1

■ Sequence 1: # instructions = 5

- Clock Cycles  
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$   
 $= 10$

- Avg. CPI =  $10/5 = 2.0$

■ Sequence 2: # instructions = 6

- Clock Cycles  
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$   
 $= 9$

- Avg. CPI =  $9/6 = 1.5$

# A Performance Quiz Series

Is instructions per second (IPS) a fair metric for:

- comparing two microarchitectures
- comparing two ISAs
- comparing two compilers
- comparing two algorithms

*Millions of instructions per second (MIPS) was once used for marketing computers (measured against a publicly known benchmark) **Please read Ch-1, Sec 1.6, PH1***

# A Performance Quiz Series

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Which components of performance are affected by:

- (A) Algorithm
- (B) Programming language
- (C) Compiler
- (D) ISA

# A Performance Quiz Series

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Which components of performance are affected by:

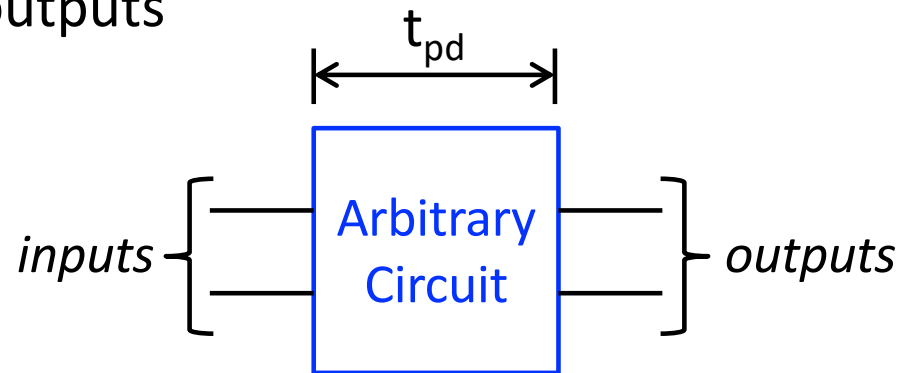
- (A) Algorithm (*# instructions, CPI*)
- (B) Programming language (*# instructions, CPI*)
- (C) Compiler (*# instructions, CPI*)
- (D) ISA (*all three*)

# Measurement Methodology

- The **good practice**: Take a program of interest and measure its execution time
- The **better practice**: Take a collection of programs like the programs of interest, and measure their performance
  - In case you do not have the program yet
  - In case someone wants to measure perf. of your machine independently
- This collection of programs is called a *benchmark suite*
  - Dhrystone and CoreMark (embedded systems)
  - SPEC (Standard Performance Evaluation Corporation)
  - SPEC CPU is standard suite for high-performance CPUs

# Speed of a Circuit

At a high level, an arbitrary digital circuit processes a group of inputs and produces a group of outputs



We need metrics to quantify the speed with which we can process inputs to produce outputs (i.e., the performance of a circuit)

- **Latency:** The time required to produce one group of outputs once the inputs arrive (propagation delay, end-to-end latency)
- **Throughput:** The number of input groups processed per unit of time



# Example: Latency/Throughput

- What is the latency and throughput for a tray of cookies?
  - Step # 1: Roll cookies (5 minutes)
  - Step # 2: Bake in the oven (15 minutes)
  - Once cookies are baked, start another tray
- Latency (**hours/tray**):
- Throughput (**trays/hour**):



# Parallelism

Many scenarios in the real-world requires us to increase the throughput of the digital system

- # add operations per second (ALU)
- # instructions per second (CPU)

Parallelism is the key technique digital systems use to increase throughput

- Process several inputs at the same time

# Defining a Task

**Task:** The process of producing a group of outputs from a group of inputs can be considered a task

- Add/Subtract
- Fourier transform

# Spatial Parallelism

**Spatial Parallelism:** Use multiple copies of hardware (circuit) to get multiple tasks done at the same time



- Suppose a task has a latency of  $L$  seconds
- **No spatial parallelism:** Throughput is  $1/L$  (one task per  $L$  seconds)
- **$N$  copies of hardware:** Throughput is  $N/L$  ( $N$  tasks per  $L$  seconds)
- Gain in throughput (**speedup**) =  $N$

# Note on Latency

**Spatial Parallelism** does not improve (reduce) the latency of the circuit. We can finish more tasks per unit of time. But each task still takes  $L$  seconds

# Temporal Parallelism

**Temporal Parallelism (pipelining):** Break down a circuit into stages, where each task passes through all stages, and multiple tasks are spread through stages

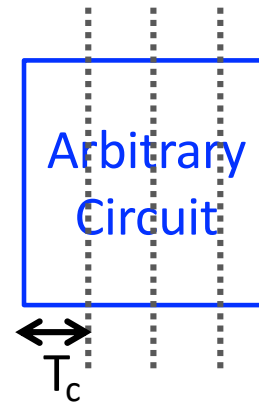
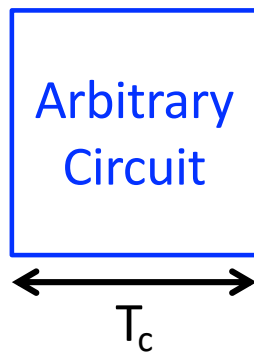
## Automotive pipeline

- Build multiple cars in parallel
- Each car goes through all stages
- Each stage requires different work
- All stages should take (roughly) the same time



# Pipelining

If a task is broken into  $N$  stages, and all stages are of equal length, then the throughput is  $N/L$



- *The challenge of pipelining is to find stages of equal length*
- *Let's go back to baking cookies*

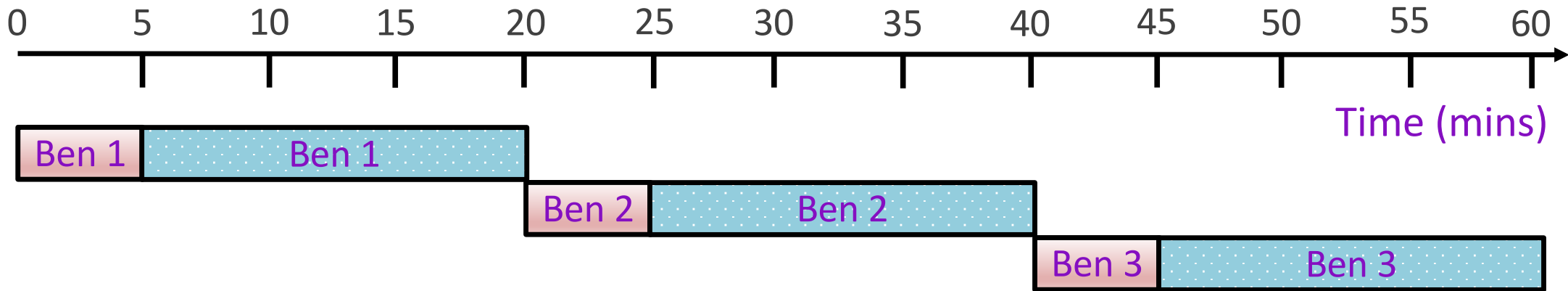
# Cookie Parallelism

Ben and Jon are making cookies. Let's study the latency and throughput of rolling/baking many cookie trays with

- No parallelism
- Spatial parallelism
- Pipelining
- Spatial parallelism + pipelining



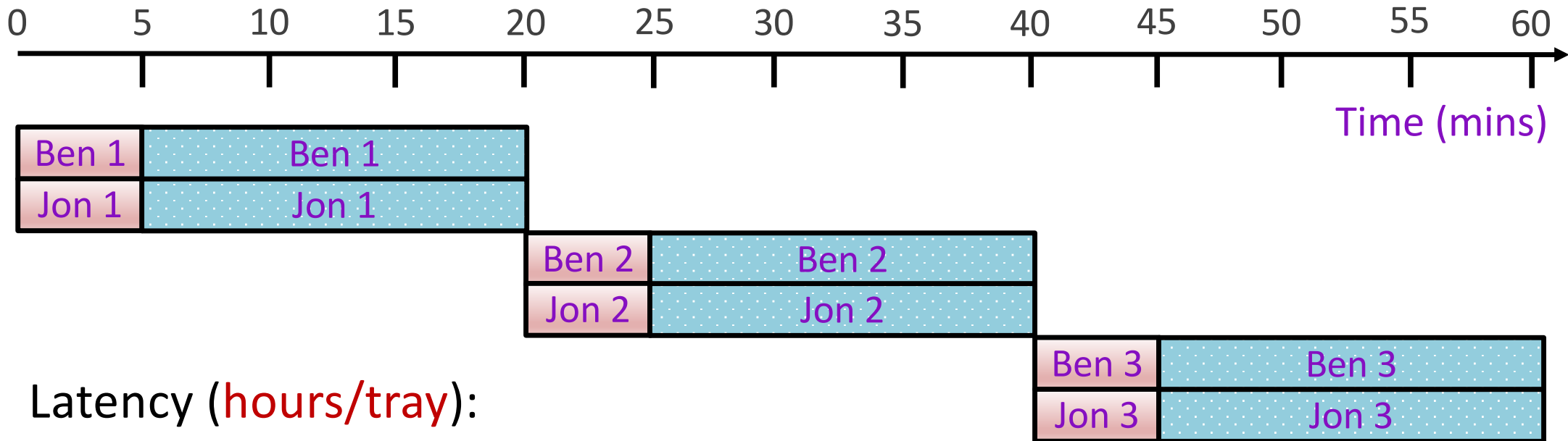
# No Parallelism (Ben Only)



Latency (**hours/tray**):

Throughput (**trays/hour**):

# Spatial Parallelism (Ben & Jon)

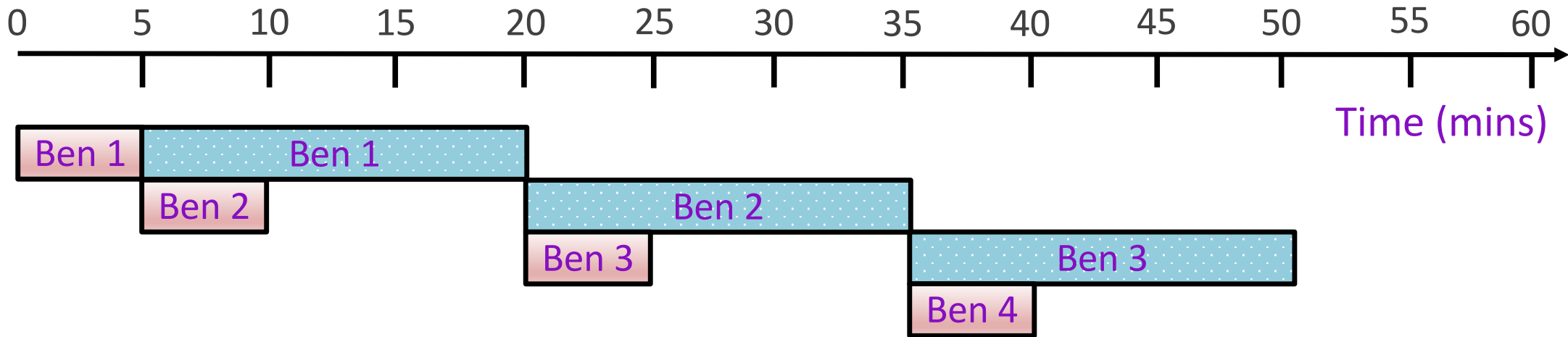


Latency (**hours/tray**):

Throughput (**trays/hour**):

Note: Jon owns a tray and oven (hardware duplication)

# Pipelining (Ben Only)

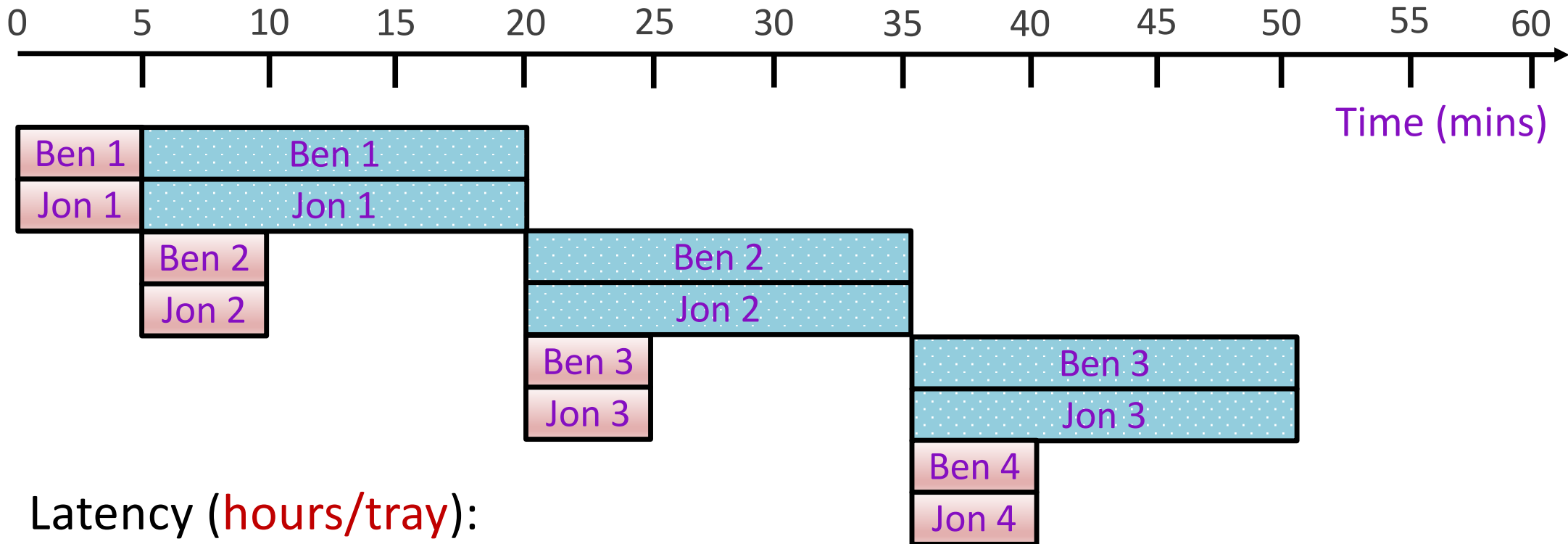


Latency (**hours/tray**):

Throughput (**trays/hour**):

Note: Ben decides not to waste a separate tray and oven

# Spatial + Temporal Parallelism



Latency (**hours/tray**):

Throughput (**trays/hour**):

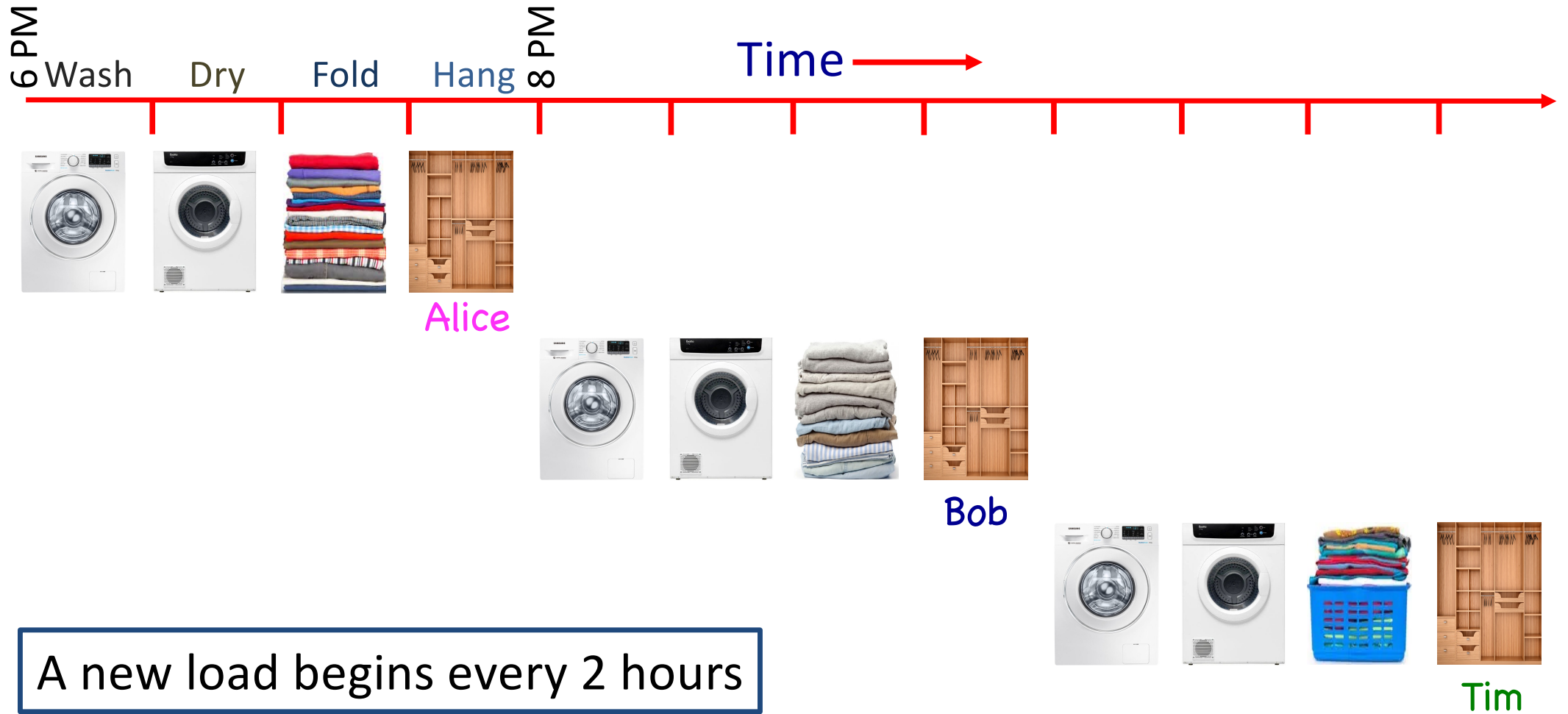
# Answers Explained

- **No parallelism**
  - Latency is clearly 20 minutes (1/3 hours/tray)
  - Throughput is 3 trays per hour
- **Spatial parallelism**
  - Latency remains unchanged as it still takes 20 mins to finish a tray
  - Throughput is doubled via duplication: 6 trays per hour
- **Pipelining**
  - Latency for a single tray remains unchanged
  - Throughput: Ben puts a new tray in the oven every 15 minutes, so the throughput is 4 trays per hour
  - Note that in the first hour, Ben loses 5 minutes to fill the pipeline
- **Spatial parallelism + pipelining**
  - Latency remains unchanged
  - Throughput: Ben & Jon combo puts two trays in the oven every 15 minutes, so the throughput is 8 trays per hour

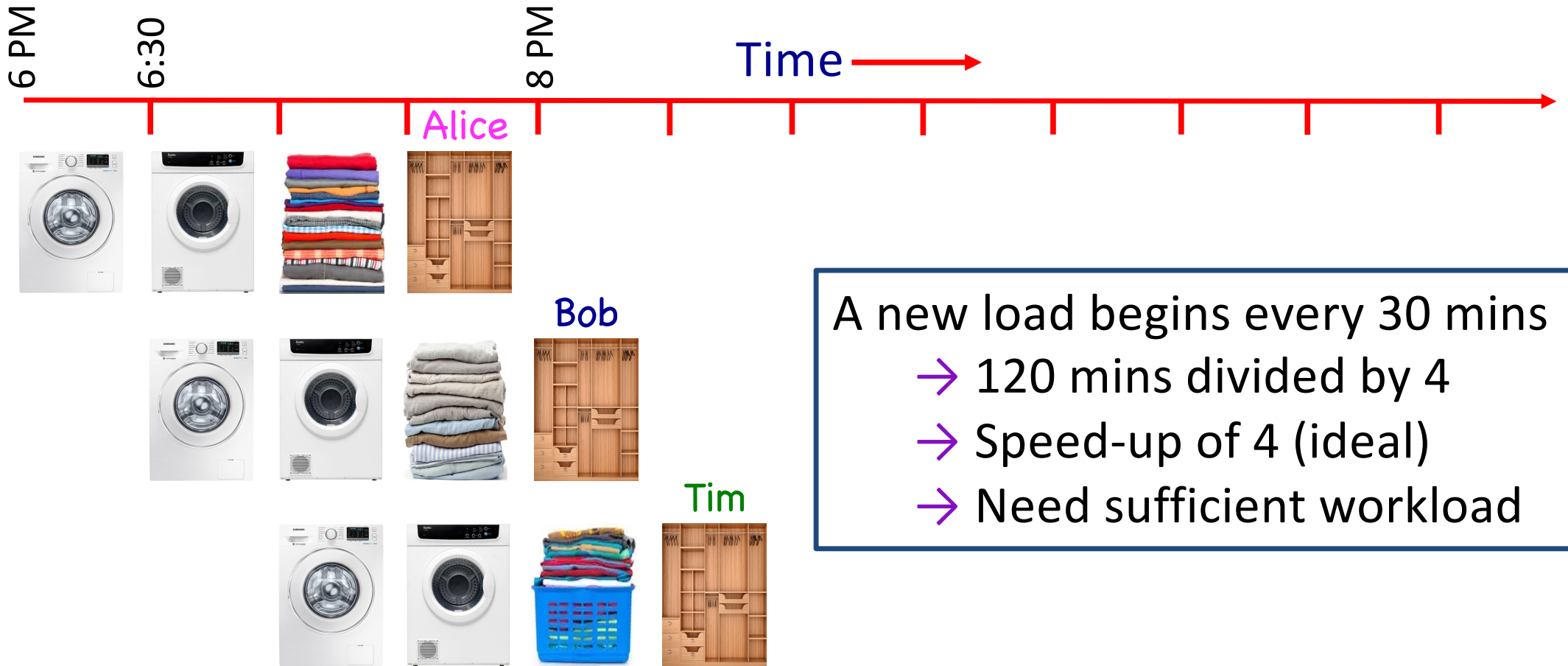
# Pipelining Circuits

- Divide a large combinational block/circuit into shorter stages
- Insert registers between the stages
  - The outputs from one stage are copied into a register and communicated to the next stage
- Run the pipelined circuit at a higher clock frequency
  - Each clock cycle, data flows through the pipeline from left to the right
  - Multiple tasks can be spread across the pipeline

# Sequential Laundry



# Pipelined Laundry



A new load begins every 30 mins

- 120 mins divided by 4
- Speed-up of 4 (ideal)
- Need sufficient workload



# Pipelined Performance

Our pipelined laundry system has four stages

- $k = 4$ , *Speed-up approaches  $k$  for large workloads*
- **Pipeline fill time:** # time units it take for the first load to reach the last stage of the pipeline
- **Pipeline drain time:** After the last load has entered the system, an additional  $k$  time units are needed to drain the pipeline

**Pipelining paradox:** *Each load still takes the same time. Pipelining improves the throughput of the laundry system via **parallelism***

# Pipelining Instruction Execution

We can pipeline the single-cycle implementation of the MIPS ISA

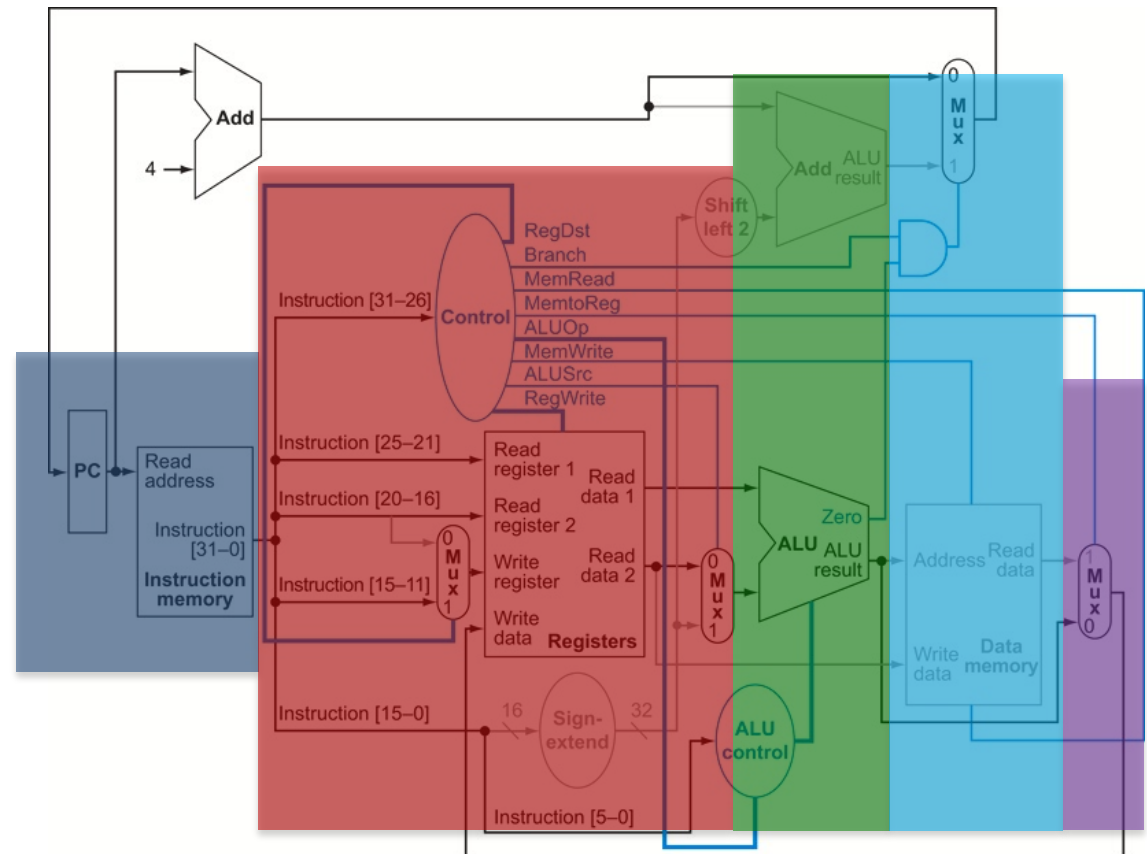
Fetch

Decode + Register Read

Execute (Address gen)

Memory Access

Writeback



# Performance Analysis

Find the average time b/w instructions of a single-cycle implementation to a pipelined implementation.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

# Performance Analysis

Find the average time b/w instructions of a single-cycle implementation to a pipelined implementation.

*internal fragmentation*

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

*single-cycle cycle time*

*pipelined cycle time*

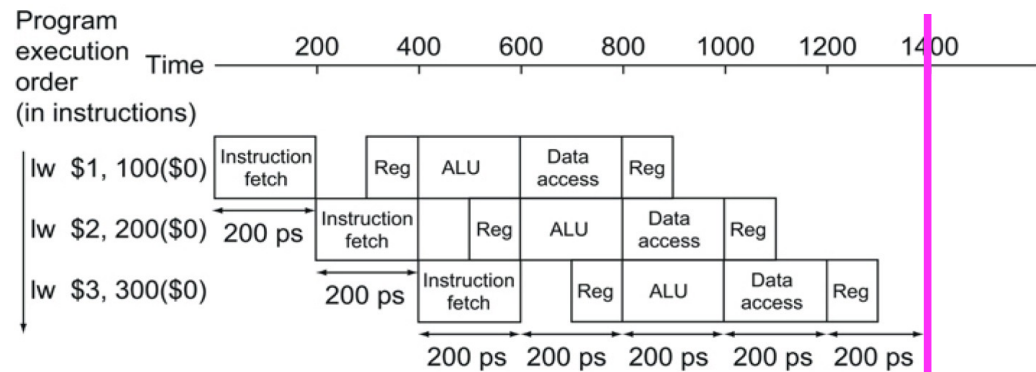
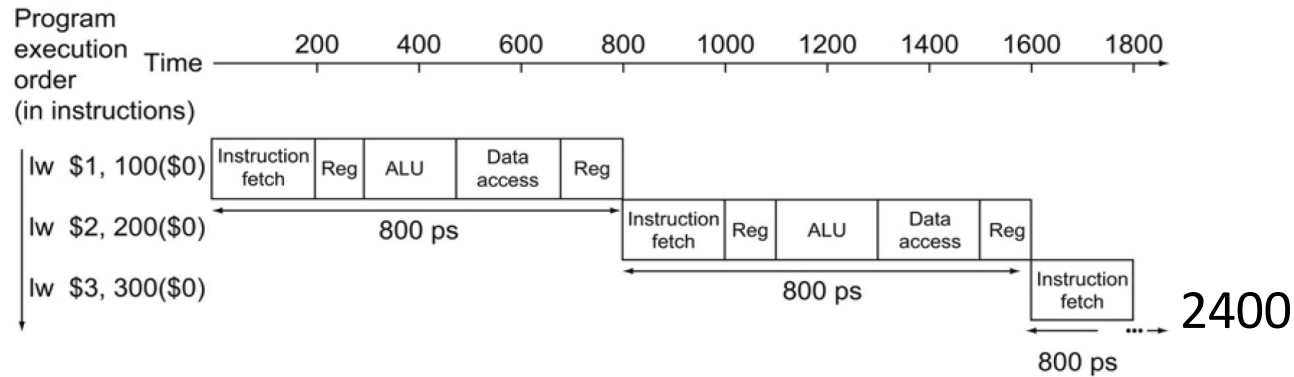
Speedup of 4 (*ideal = 5*)

*Ideal period = 160 ps*

# Performance Analysis

2400 ps (single cycle) versus 1400 ps (pipelined)  $\rightarrow$  speedup = 1.7

*Repeat the analysis for 1,000,003 instructions. Speedup?*



# Performance Analysis

*2400 ps (single cycle) versus 1400 ps (pipelined) → speedup = 1.7*

*Repeat the analysis for 1,000,000 instructions. Speedup?*

*Nonpipelined execution time =  $800 \text{ ps} * 1,000,000 + 2400 \text{ ps}$   
= 800,002,400 ps*

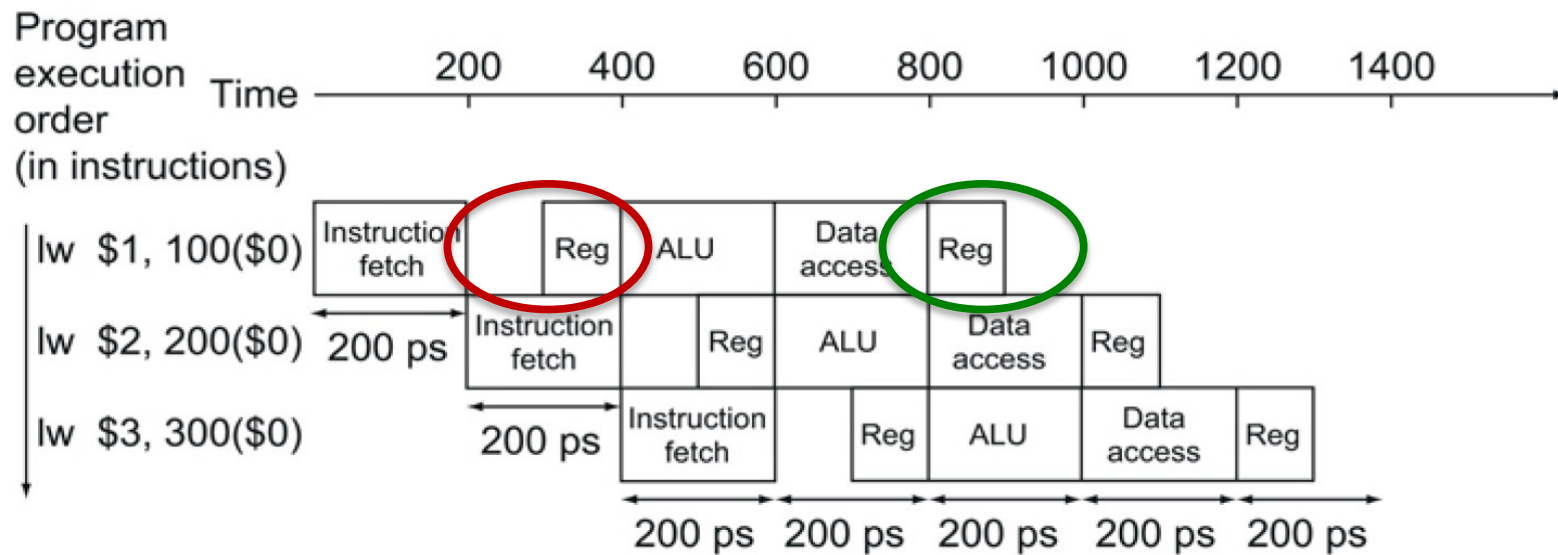
*Pipelined execution time =  $1,000,000 * 200 \text{ ps} + 1400 \text{ ps}$   
= 200,001,400 ps*

*Nonpipelined/pipelined = 4.00*

# Register File Read/Write

Register file is shared by instruction decode and writeback stages of the pipeline

- Need a policy to avoid race conditions due to simultaneous read & write
- **Read** during the second half of the clock cycle
- **Write** during the first half of the clock cycle



# Pipelining Idealism

The laundry system provides an ideal speedup of  $k (=4)$ , while realistic processor pipelines deliver sub-optimal speedup. *The disparity is due to three idealized assumptions (a.k.a. pipelined idealism) in the laundry system.*

1. *Uniform subcomputations*
2. *Identical computations*
3. *Independent computations*



# 1. Uniform Subcomputations

The computation to be pipelined can be evenly partitioned into  $k$  uniform-latency subcomputations. *If the latency (clocking period) of the original computation is  $T$ , then the clocking period of the pipelined computation is  $T/k$*

In our previous example, instruction fetch has a latency of 200 ps and register read has a latency of 100 ps

- This inefficiency is called **internal fragmentation** of the pipeline stages
- The new clock period is  $T_f$  divided by  $k$  where  $T_f$  is the worst-case stage delay

# Example

Find the internal fragmentation of the following pipeline

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

# Example

Find the internal fragmentation of the following pipeline

$$\rightarrow (200 * 5) - 800 = 200 \text{ ps}$$

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

# Uniform Subcomputations

No inefficiency due to partitioning of the computation

*No additional delays (e.g., due to buffering)*

→ *Not realistic for actual processor pipelines*

**Challenge: Balancing pipeline stages**

**Sources of imbalanced stages?**

→ E.g., Memory

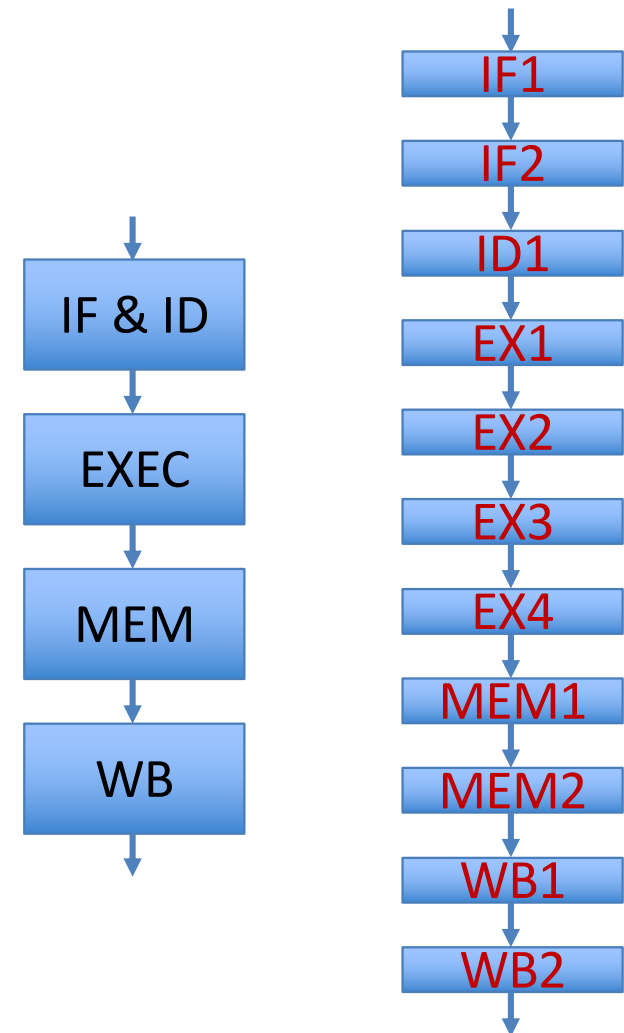
Two approaches: (1) Computation merging (2) partitioning

Is the MIPS pipeline balanced?

# Problem

Two pipeline designs of the 5-stage generic instruction sequence

→ Assume the total latency of five generic computations in the MIPS ISA is **280 ns**. The clock cycle times for a 4-stage design and the 11-stage design are **80 ns** and **30 ns**, respectively. What is the internal fragmentation for the two pipelines? What is the speedup relative to the nonpipelined implementation?



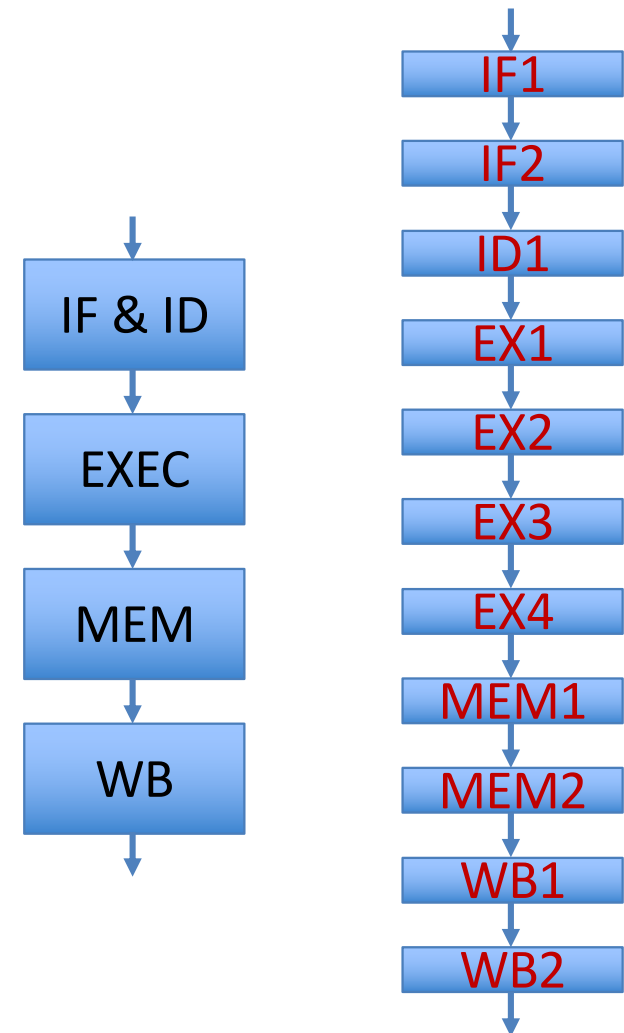
# Solution

Two pipeline designs of the 5-stage generic instruction sequence

→ Assume the total latency of five generic computations in the MIPS ISA is **280 ns**. The clock cycle times for a 4-stage design and the 11-stage design are **80 ns** and **30 ns**, respectively. What is the internal fragmentation for the two pipelines? What speedup do they offer relative to the nonpipelined implementation?

Speedup (4-stage) = 3.5X and (11-stage) = 9.3X

Frag (4-stage) = 40 ps and (11-stage) = 50 ps



## 2. Identical Subcomputations

*The same computation is repeatedly applied to different data over time*

→ *Wash, dry, fold, hang, but clothes(Alice) != clothes(Bob)*

In a general-purpose pipeline, not all stages are used by all the instructions

→ *In a load-store architecture, arithmetic instructions do not use the MEM stage*

→ *This inefficiency is called **external fragmentation** of the pipeline stages*

# Identical Subcomputations

*External fragmentation leads to idle stages for some instruction types*

*Pipeline fill and drain cycles are also a form of external fragmentation since not all stages are busy during those cycles*

***Challenge: Unifying instruction types***



# 3. Independent Subcomputations

All computations concurrently residing in the pipeline are independent (i.e., no control or data dependences)

*A pipeline with independent computations operates in “streaming mode” or at full speed. If a later computation depends on an earlier computation that has not executed yet, the later computation waits in the pipeline (officially called pipeline stall)*

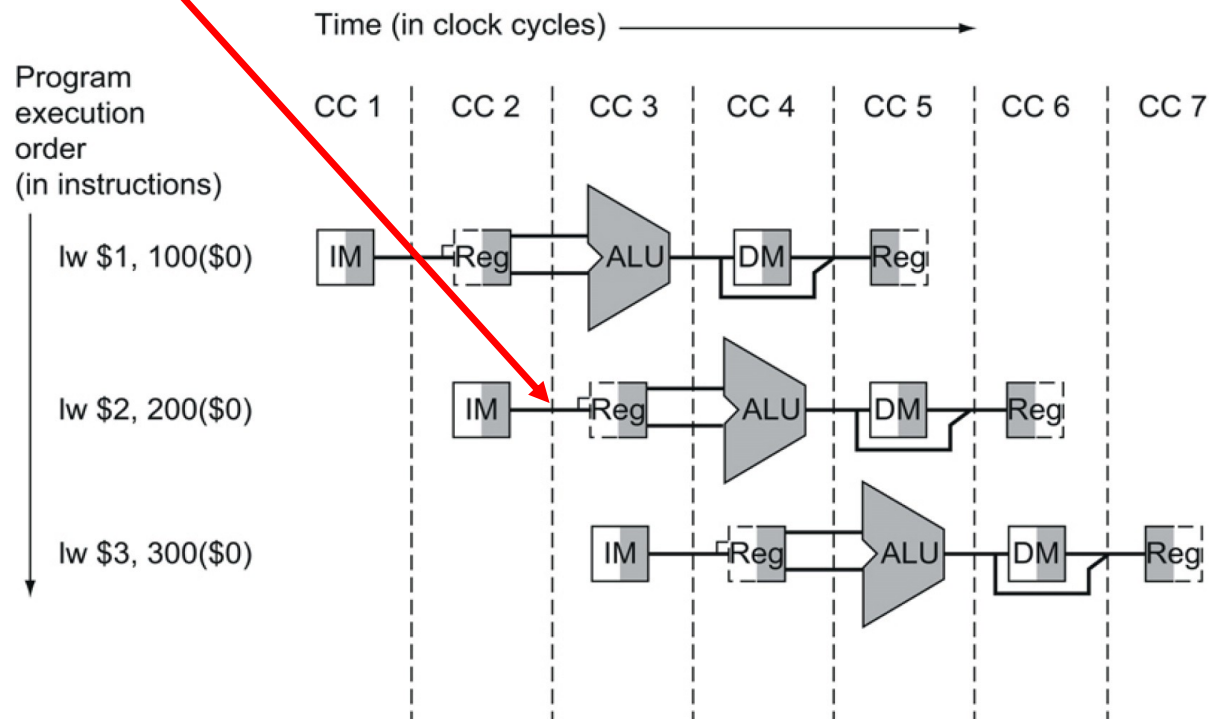
**Challenge:** *Minimizing pipelines stalls*

# Multi-Clock-Cycle Diagrams

Each instruction (Y-axis) has its own datapath. Timeline on X-axis

*Unless we save instructions, data, and control bits in pipeline registers, we lose the information about the previous instruction*

*Graph of instructions & data flow over time*



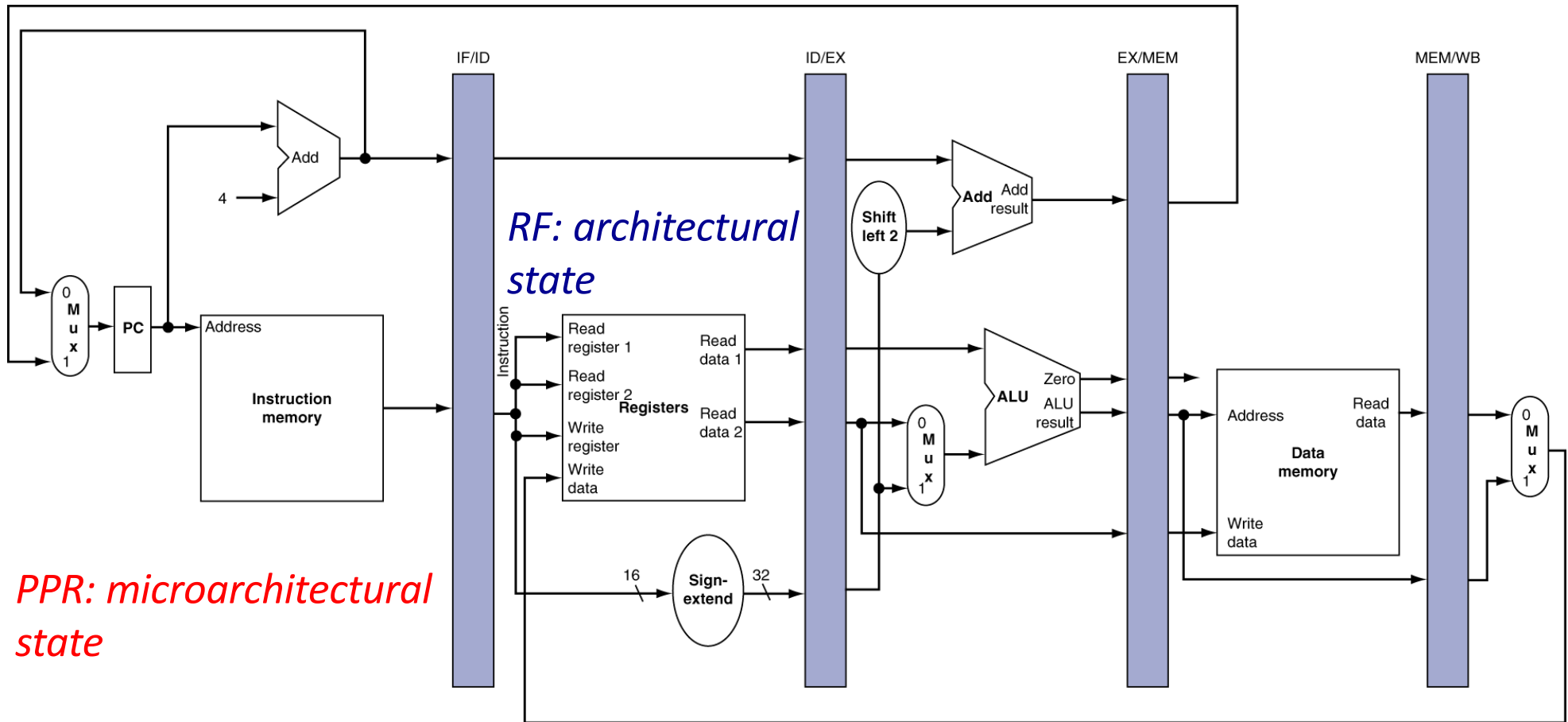
# Pipeline Registers (PPR)

IF/ID

ID/EX

EX/MEM

MEM/WB ????



# Width of Pipeline Regs

What is the width of each of the following pipeline registers?

IF/ID

ID/EX

EX/MEM

MEM/WB

Note: We will expand these registers as we build a more realistic pipeline

# Width of Pipeline Regs

What is the width of each of the following pipeline registers?

IF/ID: 64

ID/EX: 128

EX/MEM: 97

MEM/WB: 64

Note: We will expand these registers as we build a more realistic pipeline

# Single-Clock-Cycle Diagrams

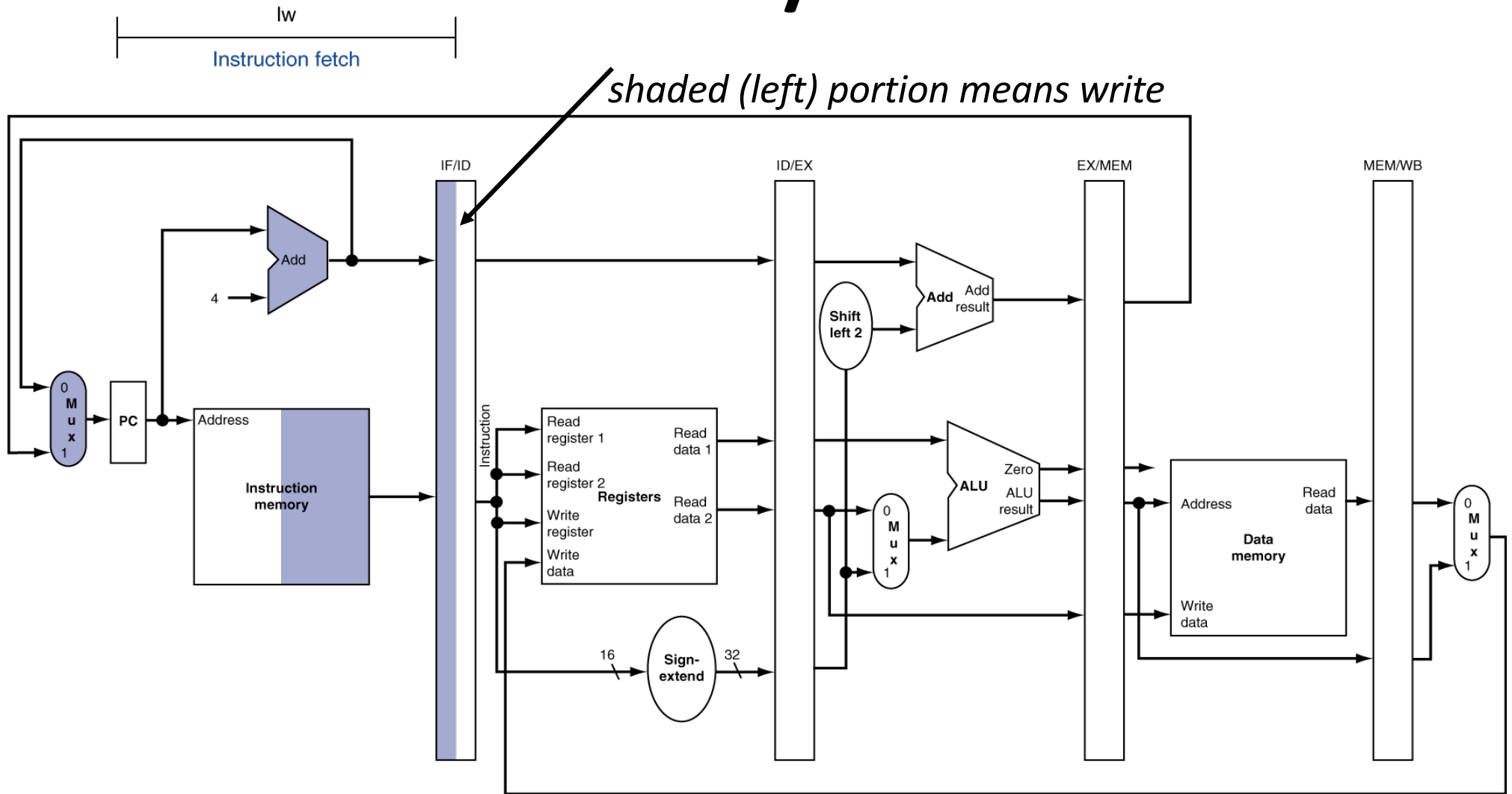
Shows the state of the entire datapath during a single cycle

*Vertical slice through a set of multiple clock cycle diagrams*

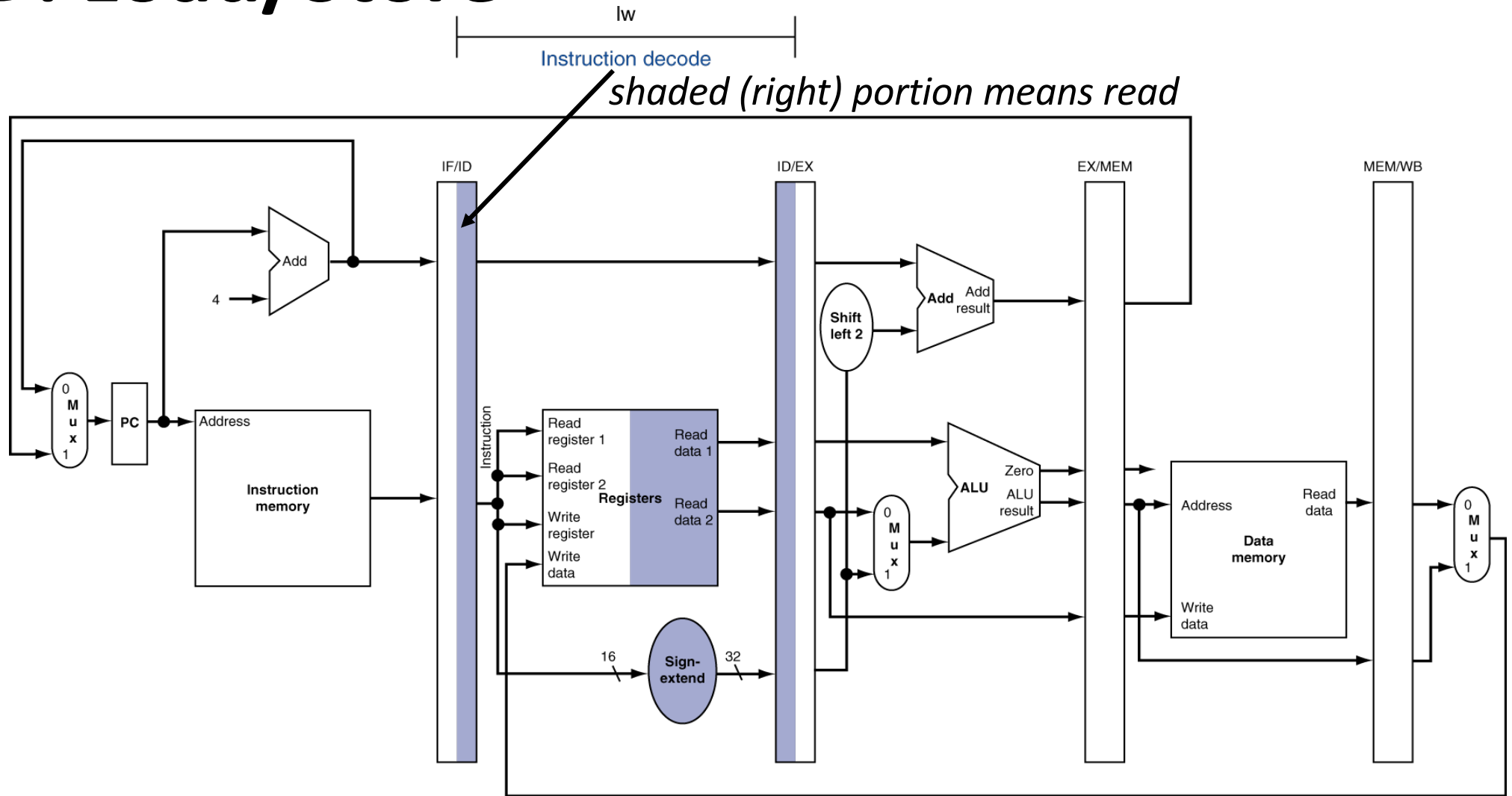
Which instructions occupy what resources in a specific cycle?

**Let's look at single-clock cycle diagrams for load and store instructions**

# IF: Load/Store



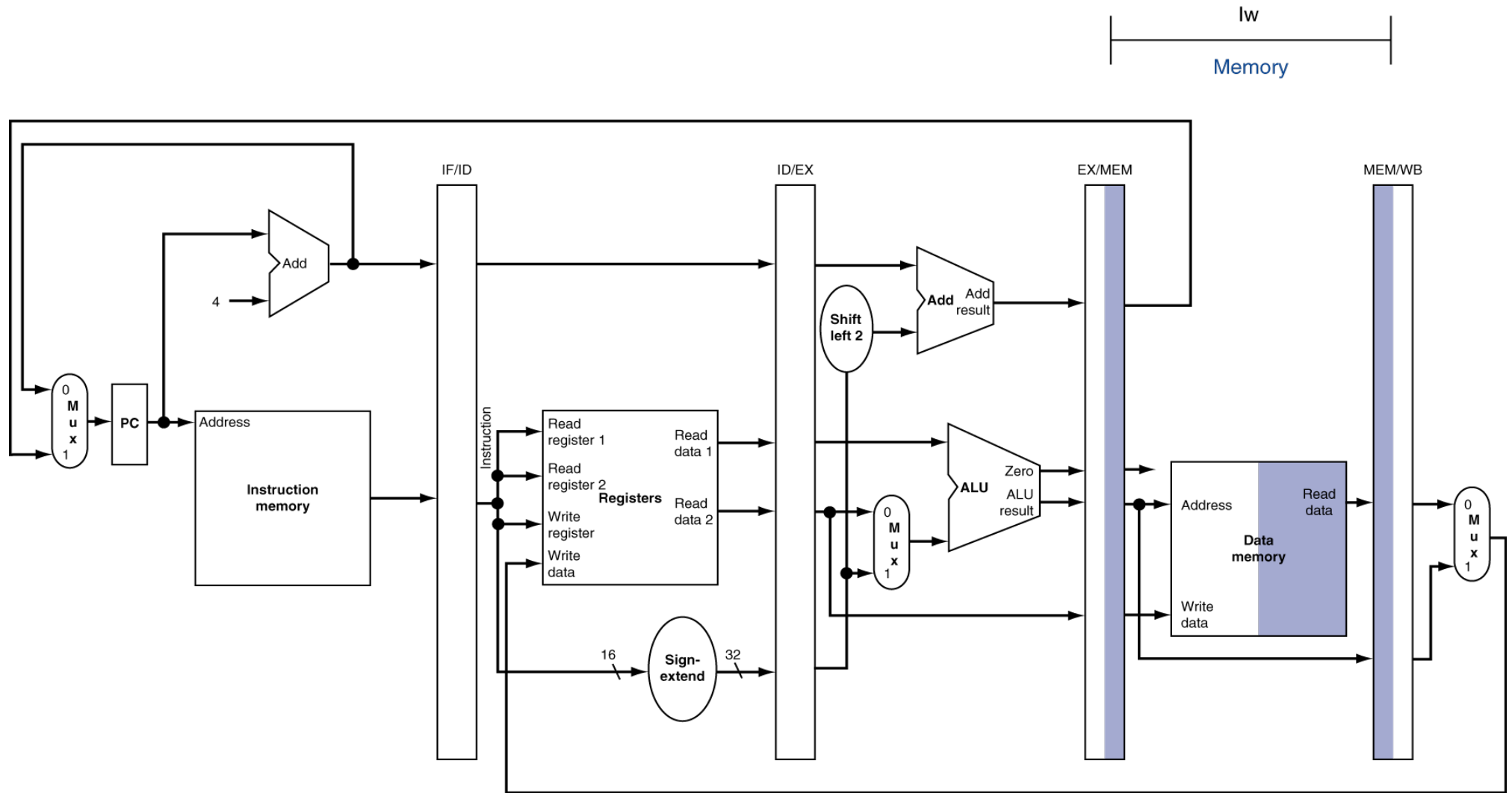
# ID: Load/Store





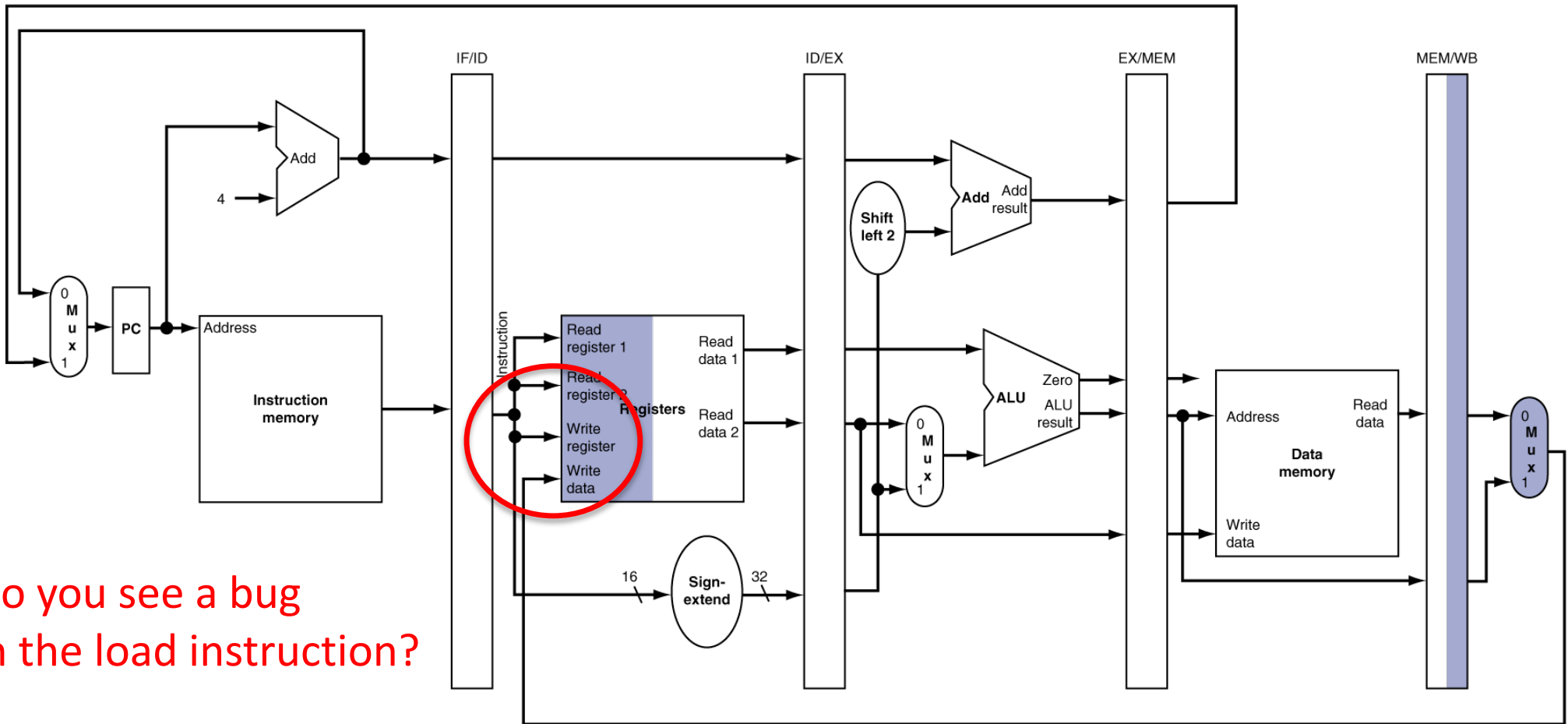


# MEM: Load



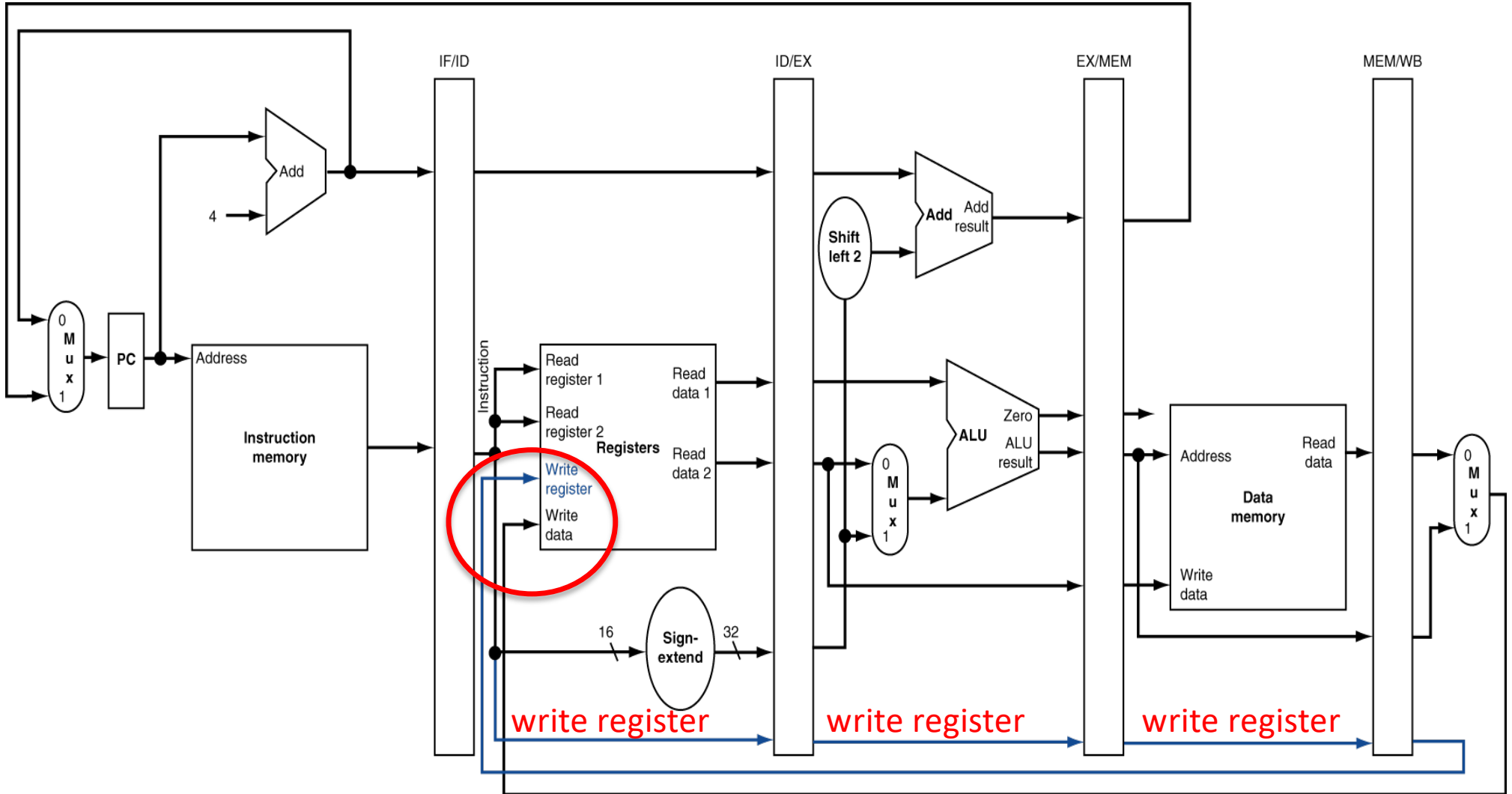
# WB: Load

lw  
Write back

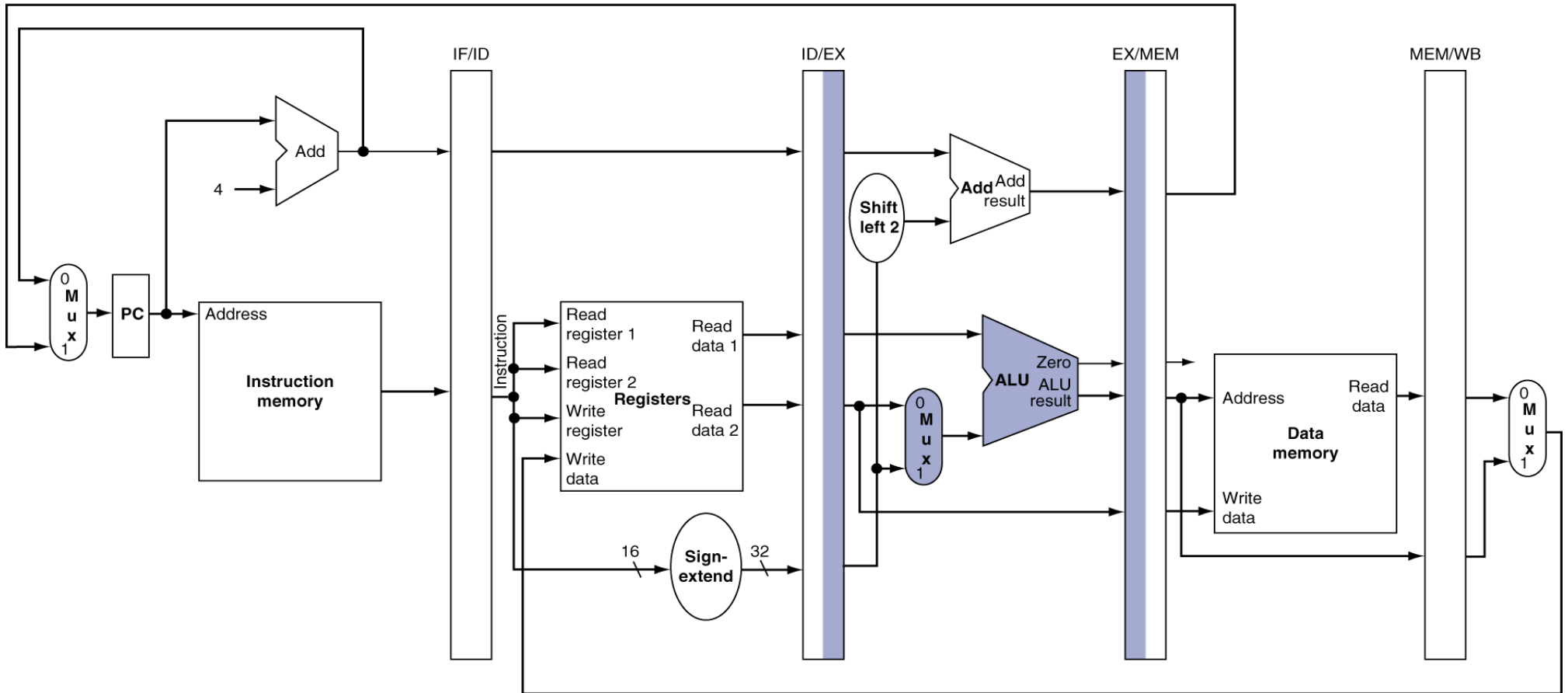


Do you see a bug  
in the load instruction?

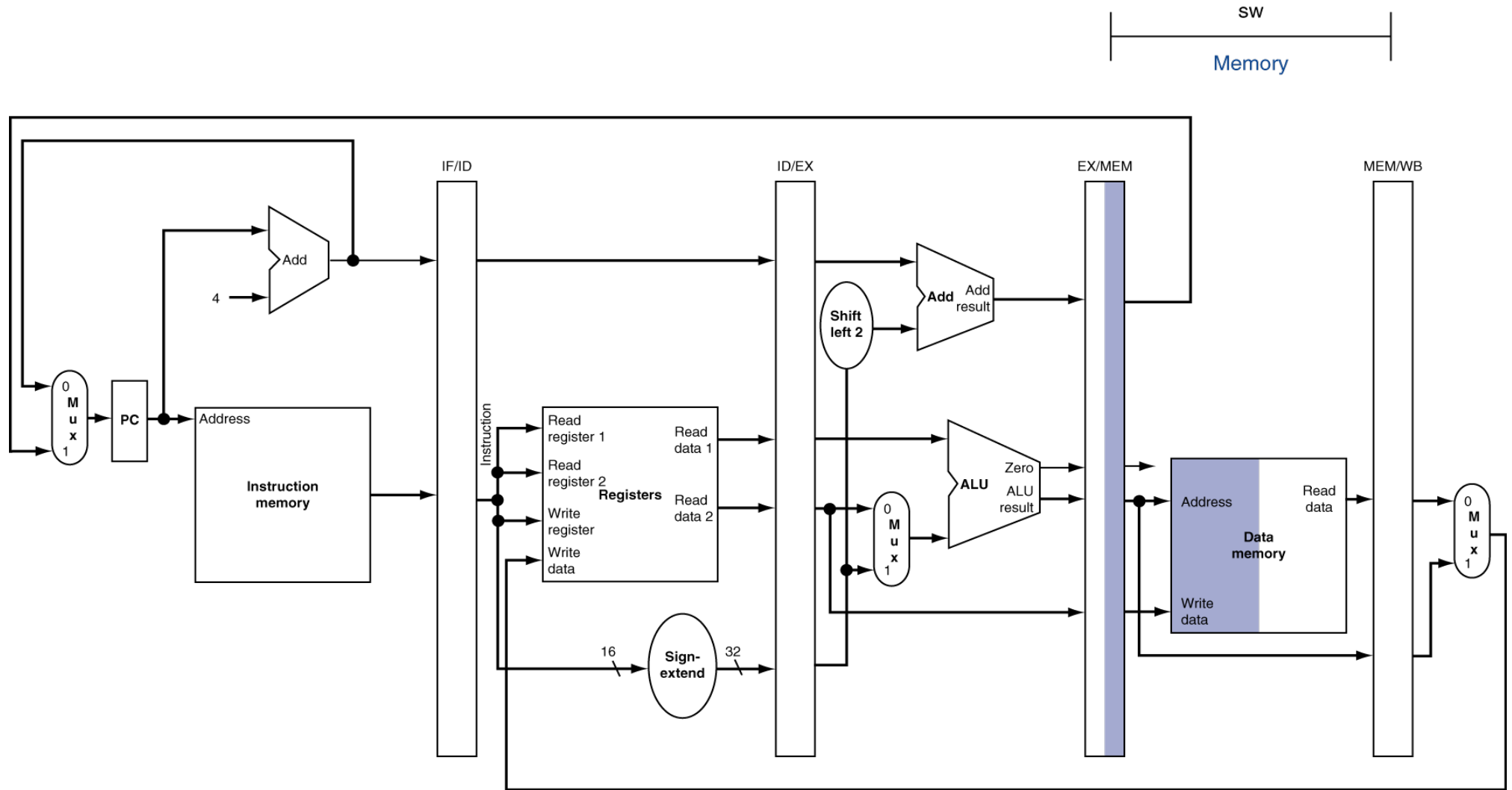
# WB: Load (Corrected)



# EX: Store

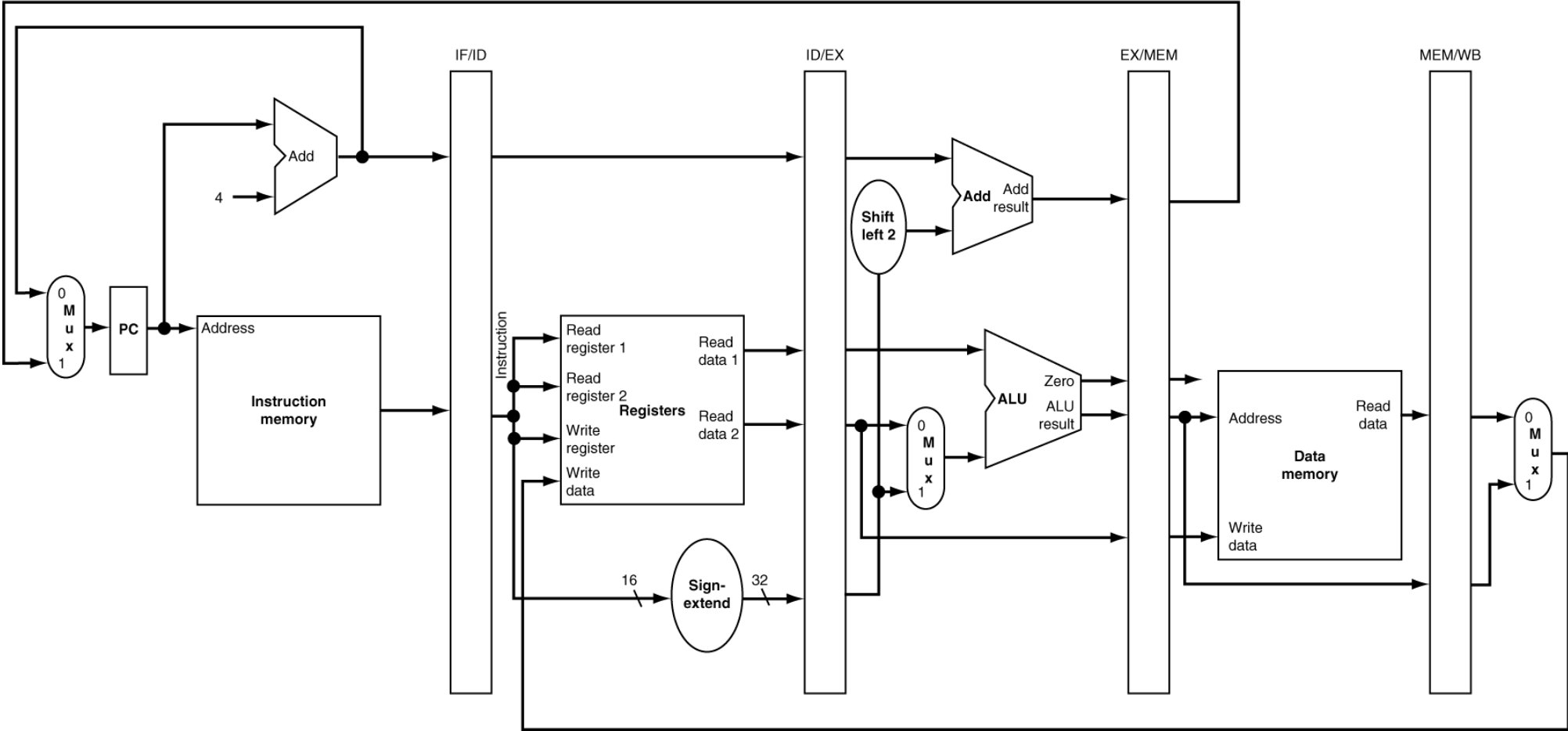


# MEM: Store

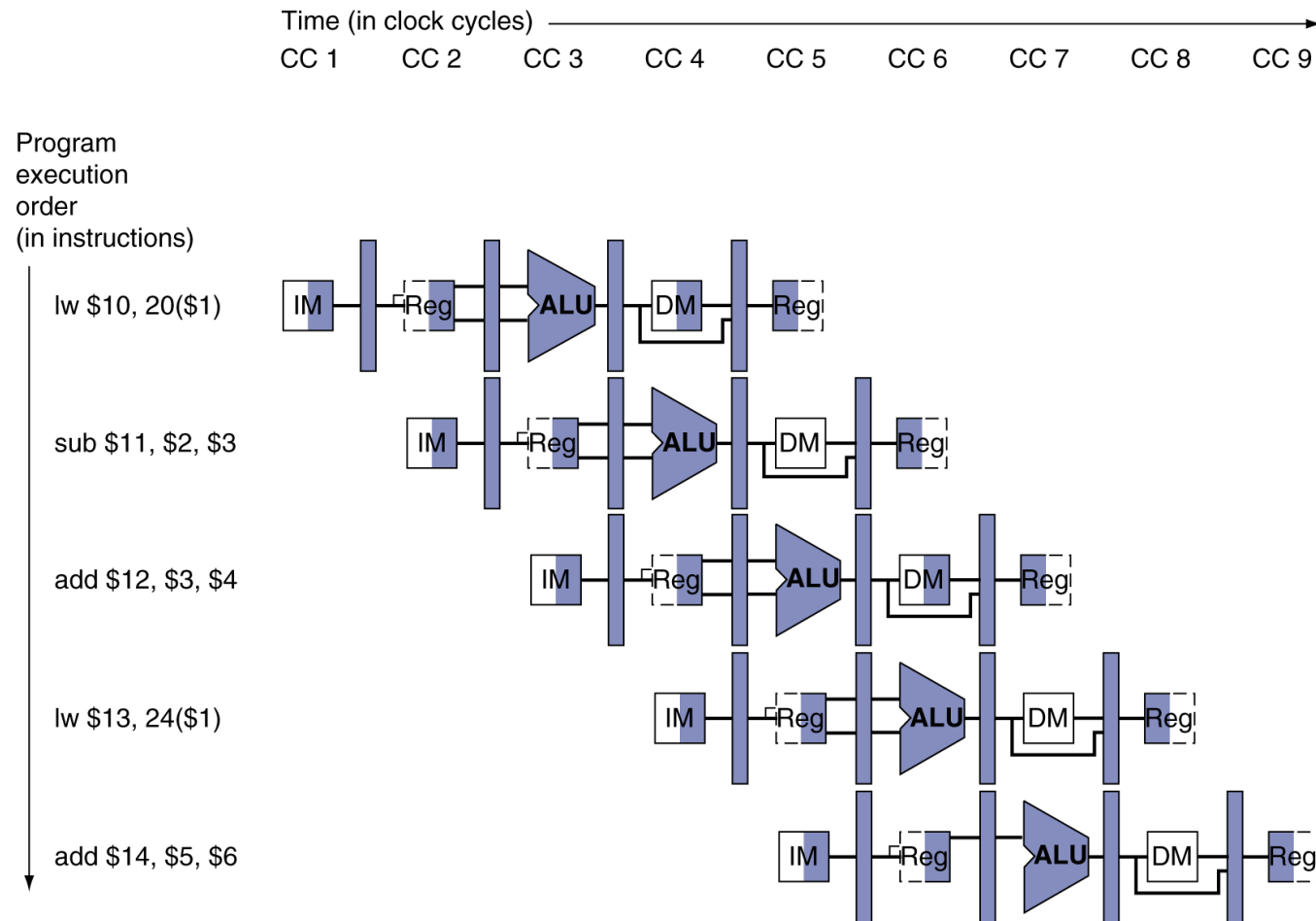


# Writeback: Store

SW  
Write-back



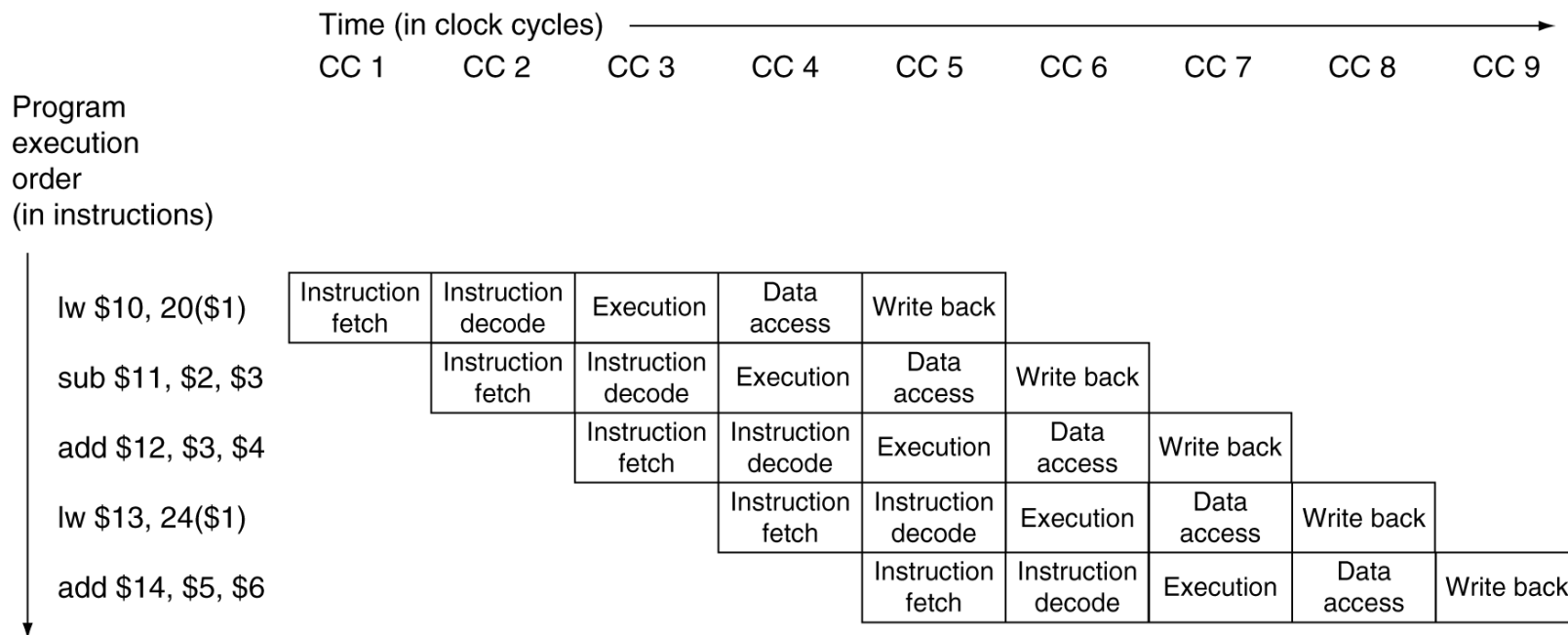
# Showing Resource Usage





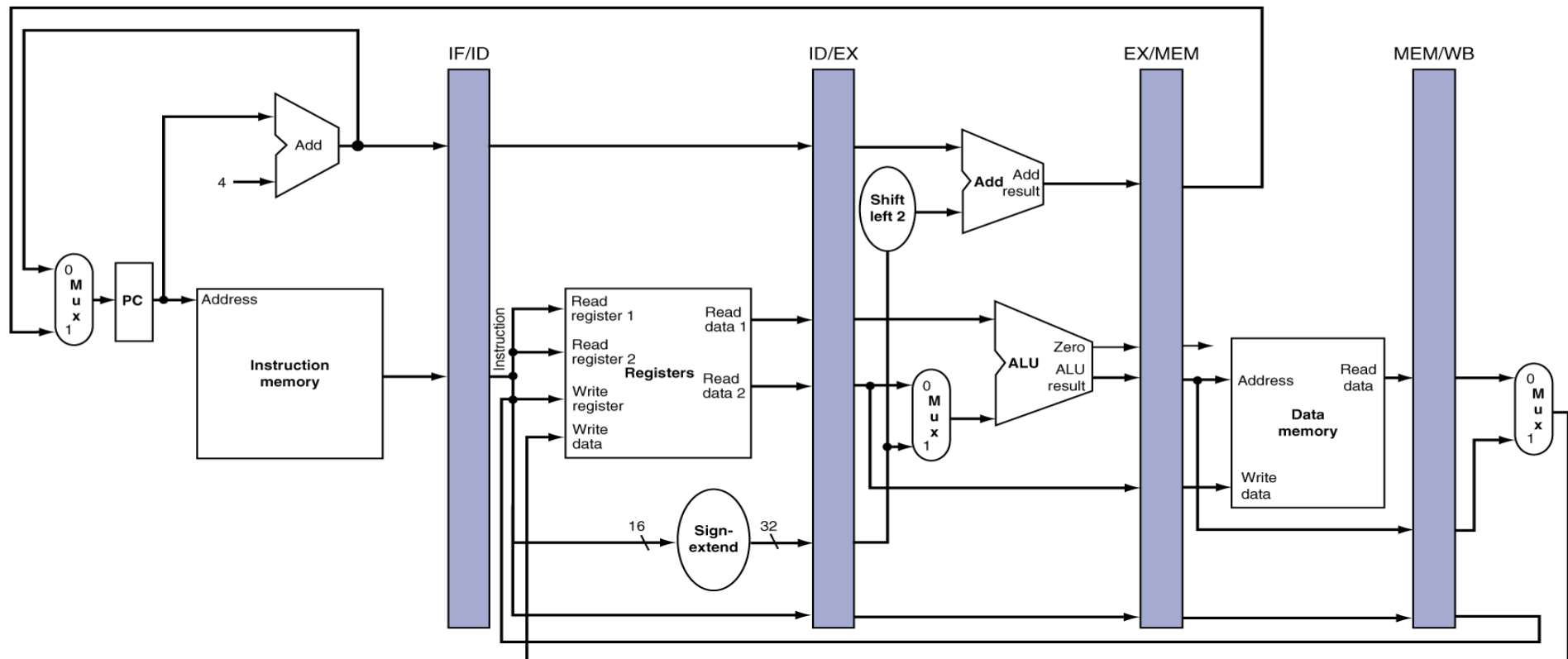
# Drawing Pipelines with Less Detail

We will see a lot of these in the in-order to out-of-order transition



# Single-Cycle w/t multiple insts

add \$14, \$5, \$6	lw \$13, 24(\$1)	add \$12, \$3, \$4	sub \$11, \$2, \$3	lw \$10, 20(\$1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



# Pipelined Control Unit

**Strategy:** Set the control lines during each stage

Each **major** component is active during a **single** stage

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

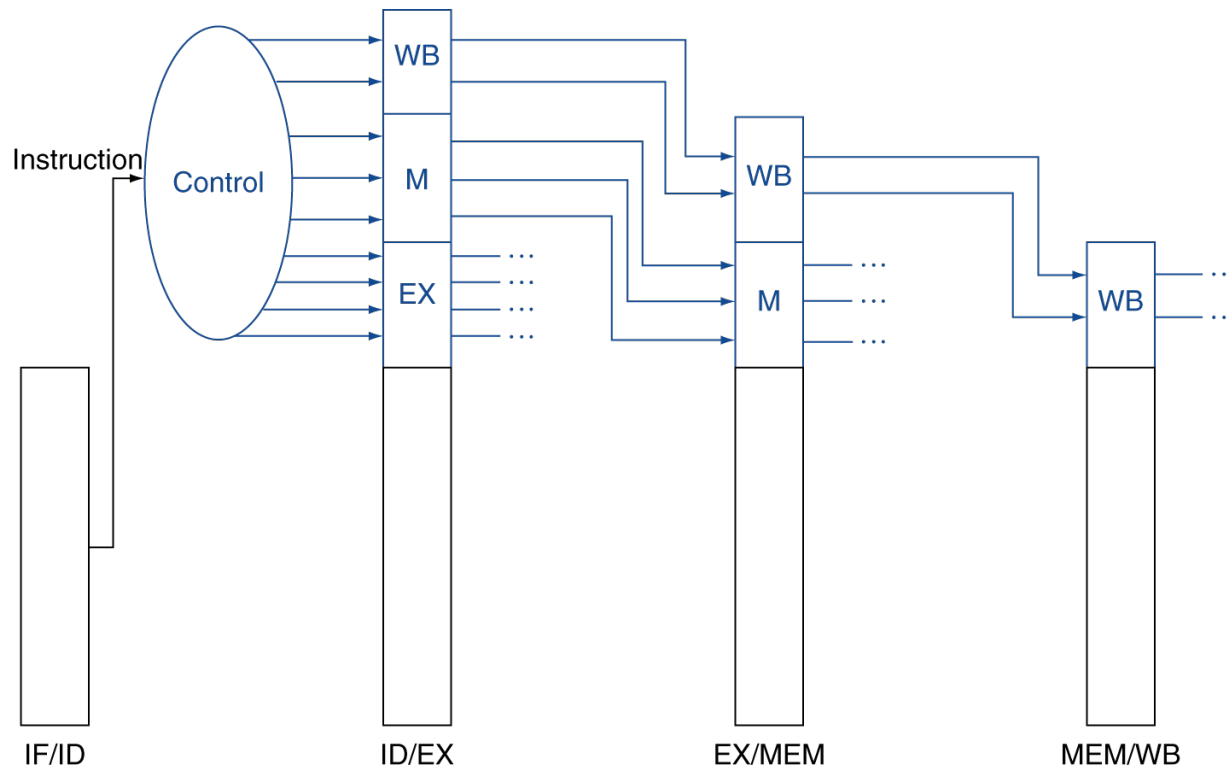
Do we need a write signal for PC and pipeline regs?

Where should we store the nine control signals for each instruction?

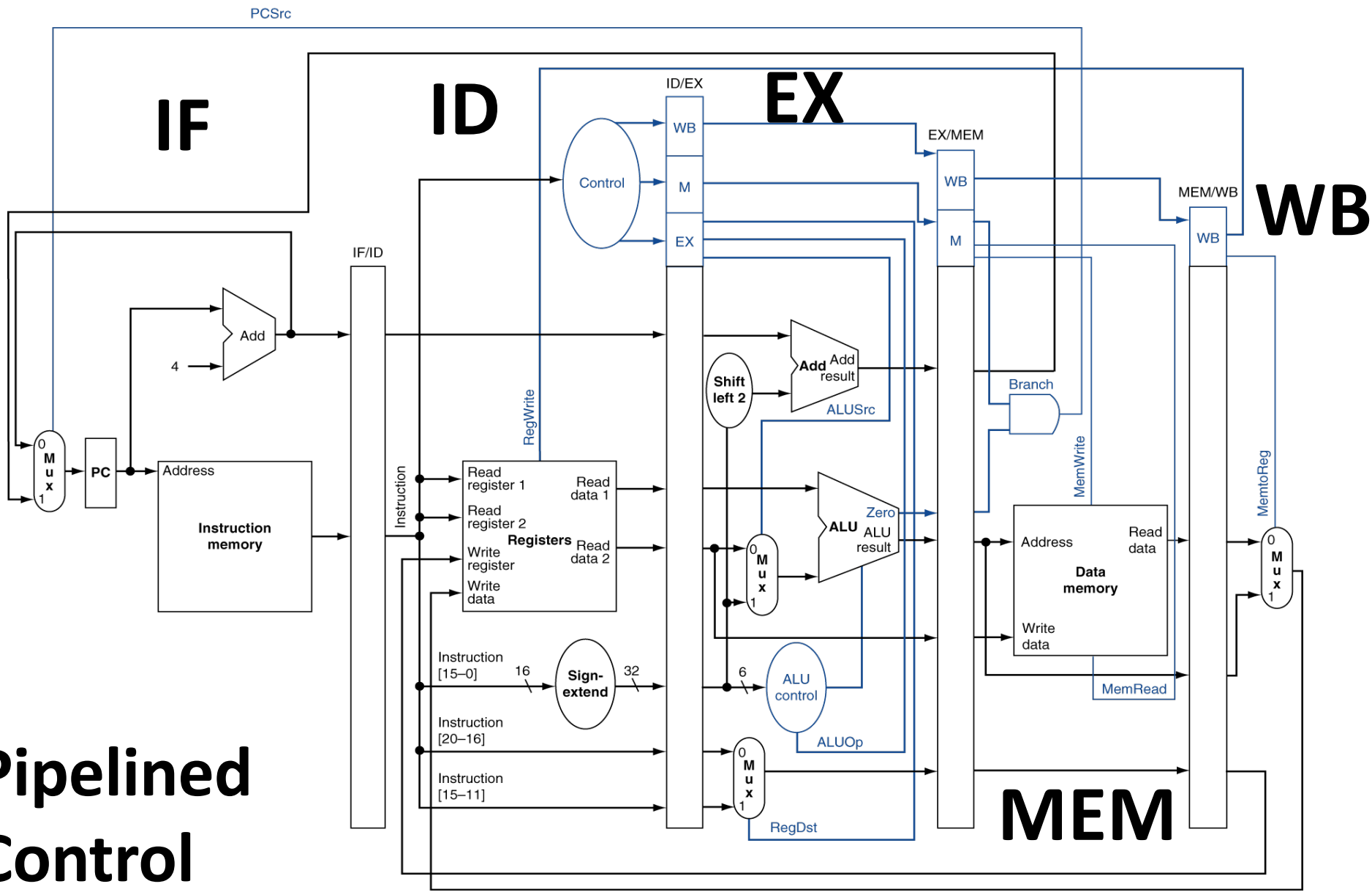
# Pipelined Control

Control signals are generated during the decode stage

They move down the pipeline like any other data



# Pipelined Control



# Practice Problem

What is the width of each of the following pipeline registers?

IF/ID

ID/EX

EX/MEM

MEM/WB

# Hazards

There are events in pipelining called *hazards* when the next instruction cannot execute in the following clock cycle

*Real life hazards*



*Coping mechanisms: Jump,  
Walk around, Forewarned,  
Stop, Find another way*

*Some hurt performance  
Others clever ways to get  
around the hazard*

# Types of Pipeline Hazards

*Structural hazards (e.g., shared instruction/data memory)*

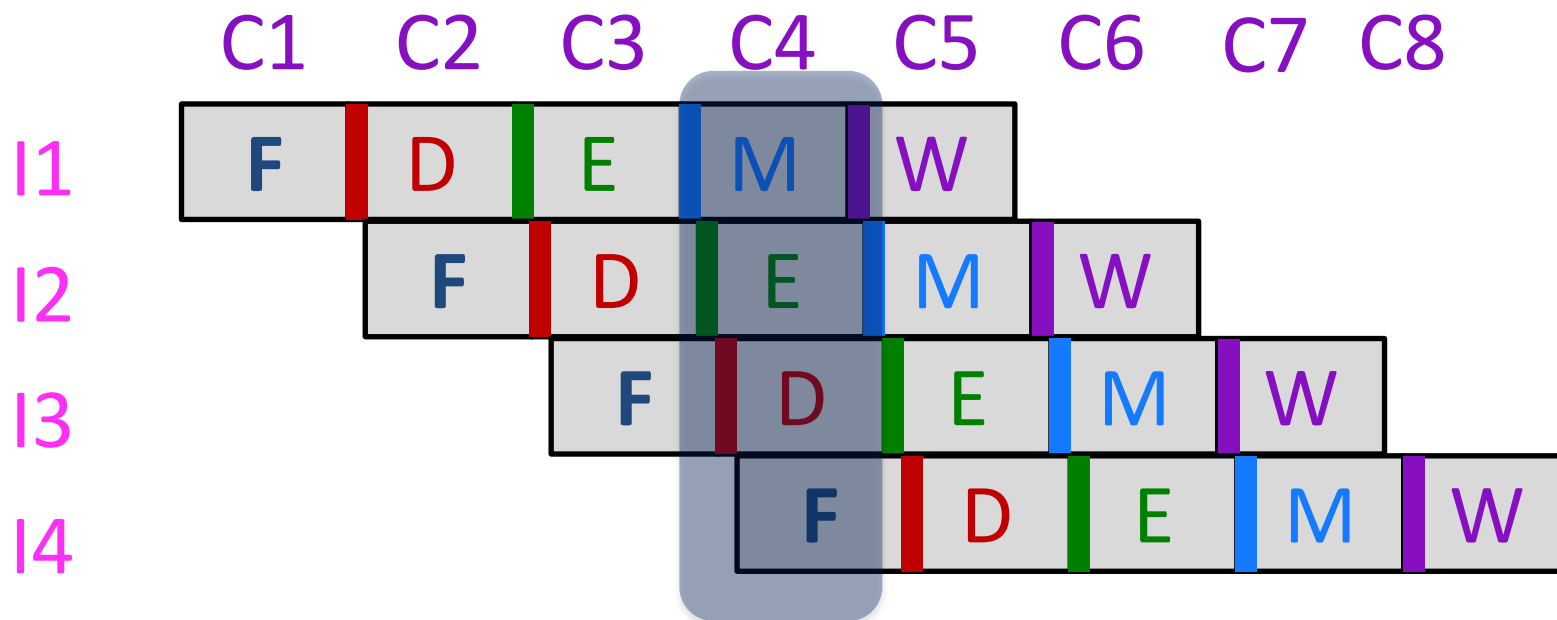
*Data hazards (dependences)*

*Control hazards (branches)*

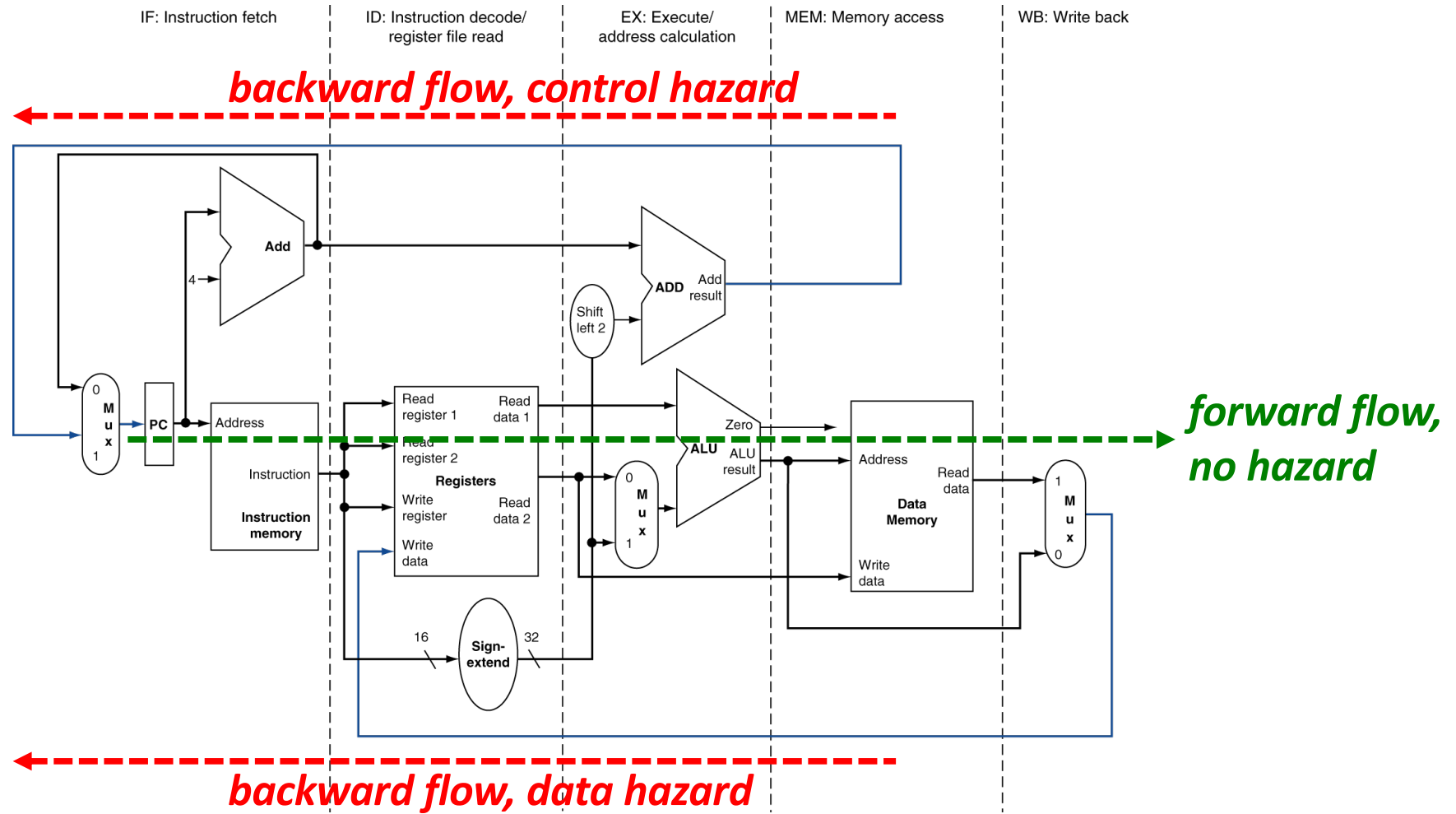


# Structural Hazards

Hardware cannot support the combination of instructions in the same clock cycle, *e.g.*, *unified instruction/data memory*



# Data/Control Hazards



# Data/Control Hazards

Instructions and data generally flow from **left** to **right**

**Right-to-left** flow affect future instructions and leads to hazards

- Writeback stage places the result into the register file (potential for data hazard)
- Selection of next PC, choice of PC + 4 or branch target address (MEM stage)

# Pipeline Data Hazards

**Data hazard:** *When an instruction cannot execute in the proper clock cycle because data is not available*

```
add    $s0    $t0    $t1
sub    $t2    $s0    $t3
```

**Question:** How many cycles the **sub** instruction needs to wait for the 5-stage MIPS pipeline?

add      \$s0      \$t0      \$t1  
 sub      \$t2      \$s0      \$t3

*No dependences*

Cycle #	add	sub
1	IF	●
2	ID	IF
3	EX	ID
4	MEM	EX
5	WB	MEM
6		WB

*Read/Write RF in different cycles*

Cycle #	add	sub
1	IF	●
2	ID	✗
3	EX	✗
4	MEM	✗
5	WB	IF
6		ID
7		EX
8		MEM
9		WB

*Read/Write RF in the same cycle*

Cycle #	add	sub
1	IF	●
2	ID	✗
3	EX	✗
4	MEM	IF
5	WB	ID
6		EX
7		MEM
8		WB

● Not Fetched  
 ✗ Stall

# Question

Who can rescue us from the pipeline stalls due to data hazards?

→ Compiler

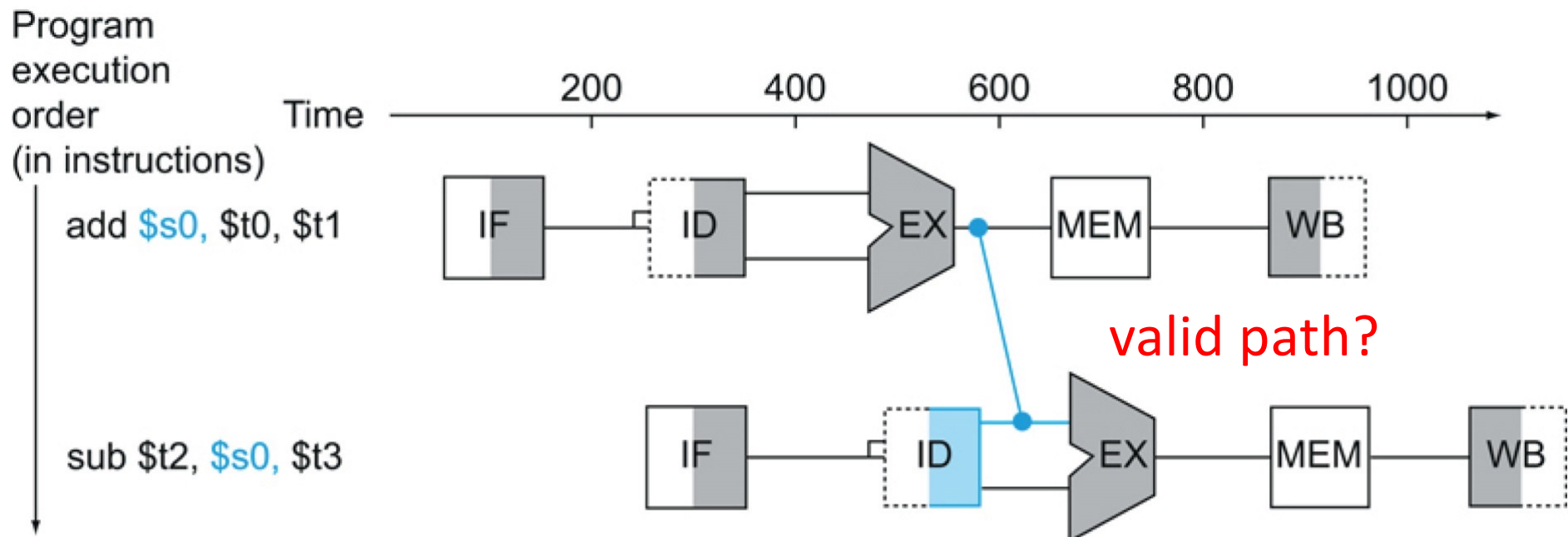
→ Hardware

*Compiler can reorder instructions or insert nops but dependences are too frequent. We need a different approach*

Question: Does the **sub** instruction need to wait until the **add** instruction writes the new value of **\$s0** to the register file?

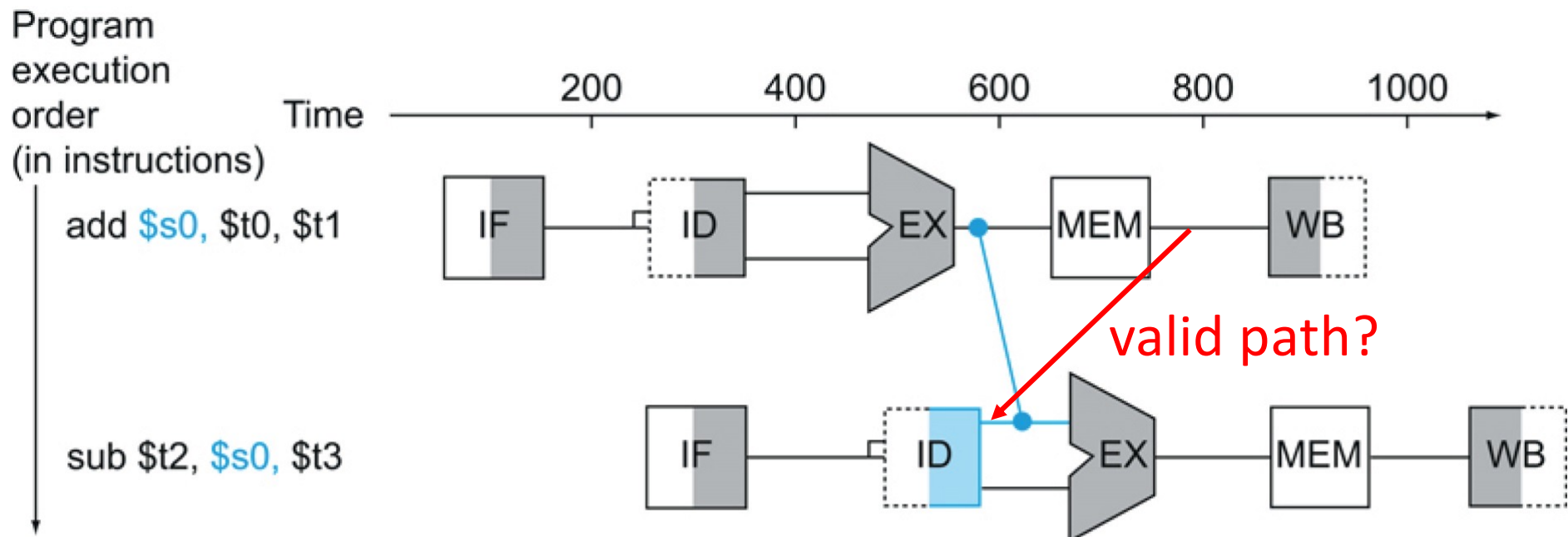
# Forwarding or Bypassing

**Forwarding:** Adding extra hardware to retrieve the missing operand early instead of waiting for the writeback (*resolving data hazards*)



# Forwarding or Bypassing

**Forwarding:** Adding extra hardware to retrieve the missing operand early instead of waiting for the writeback (*resolving data hazards*)

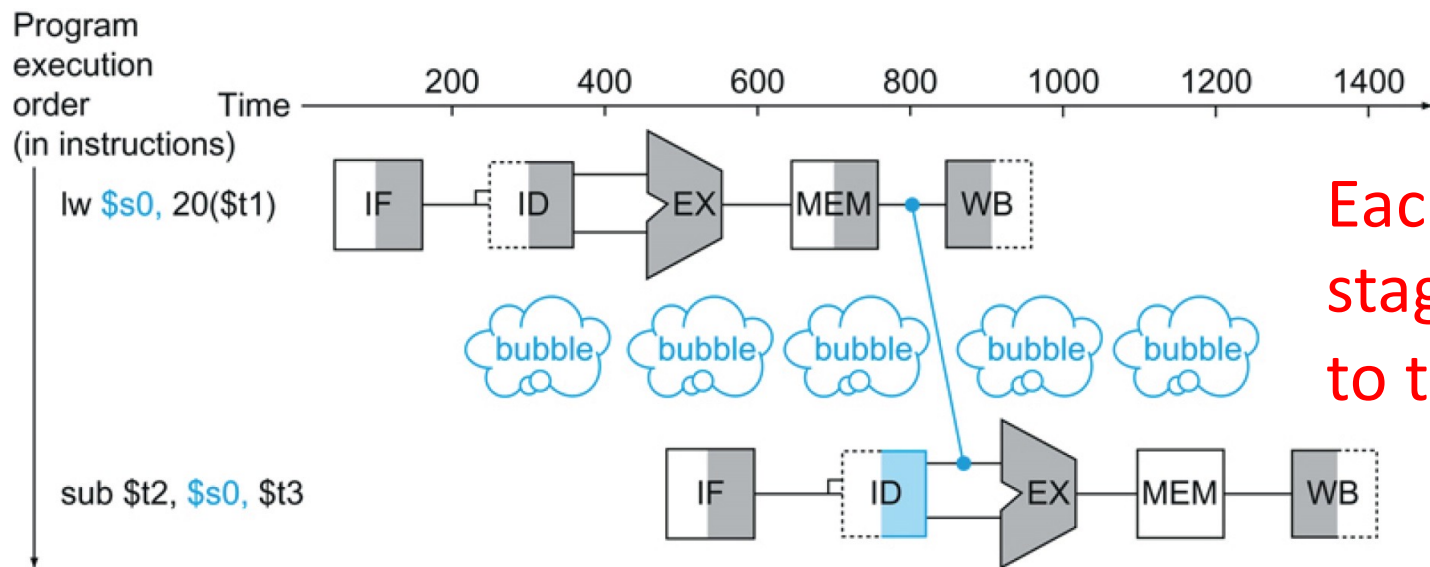




# Load-Use Data Hazard

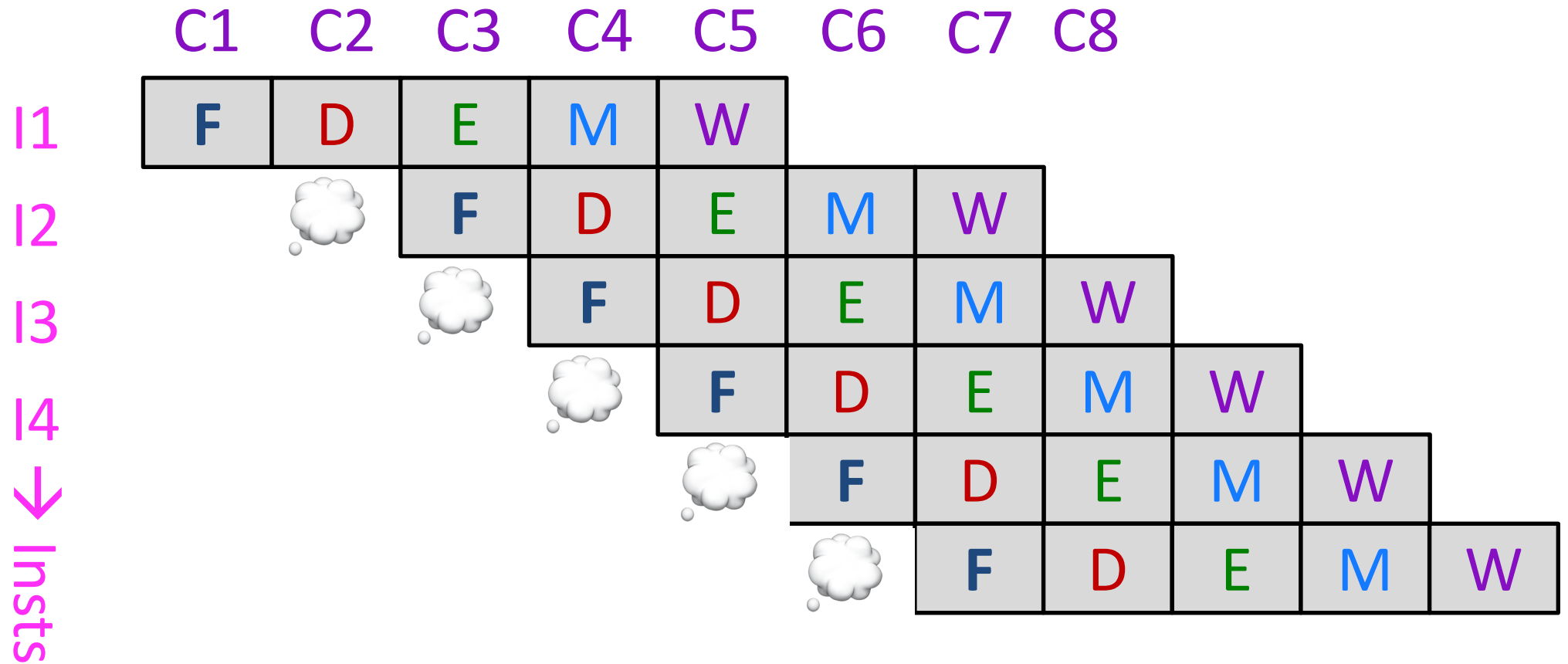
Forwarding cannot prevent all stalls (e.g., load followed by use)

**Pipeline stall/bubble:** *A stall introduced to resolve a hazard*

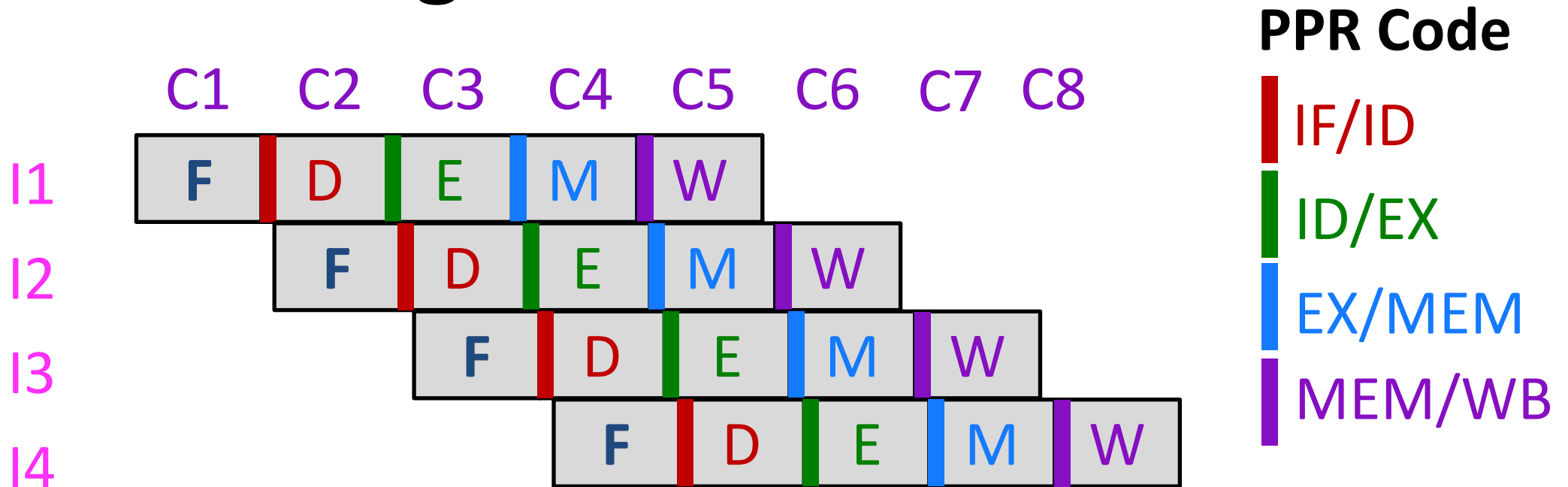


Each cycle, a stage is idle due to the bubble

# Visualizing Bubbles



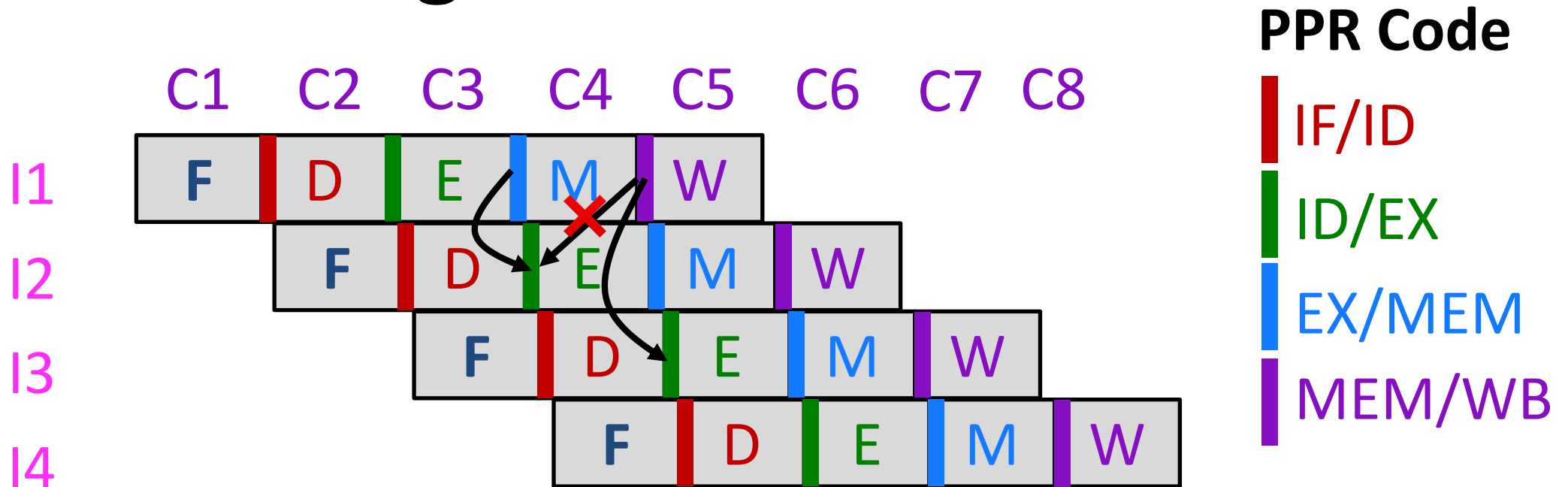
# Forwarding Exercise 1



Insts ↓

- Is forwarding from I1(EX/MEM) to I2(ID/EX) valid?
- Is forwarding from I1(MEM/WB) to I3(ID/EX) valid?
- Is forwarding from I1(MEM/WB) to I2(ID/EX) valid?

# Forwarding Exercise 1



Is forwarding from I1(EX/MEM) to I2(ID/EX) valid?

Is forwarding from I1(MEM/WB) to I3(ID/EX) valid?

Is forwarding from I1(MEM/WB) to I2(ID/EX) valid?

# Forwarding Exercise 2

Find all the **hazards** and **reorder** the instructions to avoid stalls.

```
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)
```

# Dependences, Data Hazard, Stall, and Forwarding

Data dependence is a property of the program

→ *Instruction  $j$  reads what instruction  $i$  writes*

Data dependence leads to hazard (depending on the microarchitecture)

→ *We call it read-after-write (RAW) hazard*

Stall is a hardware-based RAW-hazard-avoiding mechanism

→ *Hurts performance because pipeline does not operate at full speed*

Forwarding is another hardware-based RAW-hazard-avoiding mechanism

→ *Except load-use hazards, pipeline operates at full speed*

Compiler can help

→ *Insert nops (still need a hardware mechanism to stall just in case)*

→ *Reorder instructions (difficult because RAW is too common)*

# Next Steps

1. *Forwarding logic and control*
2. *Hazard detection logic (for load-use hazards)*
3. *Inserting pipeline bubbles/stalls/nops*

# Forwarding Implementation

1. *Detecting the need to forward (forwarding conditions)*
2. *Designing a new forwarding unit for the processor*
3. *Introducing forwarding paths*
4. *Finalize changes to information in PPRs, and new control signals*



# Dependences and Hazards

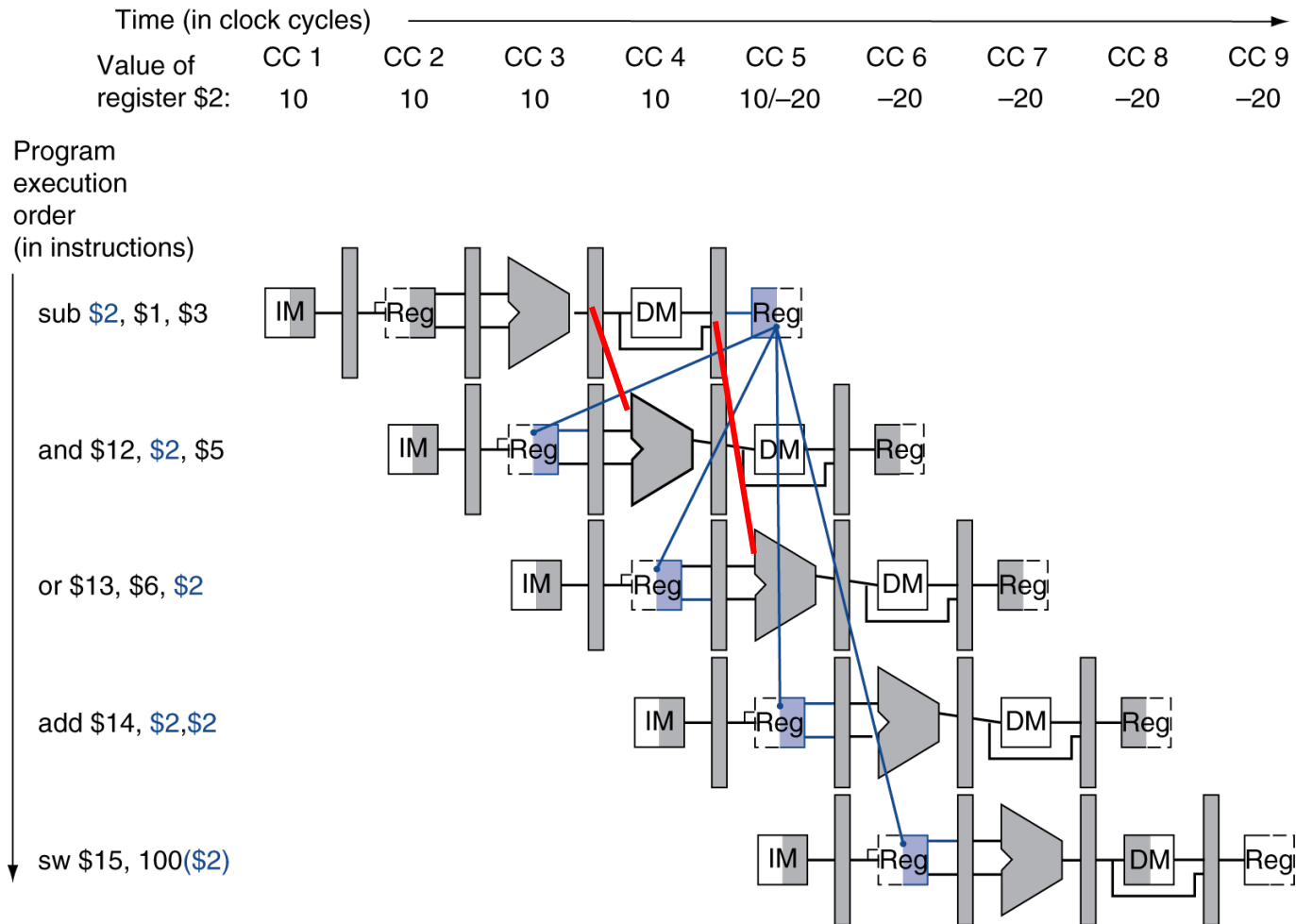
sub	\$2	\$1	\$3
and	\$12	\$2	\$5
or	\$13	\$6	\$2
add	\$14	\$2	\$2
sw	\$15	100(\$2)	

Register \$2 before **sub** instruction = 10

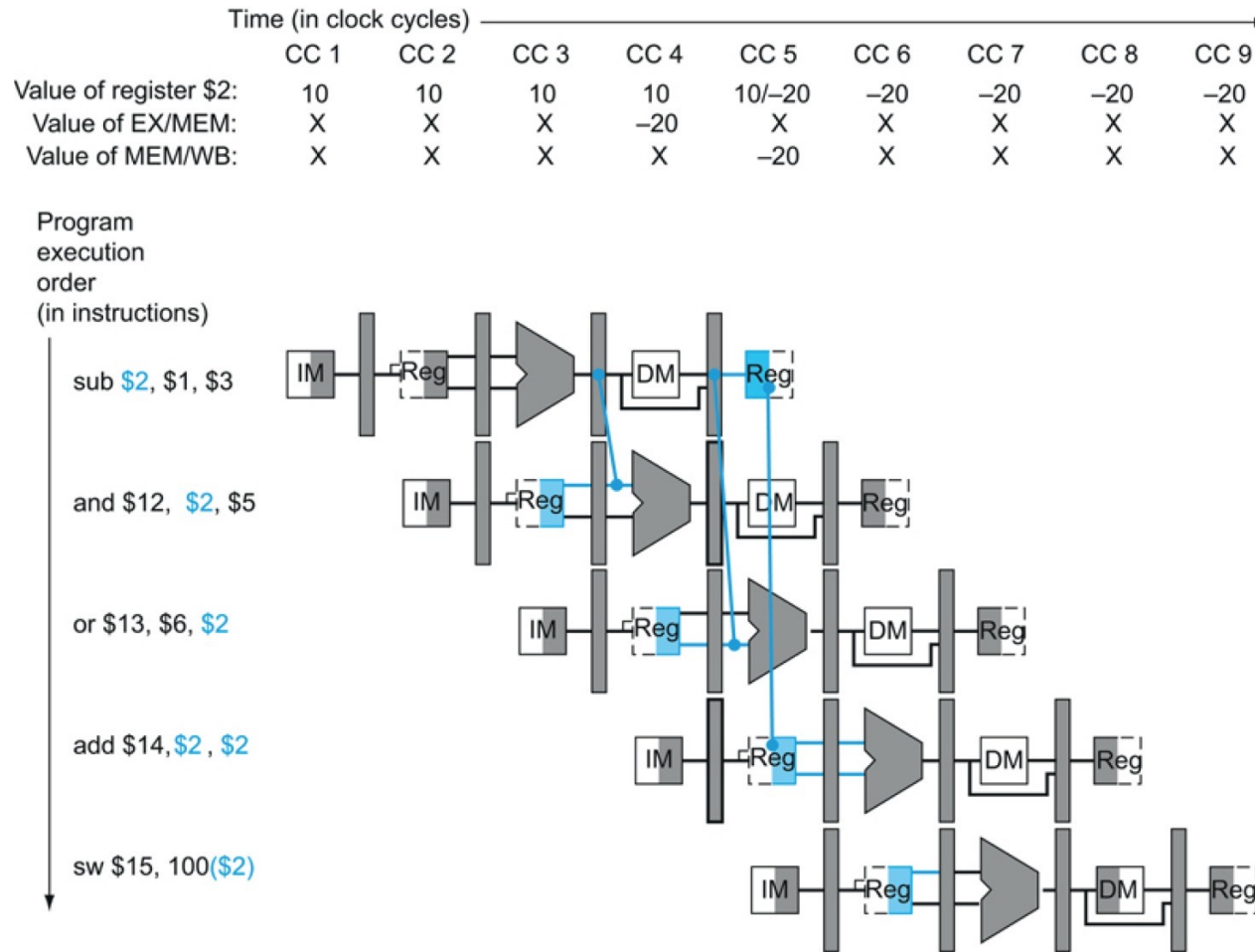
After sub = -20

All subsequent instructions must see -20 for \$2

# Dependences and Hazards



# Dependences and Hazards



# Detecting the Need to Forward

**The previous figure illustrates the goal of forwarding:** *When an instruction needs a register in the EX stage that an older instruction currently in MEM or WB stage intends to write, forward the result to the input of the ALU*

# Detecting the Need to Forward

*We need a notation to check for dependences between instructions about to execute and older ones in MEM/WB stages*

**PPR.Field** → A specific **Field** in the pipeline register **PPR**

**EX/MEM.Rd** → In the EX/MEM pipeline register, the 5-bit field for the destination register

**ID/EX.Rs** → In the ID/EX pipeline register, the 5-bit field for the first source register

# Four Hazard Types

Type 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs	} Forward from EX/MEM PPR
Type 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt	
Type 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs	} Forward from MEM/WB PPR
Type 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt	

Only forward when *the older instruction intends to write*

→ EX/MEM.RegWrite is 1

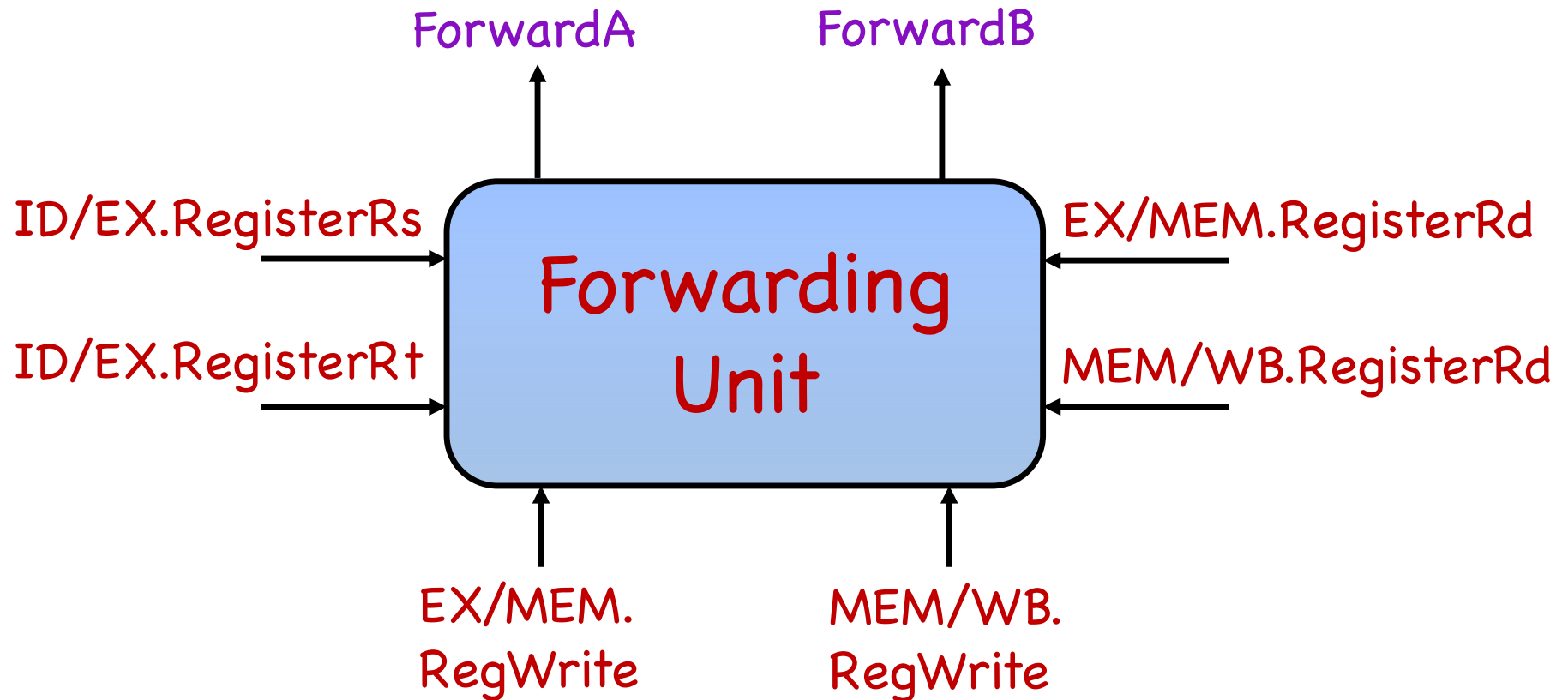
→ MEM/WB.RegWrite is 1

# Exercise

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

Classify the hazard types in the above sequence?

# Forwarding Unit





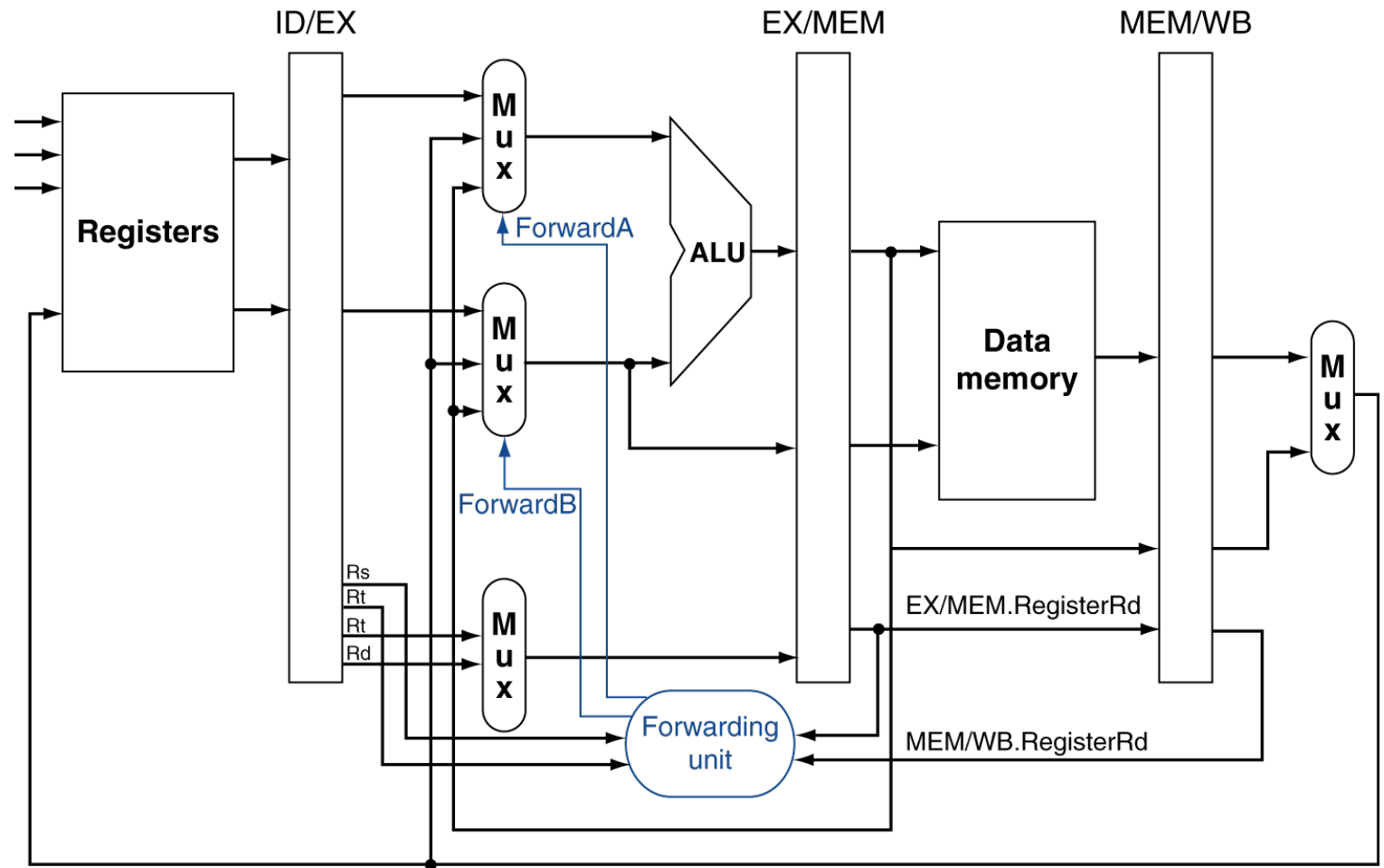
# Question

Where should we place the forwarding unit?

How does forwarding impact the logic at the inputs of the ALU in the EX stage?

# Forwarding Paths

Expanded multiplexors to add forwarding paths



Mux control	Source	Explanation
ForwardA = 00	ID/EX	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.

# Forwarding Conditions

## EX hazard:

```
if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
    ForwardA = 10
```

```
if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
    ForwardB = 10
```

# Forwarding Conditions

## MEM hazard:

```
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
    ForwardA = 01
```

```
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
    ForwardB = 01
```

# Double Data Hazard

```
add    $1    $1    $2
add    $1    $1    $3
add    $1    $1    $4
```

Hazards between the results of an instruction in the WB stage, the result of an instruction in the MEM stage, and the source operand of an instruction in the ALU stage

→ *Which stage should forward the result to last add?*

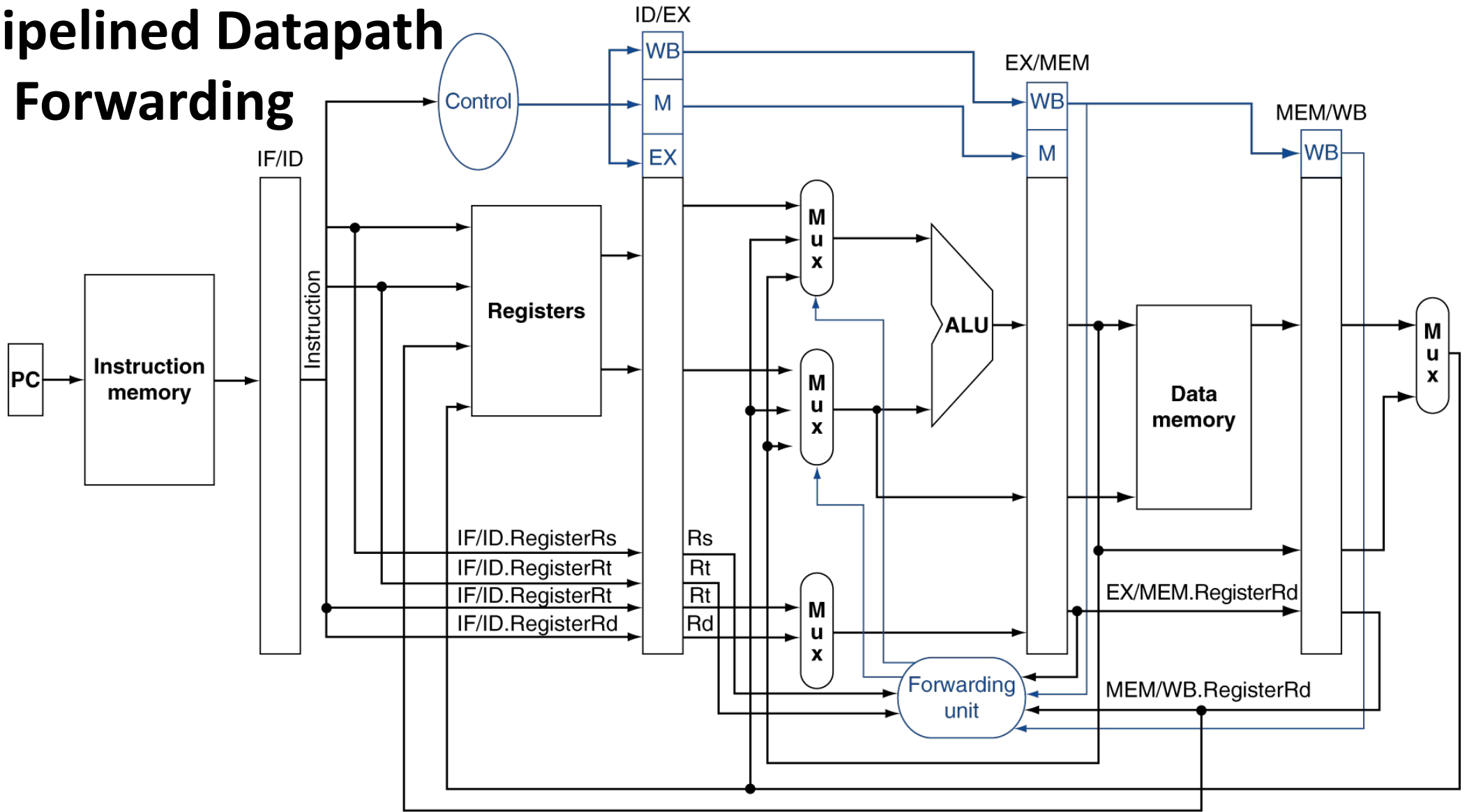
# Forwarding Conditions

## MEM hazard:

```
if (MEM/WB.RegWrite  
and not (EX/MEM.RegisterWrite  
         and (EX/MEM.RegisterRd = ID/EX.RegisterRs)))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
    ForwardA = 01
```

```
if (MEM/WB.RegWrite  
and not (EX/MEM.RegisterWrite  
         and (EX/MEM.RegisterRd = ID/EX.RegisterRt)))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
    ForwardB = 01
```

# Pipelined Datapath + Forwarding



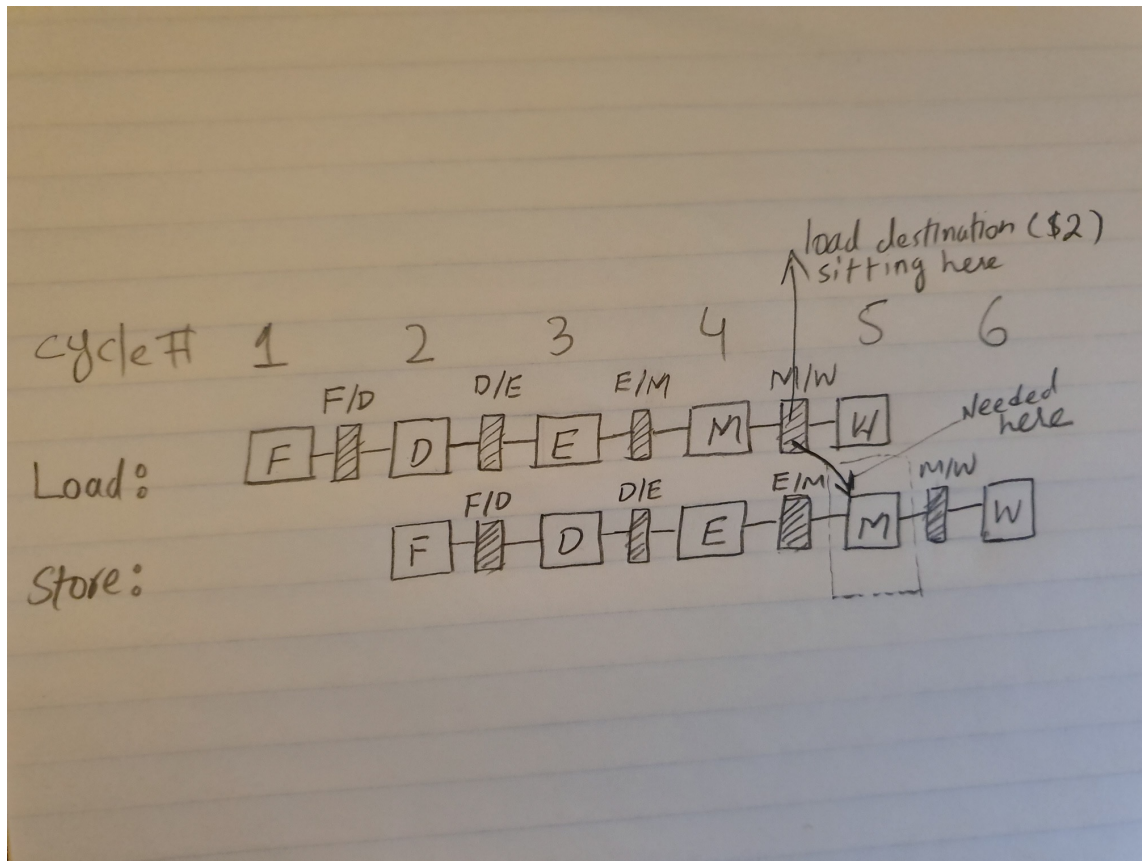


# Questions to Ponder

What about forwarding in other pipeline stages?

*What type of forwarding do you need to accelerate memory copies?*

*What is the critical path impact of forwarding?*



# ISA Impacts

Fixed width (MIPS) vs. variable width (x86)

→ Fetch (1<sup>st</sup> stage) + Decode (2<sup>nd</sup> stage) is difficult with a more complex ISA

Complex addressing modes such as register-memory increase external fragmentation (makes unification of formats harder)

MIPS ISA has a few symmetric instruction formats. Asymmetric formats leads to fragmentation as above

# ISA Impacts

Load/Store architecture makes *dependence detection* easy because there are no memory references except load/store insts

Operands in MIPS are **aligned** in memory (One memory transfer)

Clean and symmetric formats provides a clean ***dynamic-static interface*** (DSI), i.e., what to do statically (compile time) vs. dynamically (run-time)

MIPS writes one result to the register file and always in the last stage (**forwarding is easy**)