CSE261: Computer Architecture

12. Processor (4): Pipelining #1

Seongil Wi

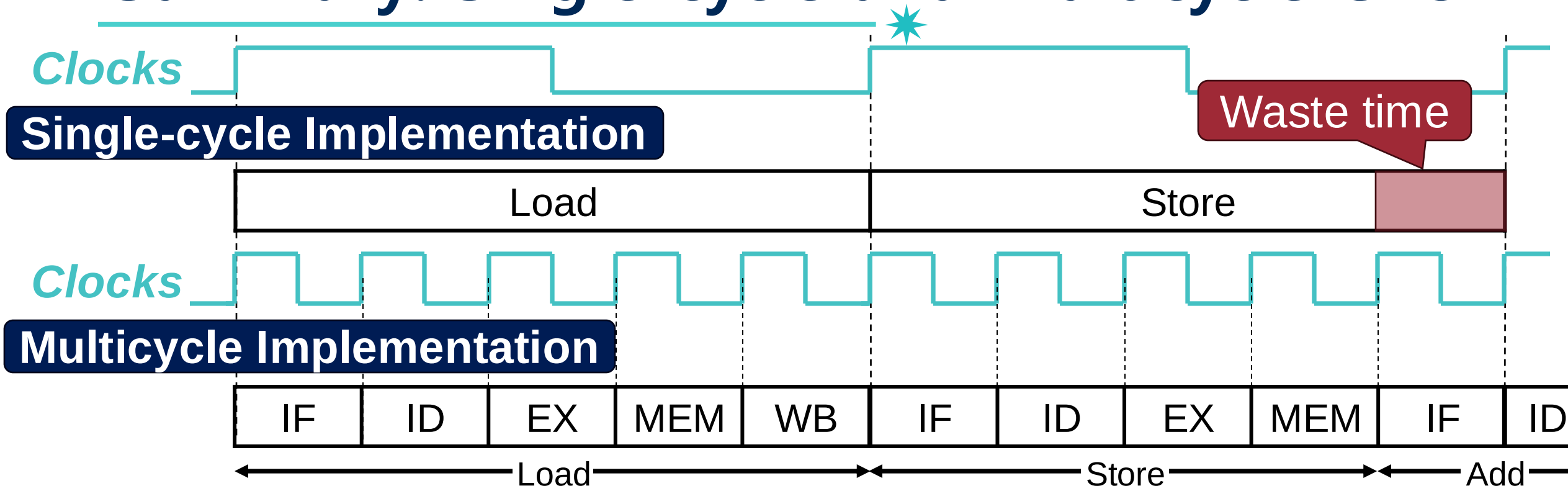
Department of Computer Science and Engineering

HW2 will be Released Soon!

Implementing single-cycle datapath and control

- Due: Nov. 21, 11:49PM

Summary: Single-cycle and Multicycle CPU



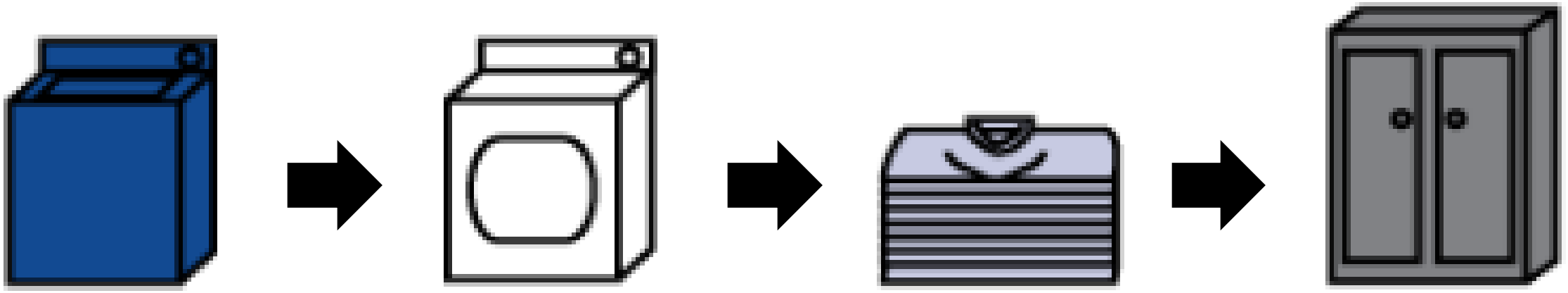
Is there a way to increase the speed further?

Pipelining Concept

Pipelining

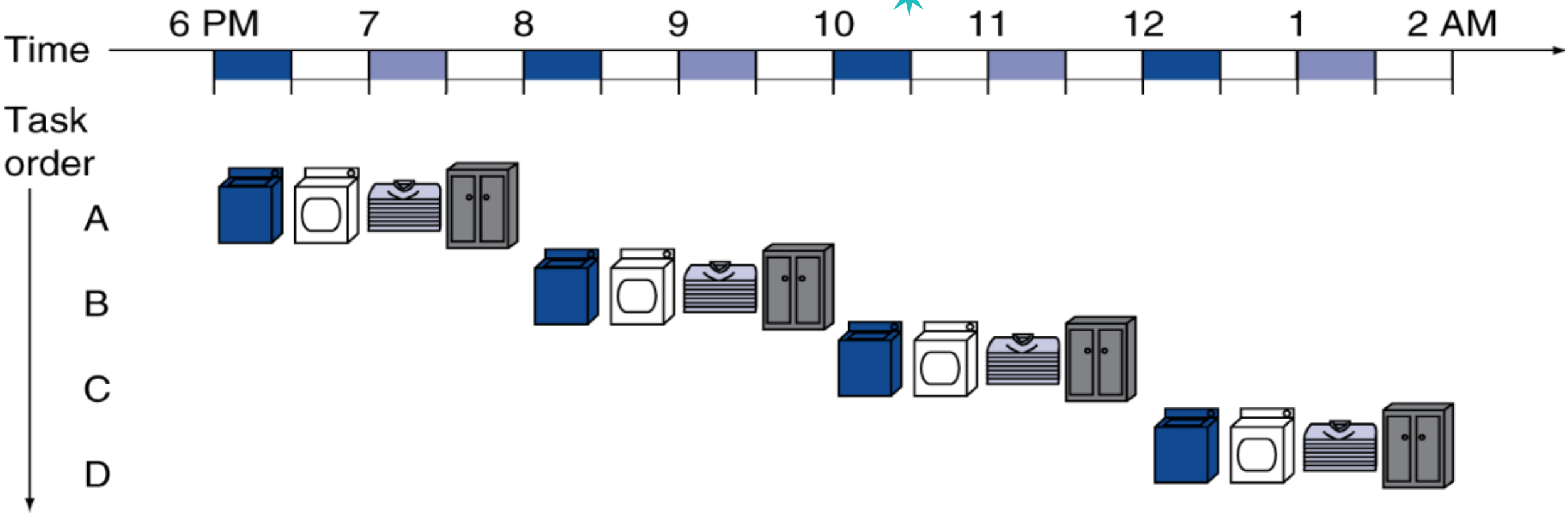
Several instruction are executed at the same time to enhance the performance

Motivation Example: Sequential Laundry Processing



Assume you have a set of clothes to wash: SET A, B, C, D

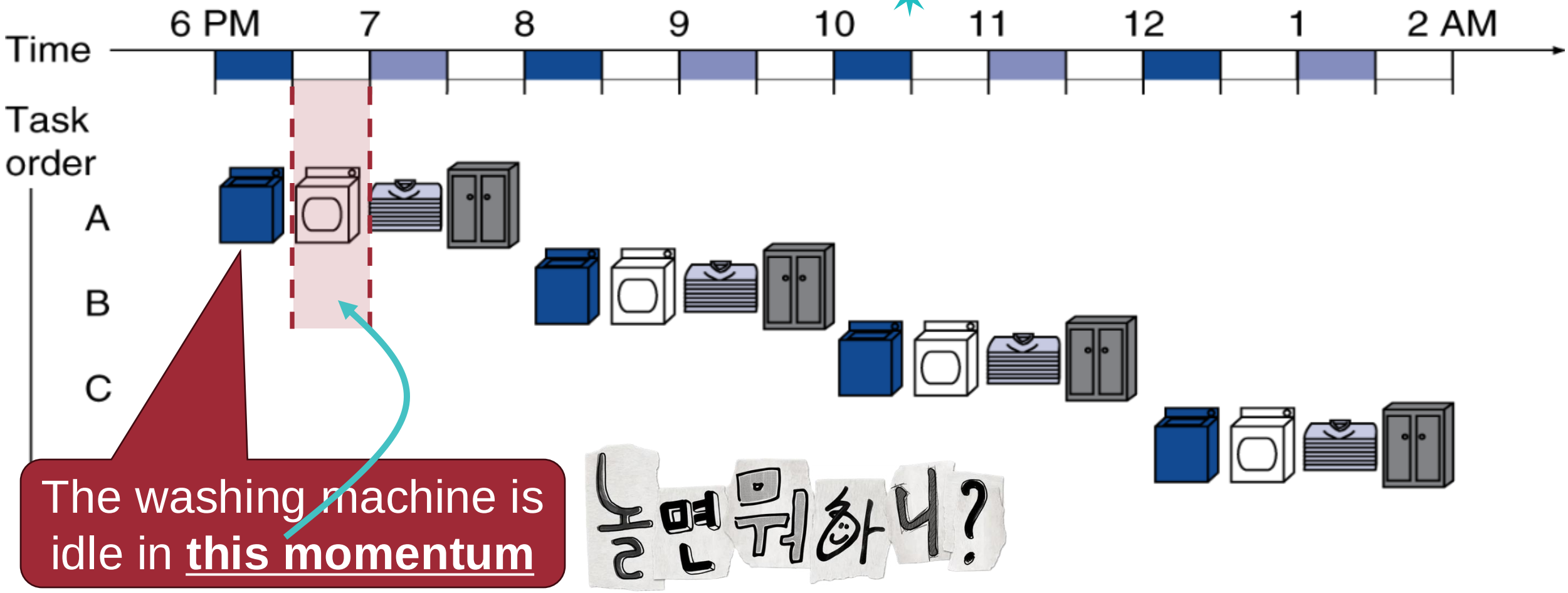
Motivation Example: Sequential Laundry Processing



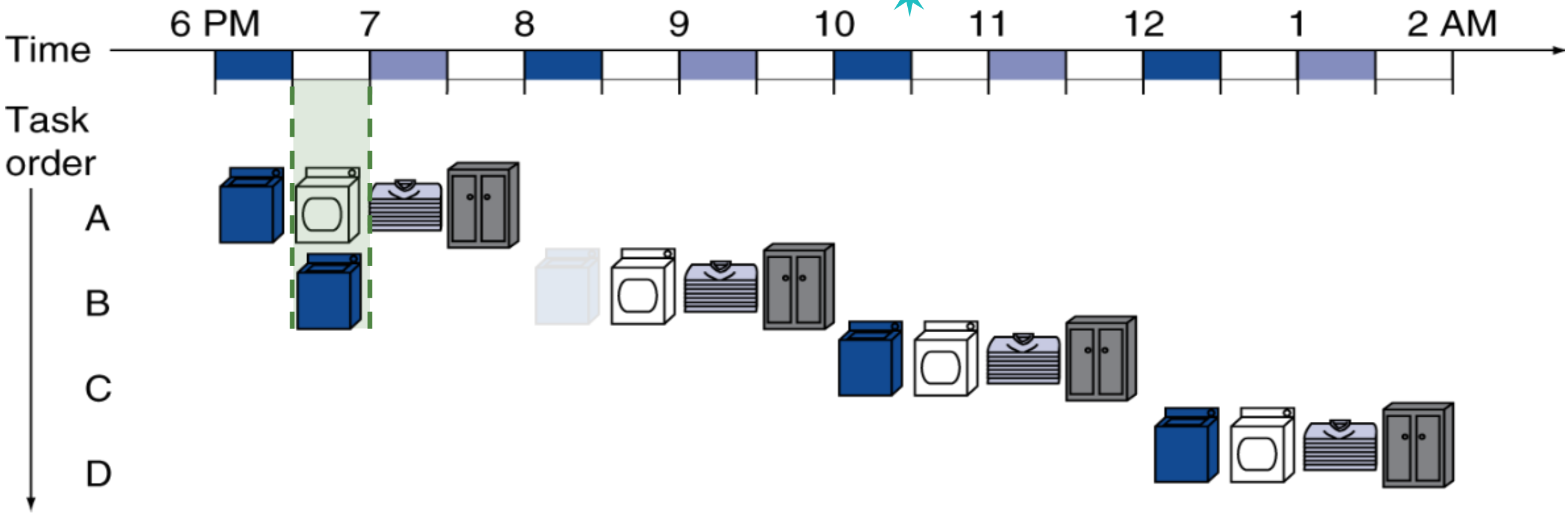
Single- and multi-cycle implementation start the next task only after the previous task has been completed

Execution time:
2 + 2 + 2 + 2
= 8 hours

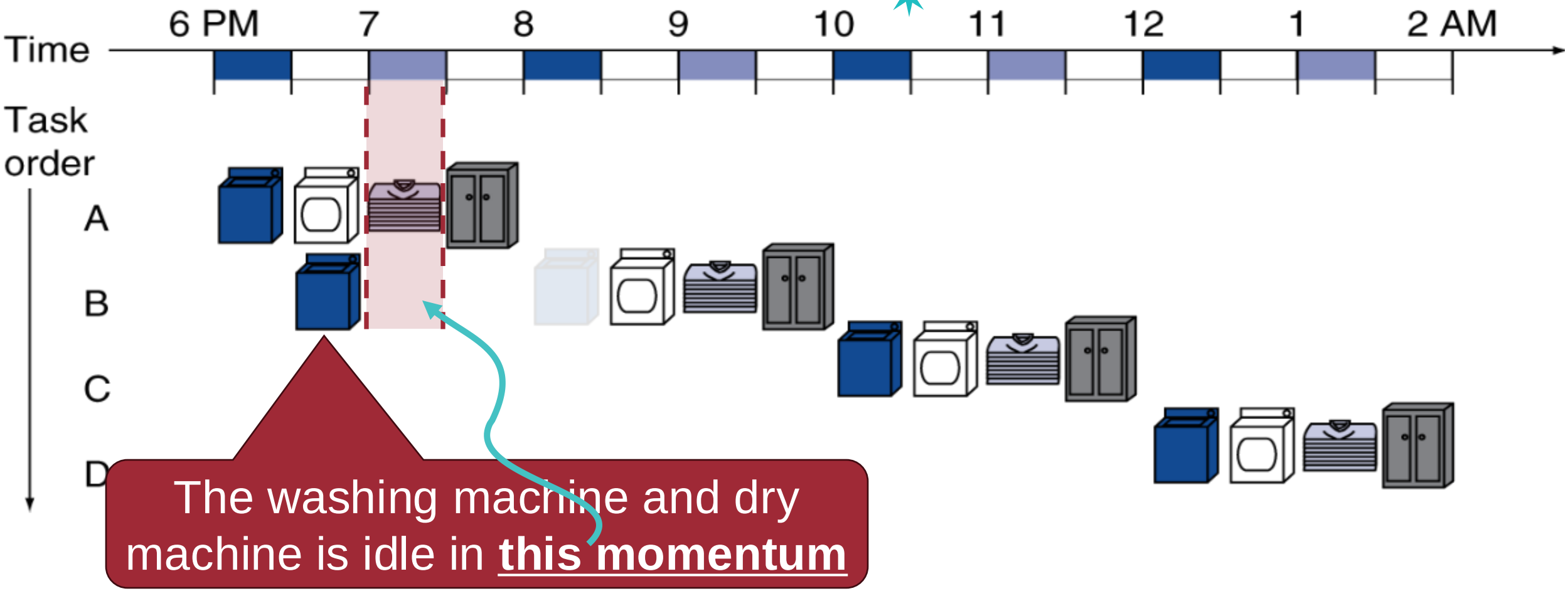
Let's Think about it in a More Natural Way!



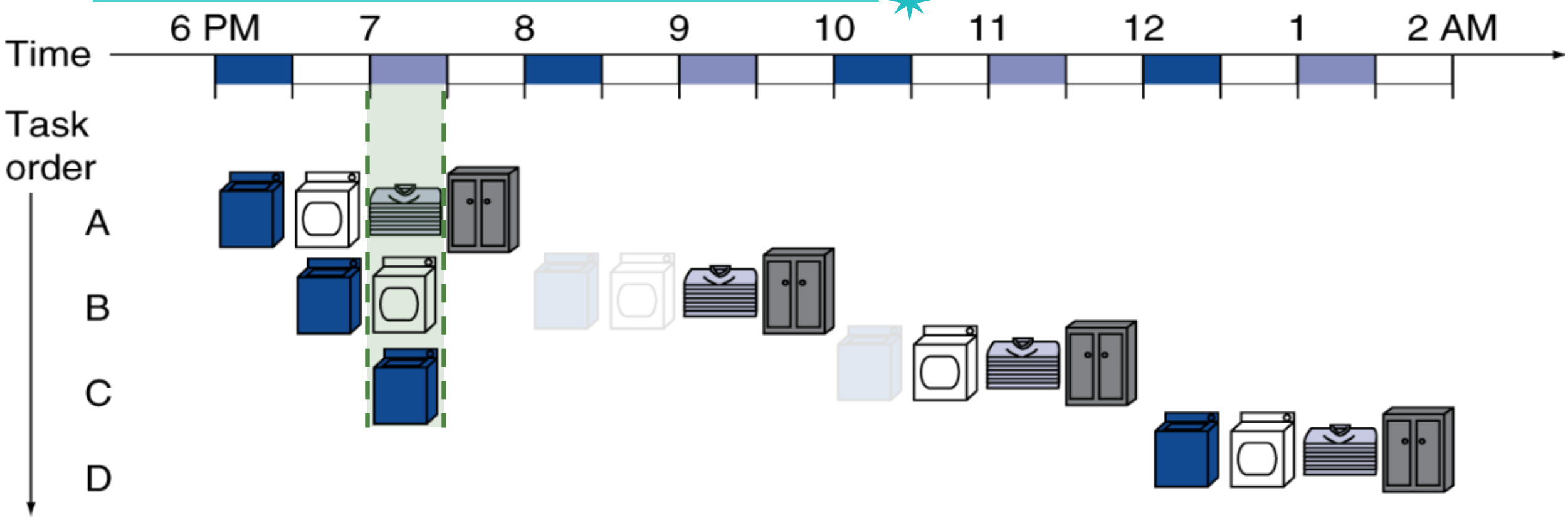
Solution: Execute Washing at the Same Time



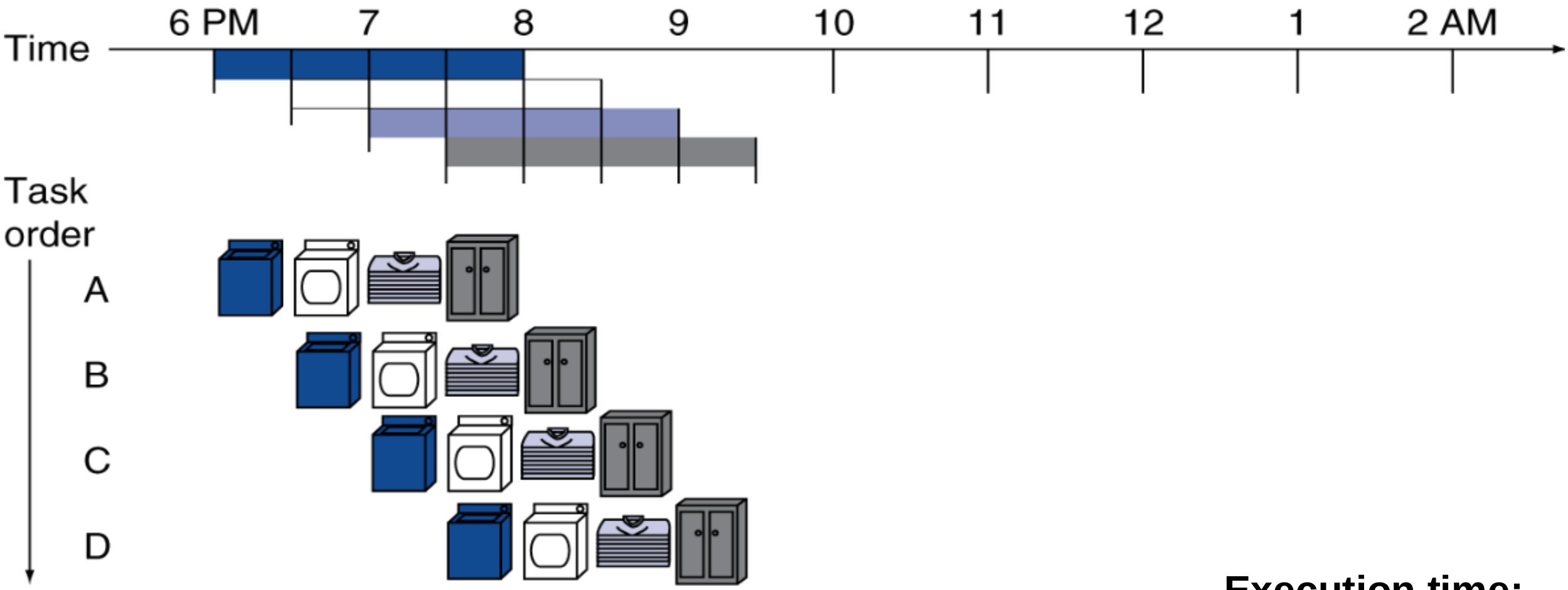
Problem Again...



Solution: Execute Washing/drying at the Same Time



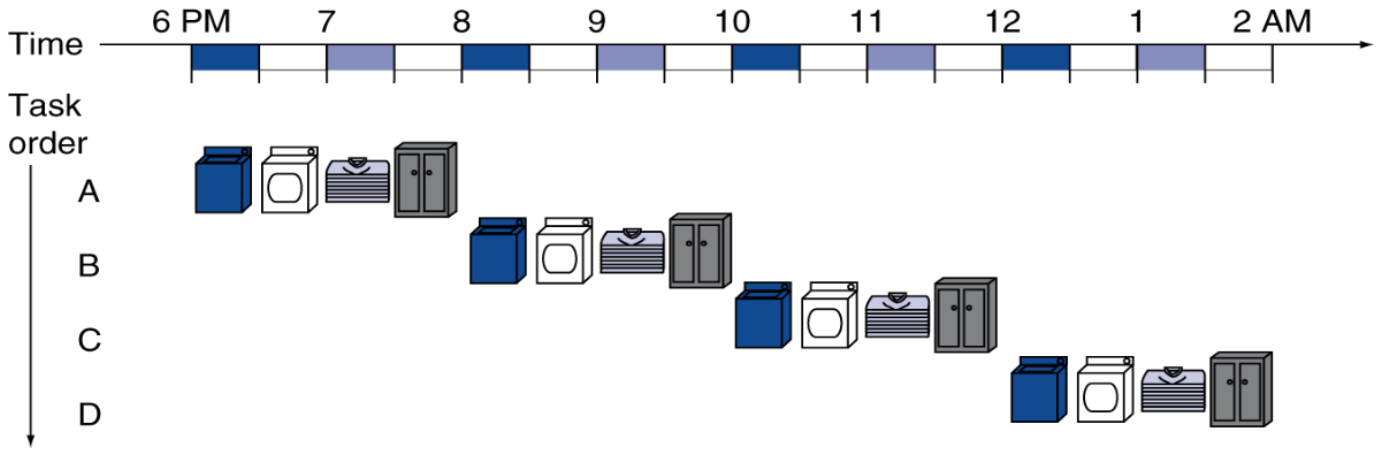
Pipelining Analogy



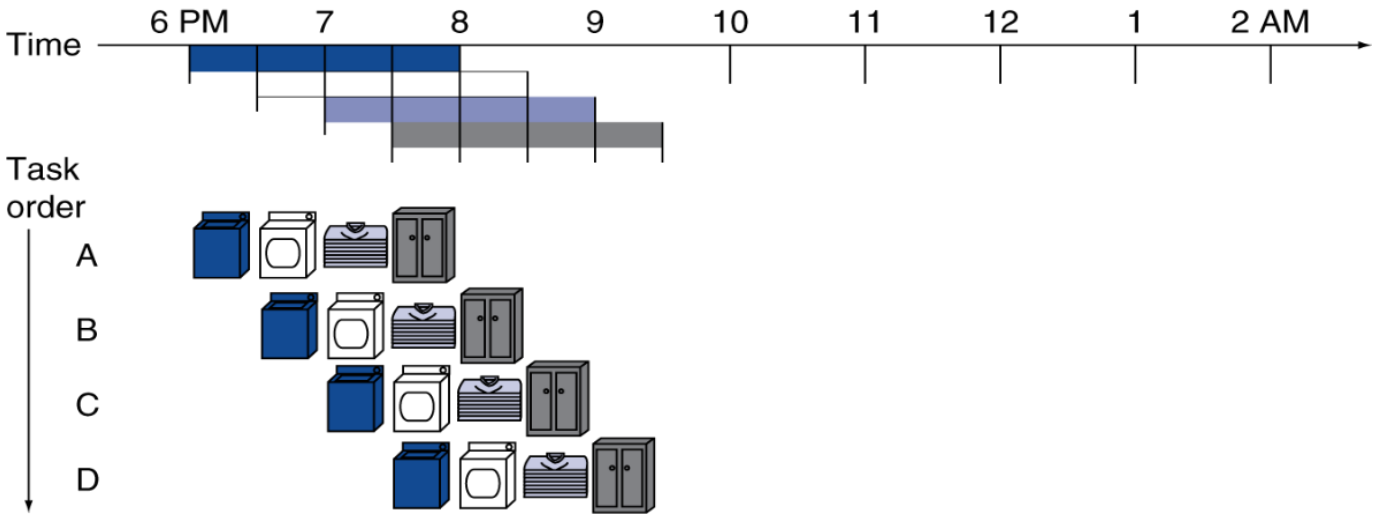
Pipelining overlaps execution

Execution time:
2 + 0.5 + 0.5 + 0.5
= 3.5 hours

Performance Comparison: Speedup



Without pipelining execution time:
2 + 2 + 2 + 2
= 8 hours

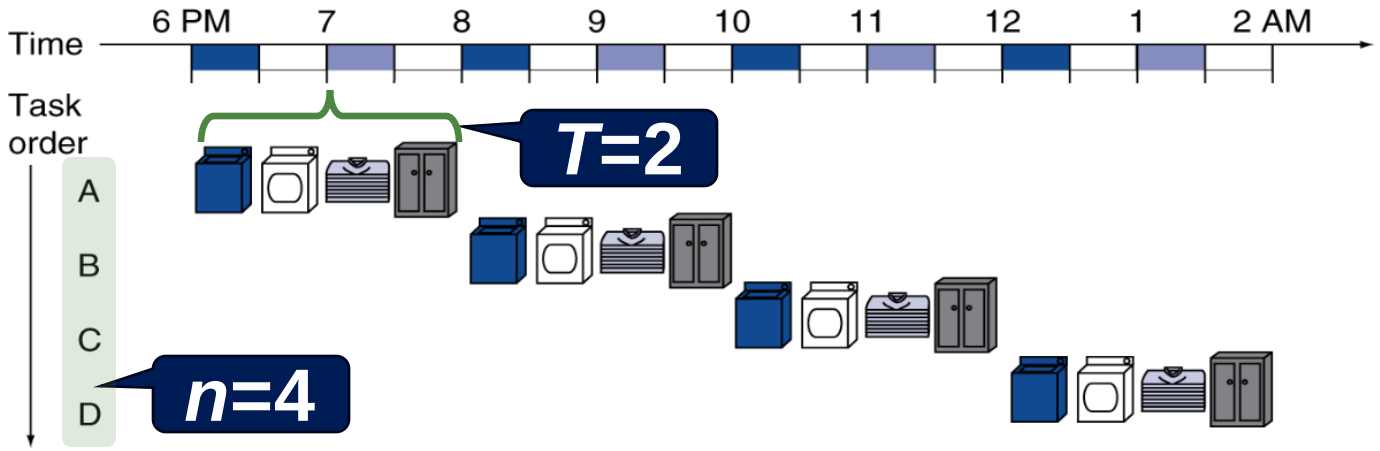


Pipelining execution time:
2 + 0.5 + 0.5 + 0.5
= 3.5 hours

Speedup?
8/3.5 = 2.3

Let's Think about Speedup in General Terms

Suppose there are ***n* jobs to execute** and **each job takes *T* hours**

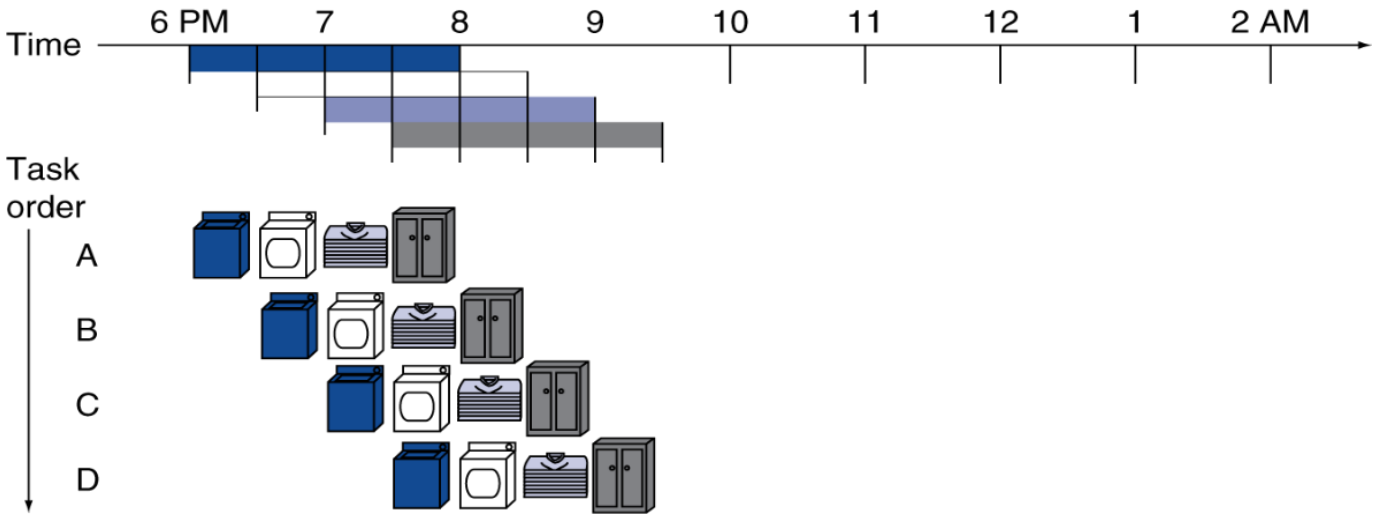


Without pipelining execution time:

2 + 2 + 2 + 2

= 8 hours

***n*T**



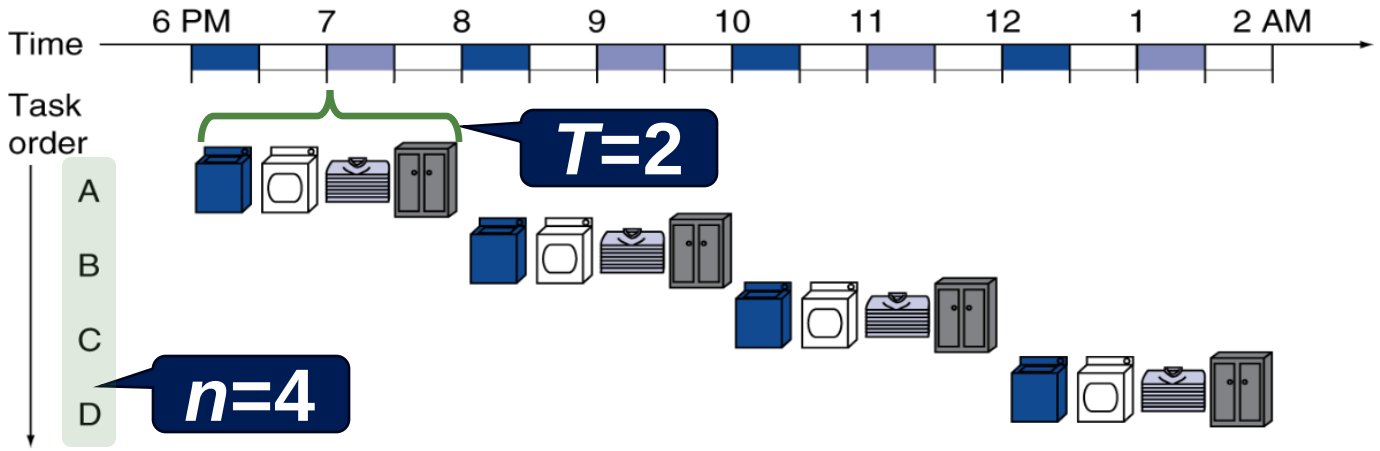
Pipelining execution time:

2 + 0.5 + 0.5 + 0.5

= 3.5 hours

Let's Think about Speedup in General Terms

Suppose there are ***n* jobs to execute** and **each job takes *T* hours**



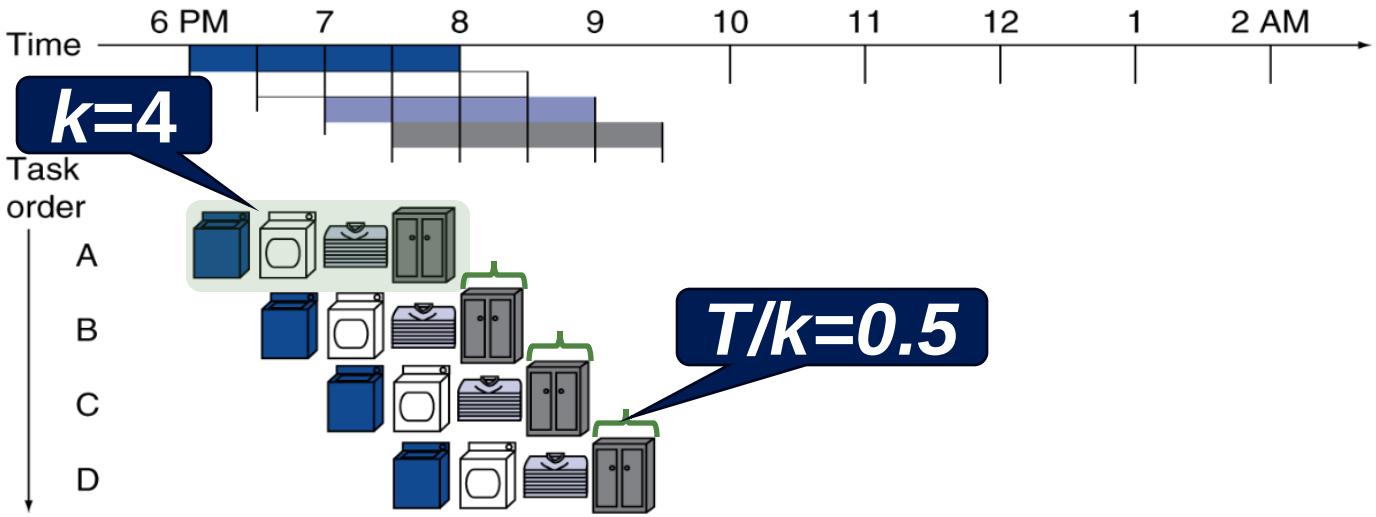
Without pipelining execution time:

2 + 2 + 2 + 2

= 8 hours

***n*T**

Suppose we have *k* stage



(n+k-1) * T/K

Pipelining execution time:

2 + 0.5 + 0.5 + 0.5

= 3.5 hours

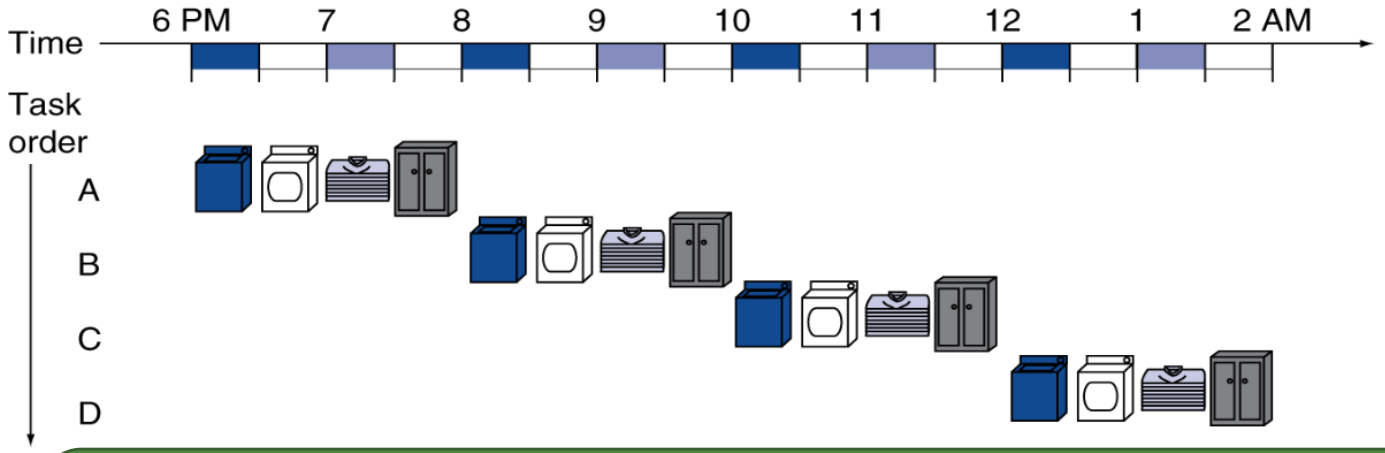
Pipelining Speedup

- Execution time *without pipelining*: $n*T$
- Execution time *with pipelining*: $(n+k-1)*T/k$
- **Pipelining speedup**: $\frac{n*T}{(n+k-1)*T/k} = \frac{n*k}{n+k-1}$
- **If *n* goes infinity, pipelining speedup becomes *k***

$$\lim_{n\to\infty} \frac{n*k}{n+k-1} = k$$

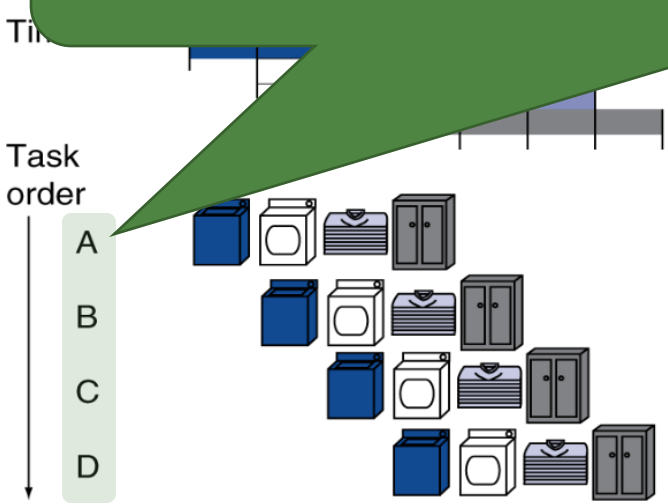
If we have a large number of jobs, we can achieve **a speedup of about *k* times** through pipelining

Performance Comparison: Speedup



Without pipelining execution time:
2 + 2 + 2 + 2
= 8 hours

If we have a large number of jobs, we can achieve **about 4 speedup (x4)** through pipelining



time:
2 + 0.5 + 0.5 + 0.5
= 3.5 hours

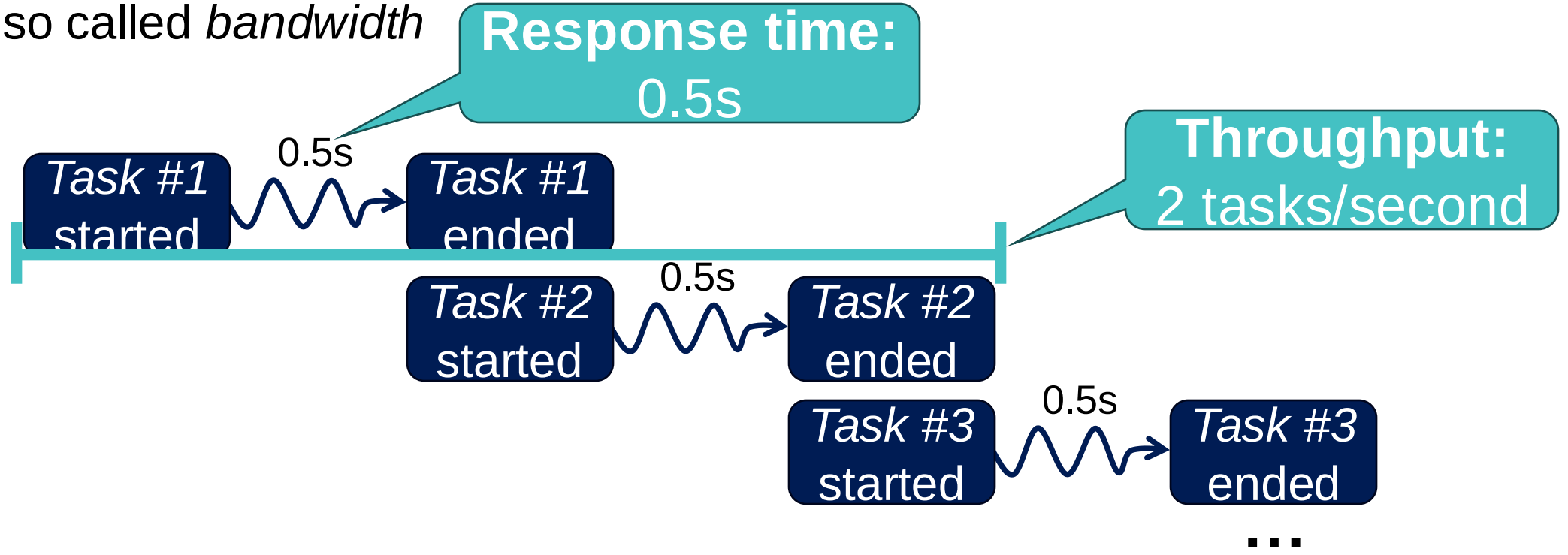
Speedup?
8/3.5 = 2.3

Pipeline Lessons

- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload

Performance Metrics

- **Response time**: the time between the start and completion of a task
  - Also called *execution time*, *latency*
- **Throughput:** total work done per unit time
  - E.g., *# of tasks/transactions/… per hour*
  - Also called *bandwidth*

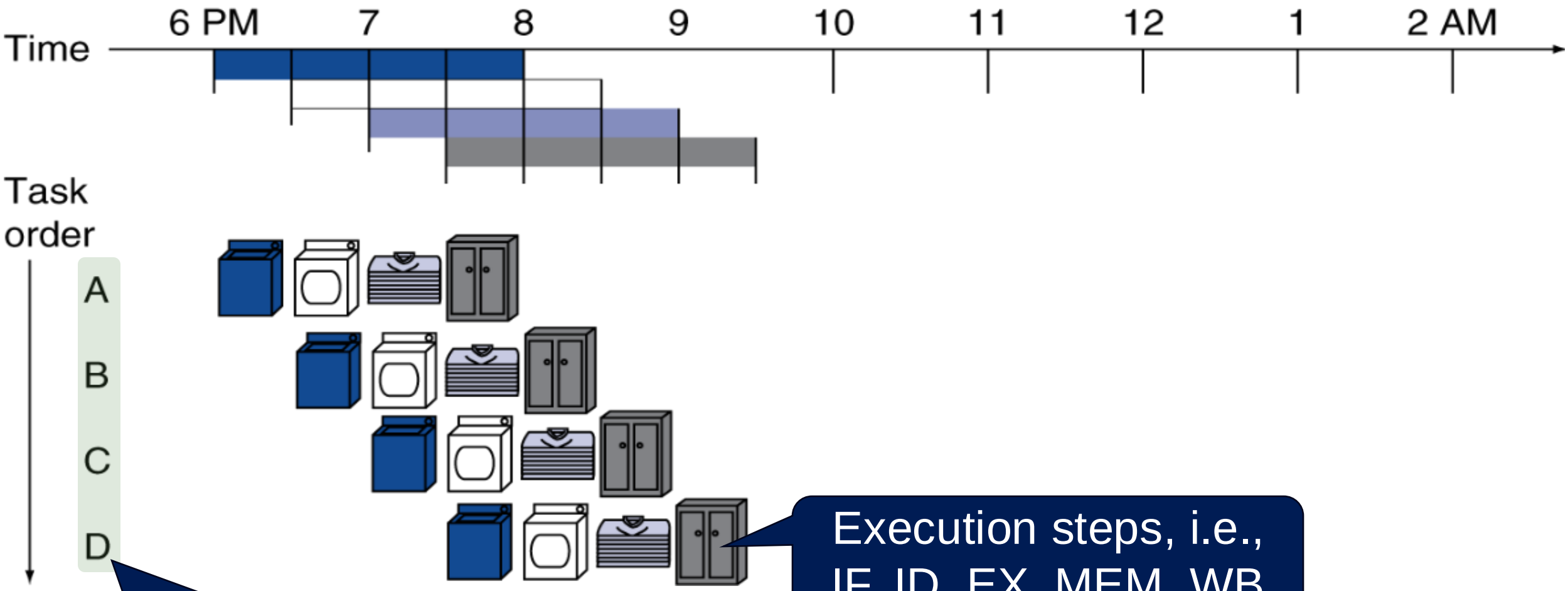


Pipeline Lessons

- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = # pipe stages

Pipelining Overview

Pipelining Overview

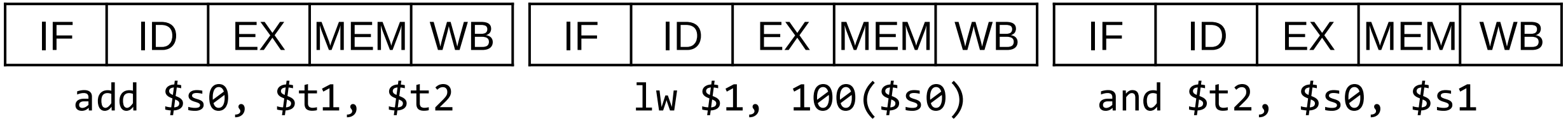


MIPS Pipelining

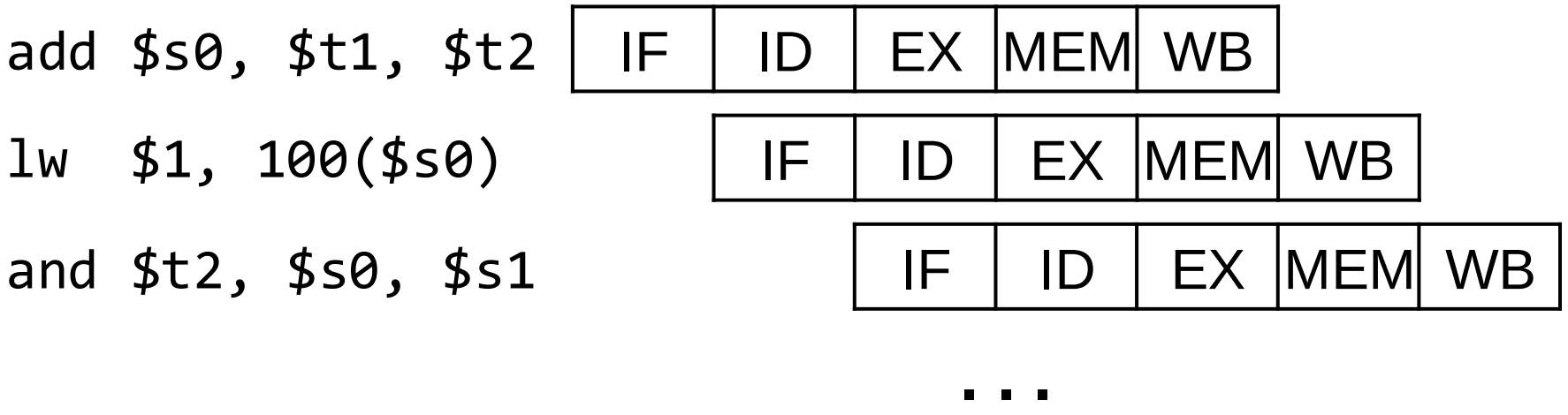
- Five stages (*k*=5), one step per stage
  1. **IF**: Instruction fetch from memory
  2. **ID**: Instruction decode & register read
  3. **EX**: Execute operation or calculate address
  4. **MEM**: Access memory operand
  5. **WB**: Write result back to register

MIPS Pipelining

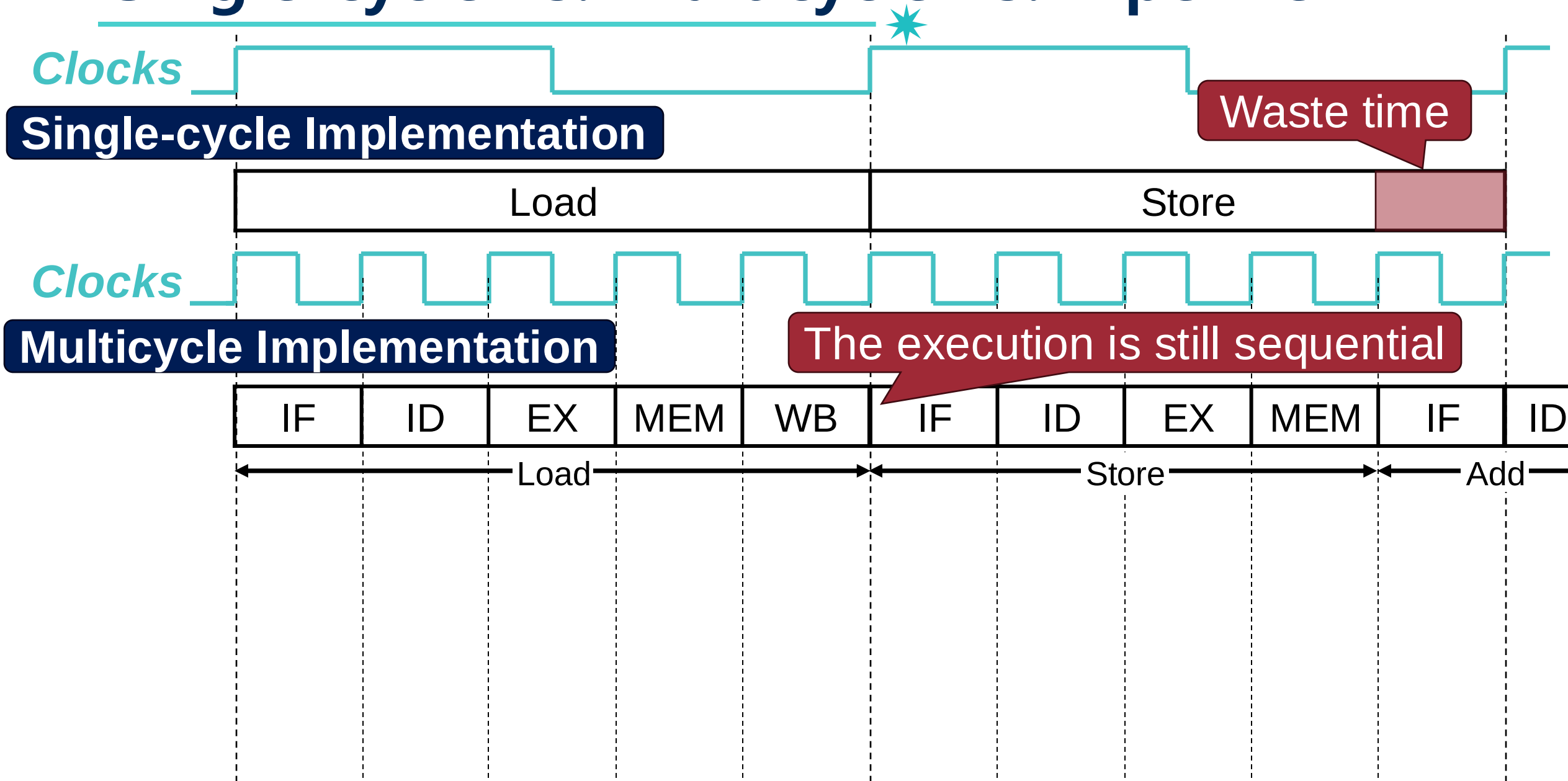
- **Sequential execution**



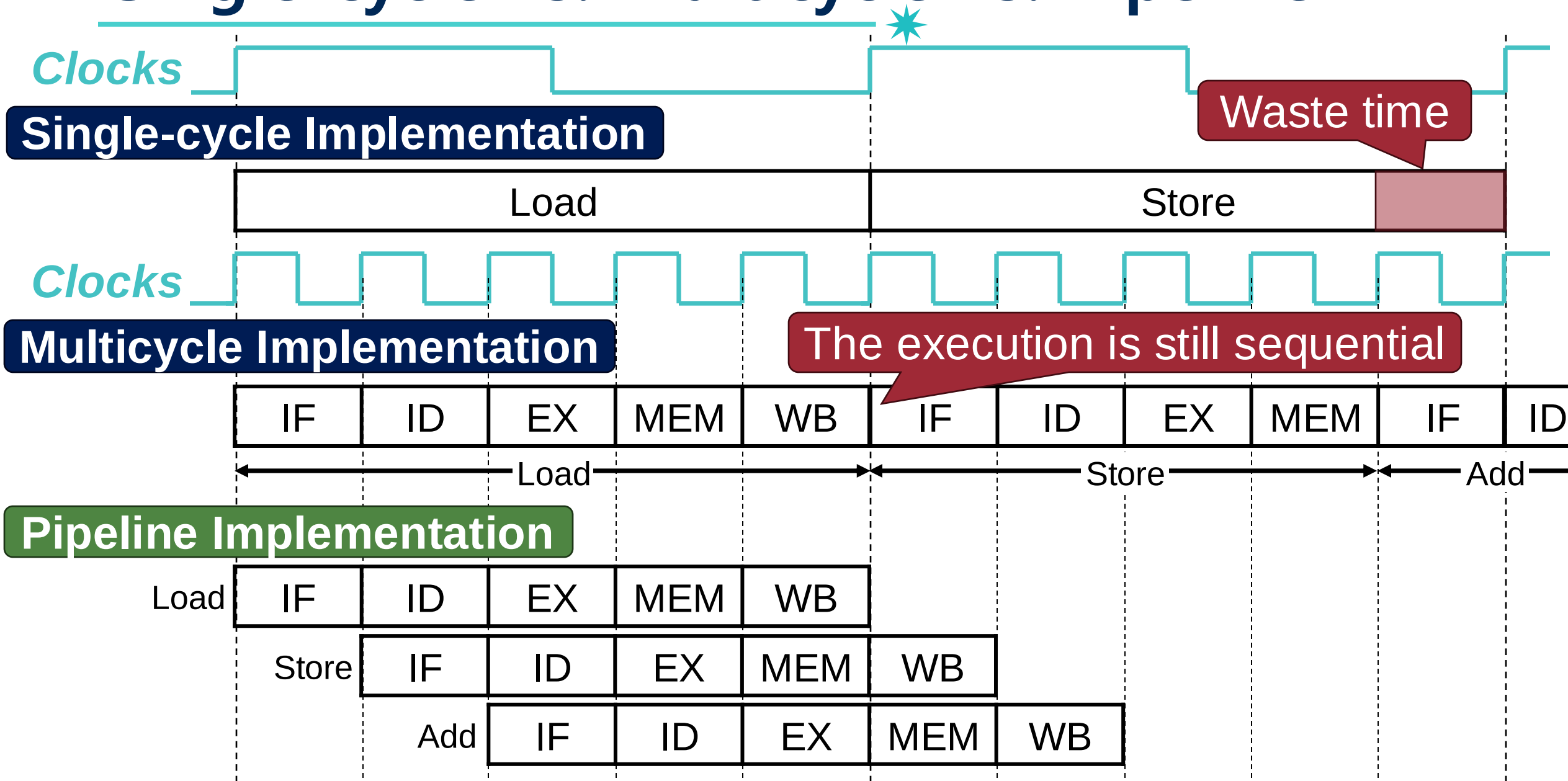
- **Pipelined execution**



Single-cycle vs. Multicycle vs. Pipeline



Single-cycle vs. Multicycle vs. Pipeline

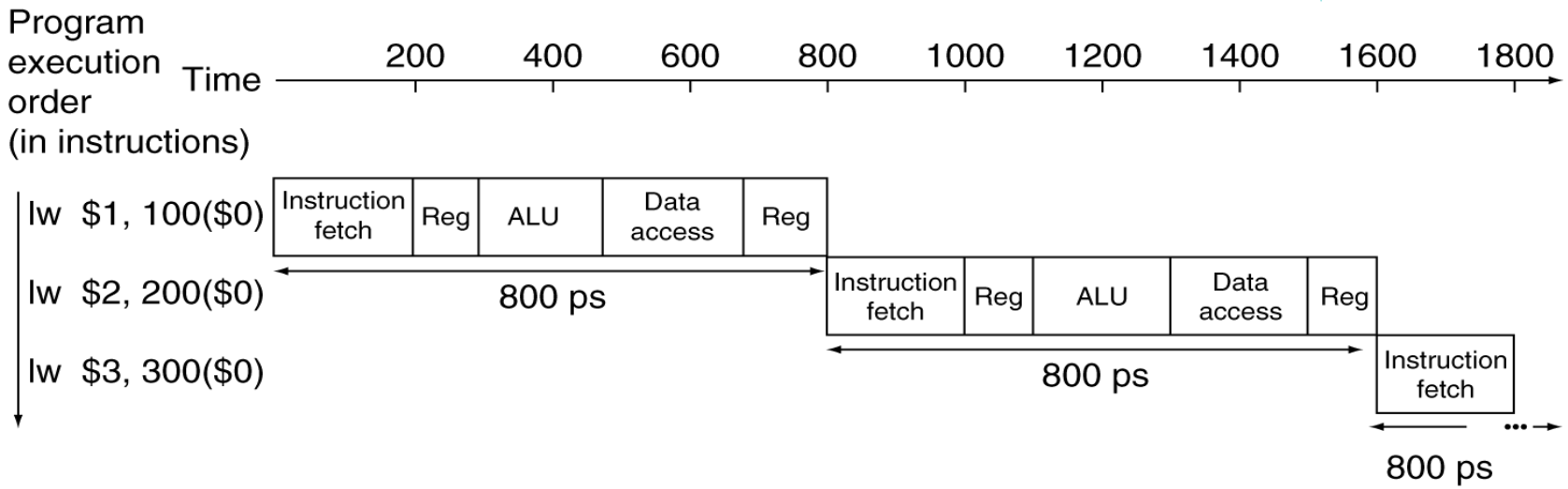


Pipeline Performance

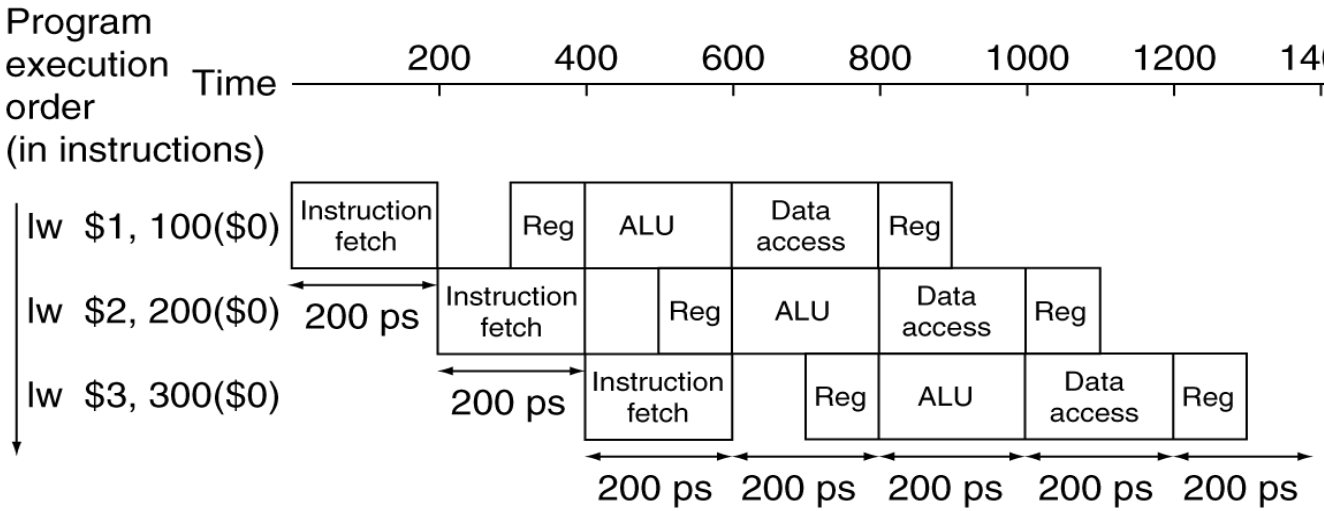
- Calculate cycle time assuming negligible delays except:
  - Memory access (200ps), ALU (200ps), Register access (100ps)

Instruction	Functional units used by the instruction class					
	Instruction fetch	Instruction decode	Execution	Memory Access	Register Write-back	
R-type	200ps	100ps	200ps		100ps	→ 600ps
Load word	200ps	100ps	200ps	200ps	100ps	→ 800ps
Store word	200ps	100ps	200ps	200ps		→ 700ps
Conditional Branch	200ps	100ps	200ps			→ 500ps
Jump	200ps					→ 200ps

Pipeline Performance: Speed Up?



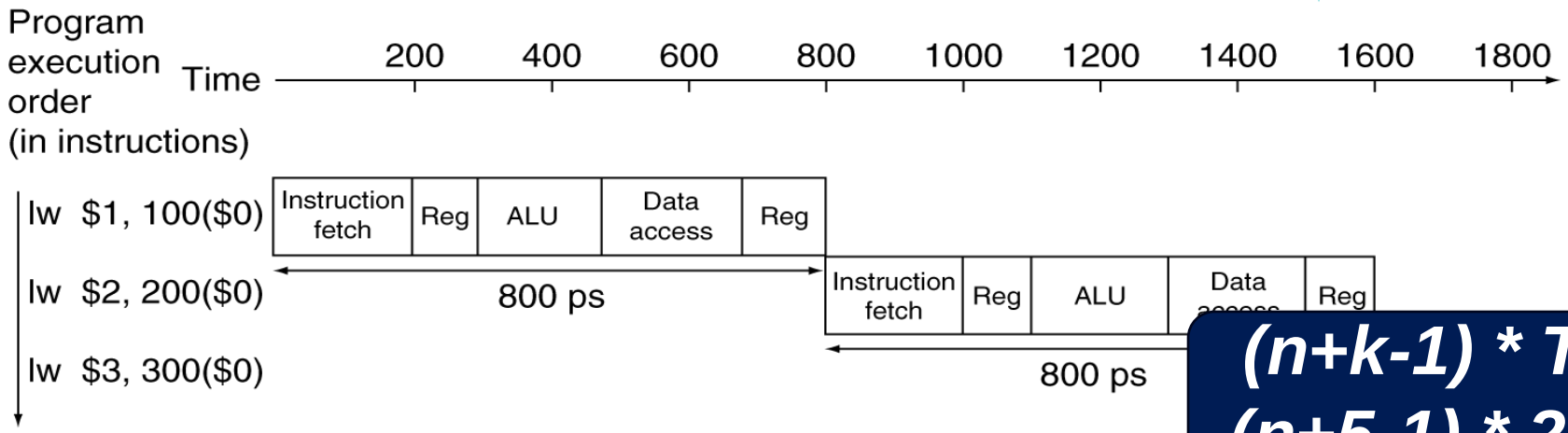
Without pipelining execution time:
800 + 800 + 800
= 2400 ps



Pipelining execution time:
800 + 200 + 200 + 200
= 1400 ps

Speedup?
2400/1400 ≅ 1.7

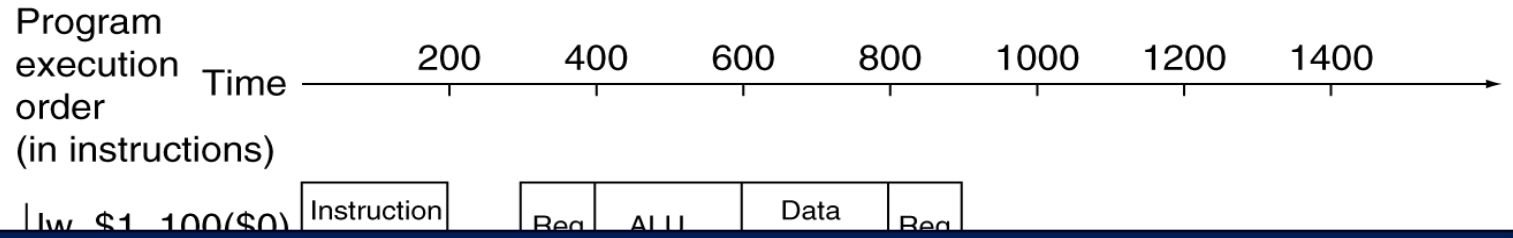
What if We have 1,000,003 Instructions?



Without pipelining execution time:

n*T = n*800ps

(n+k-1) * T/K = (n+5-1) * 200ps



Pipelining execution time:

$$\text{Speedup} = \frac{n * 800ps}{(n+4) * 200ps} \quad (n=1{,}000{,}003)$$

$$= 800{,}002{,}400 \text{ ps} / 200{,}01{,}400 \text{ ps} \cong 4.00$$

MIPS ISA is Designed to Work Well with Pipelining

- **All instructions are 32-bits**
  - Consistent and short instruction fetch/decode times
- **Few and regular instruction formats**
  - Consistent and short instruction decode time
- **Load/store addressing**
  - Fixed sequence: calculate address in 3rd stage, access memory in 4th stage

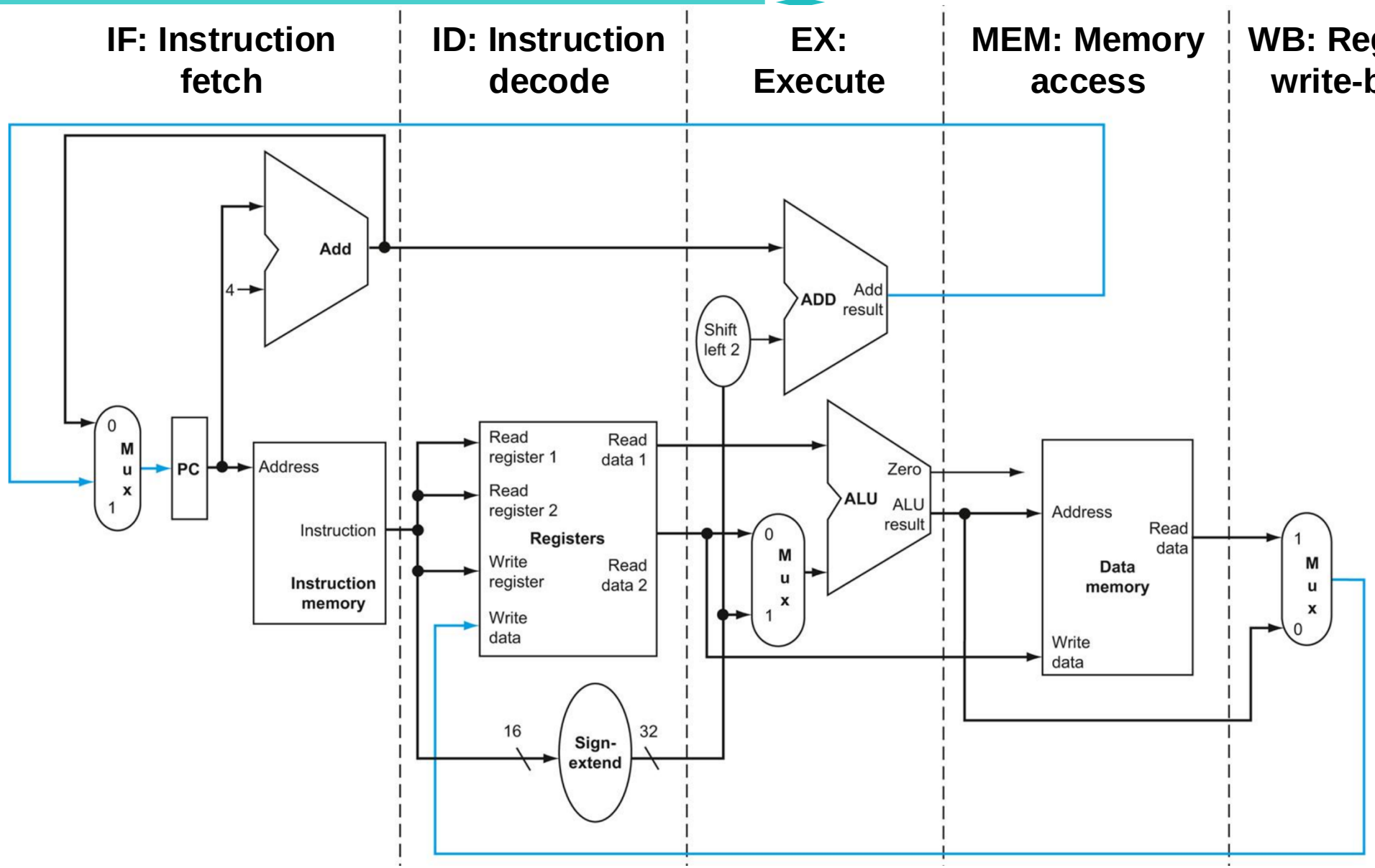
Pipeline Summary

- Pipelining improves performance by increasing instruction *throughput*
  - Executes multiple instructions in parallel
  - Each instruction has the same latency

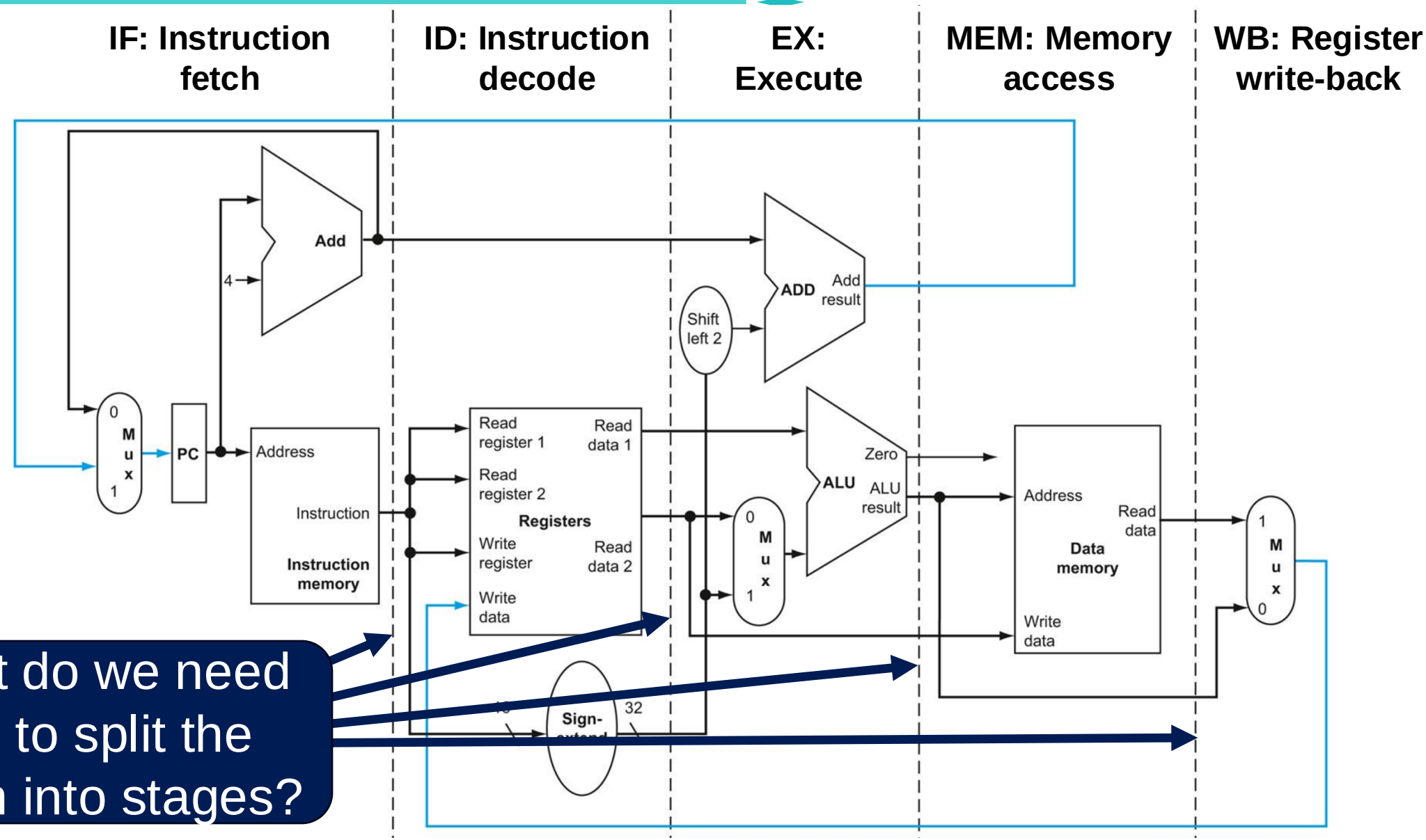
- The speedup depends on longest pipe stage and the number of pipe stages
  - Length of pipe stage ↓ → speedup ↑
  - # of pipe stage ↑ → speedup ↑
  - Instruction set design affects complexity of pipeline implementation

Pipelined Datapath

MIPS Pipelined Datapath



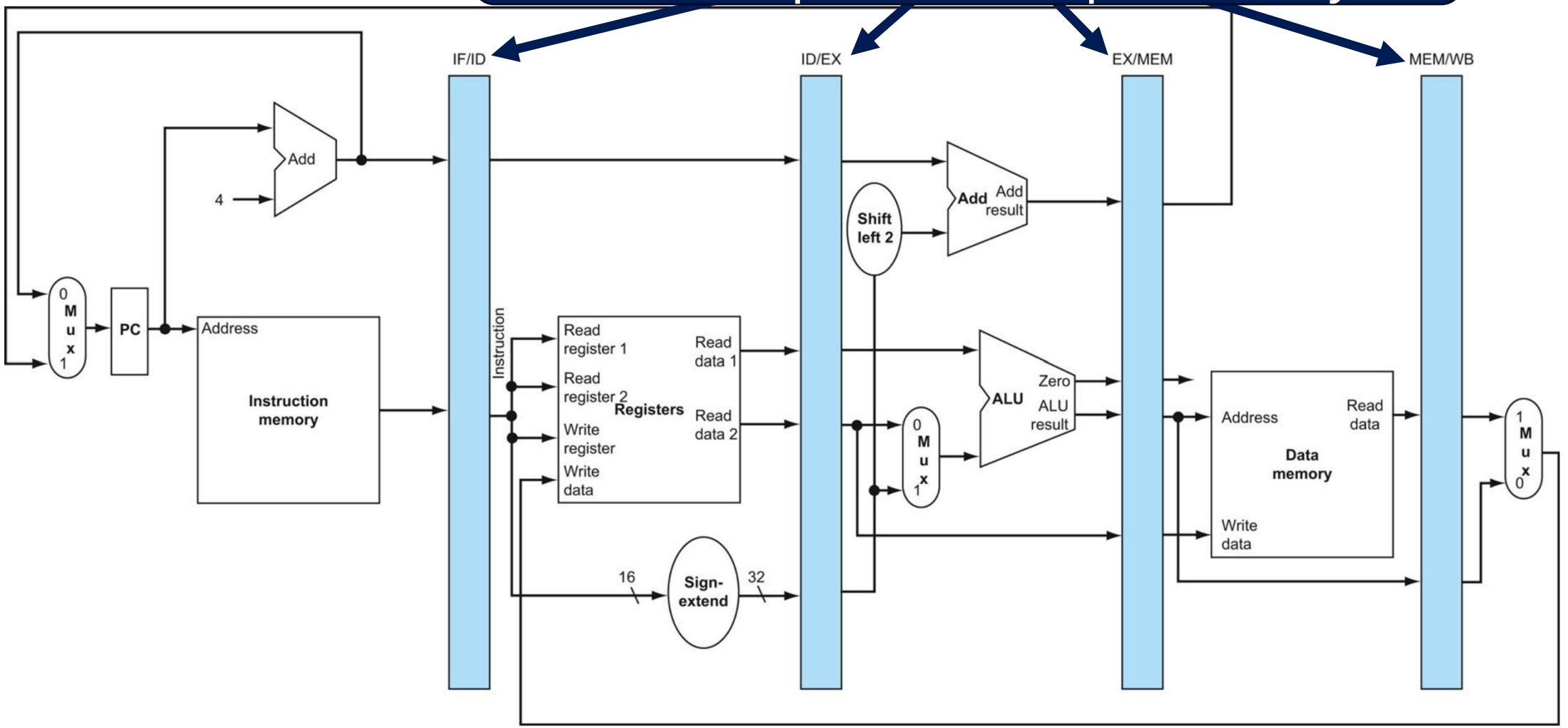
MIPS Pipelined Datapath



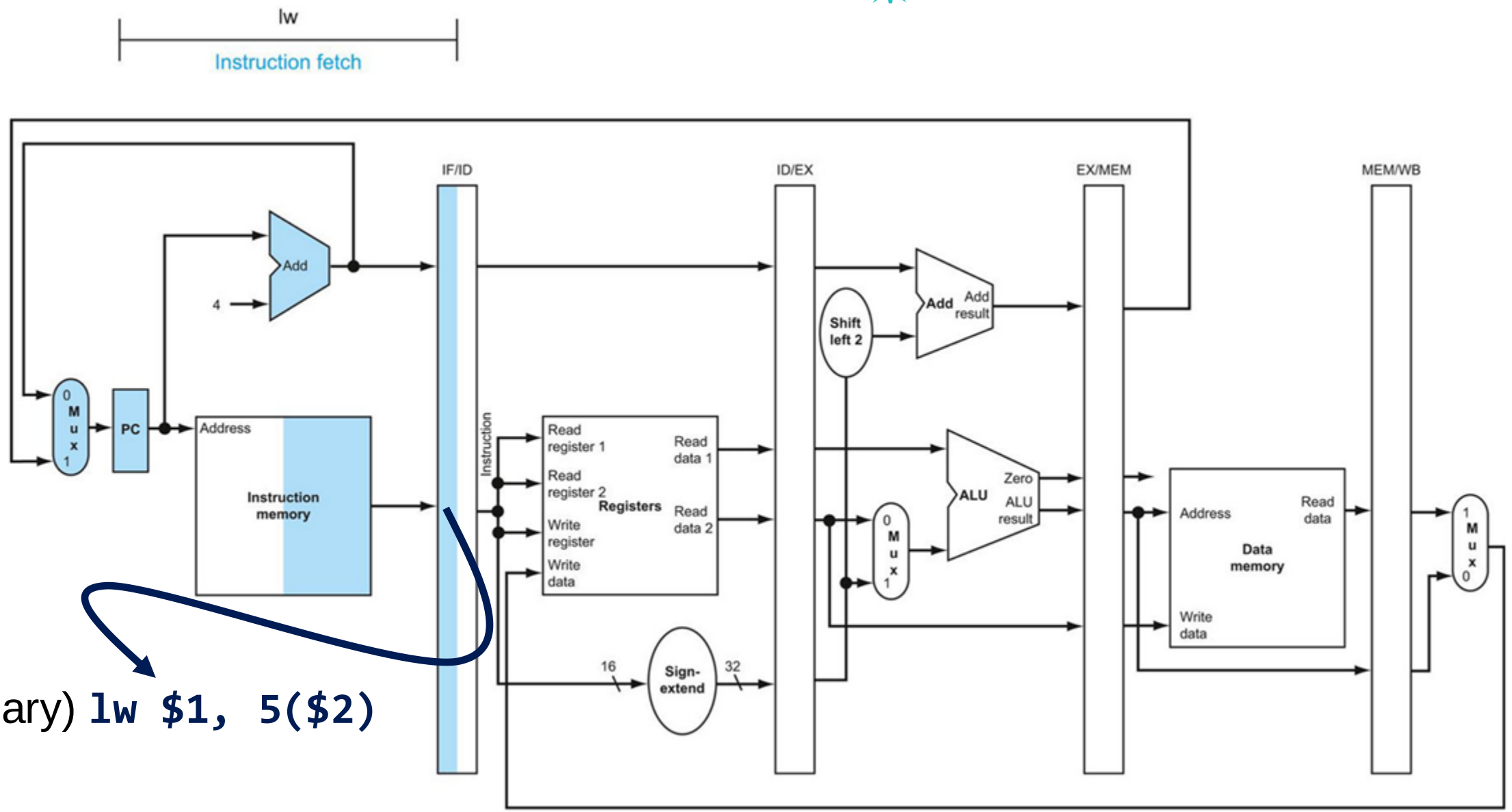
Q. What do we need to add to split the datapath into stages?

The Pipelined Version of the Datapath

Registers between stages to hold information produced in previous cycle

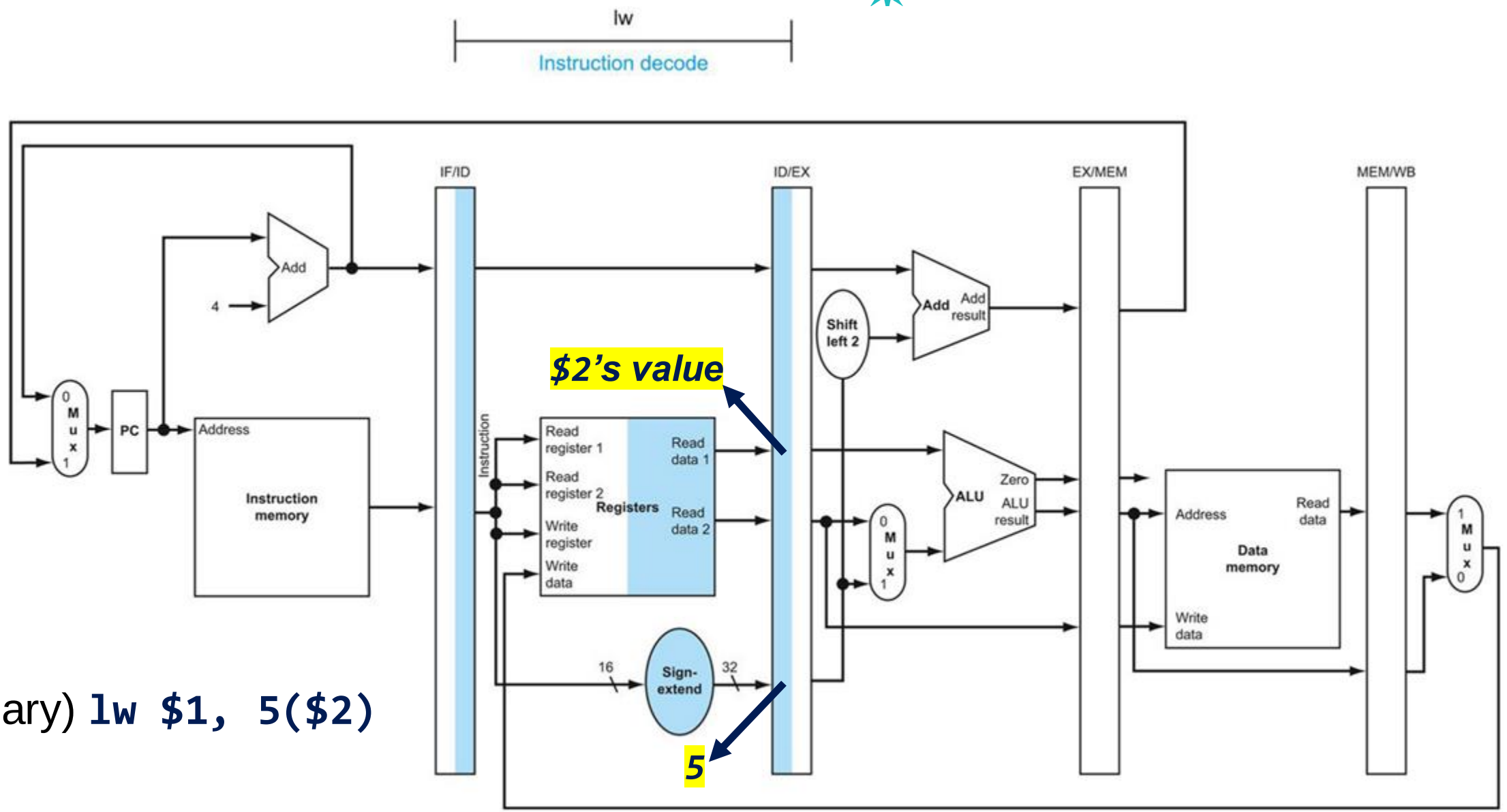


The Pipelined Version of the Datapath



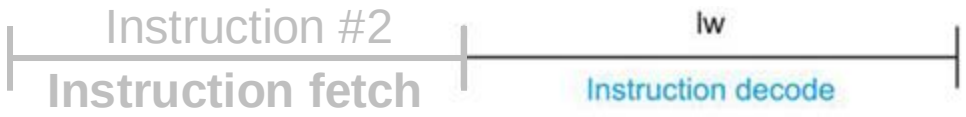
(binary) `lw $1, 5($2)`

The Pipelined Version of the Datapath

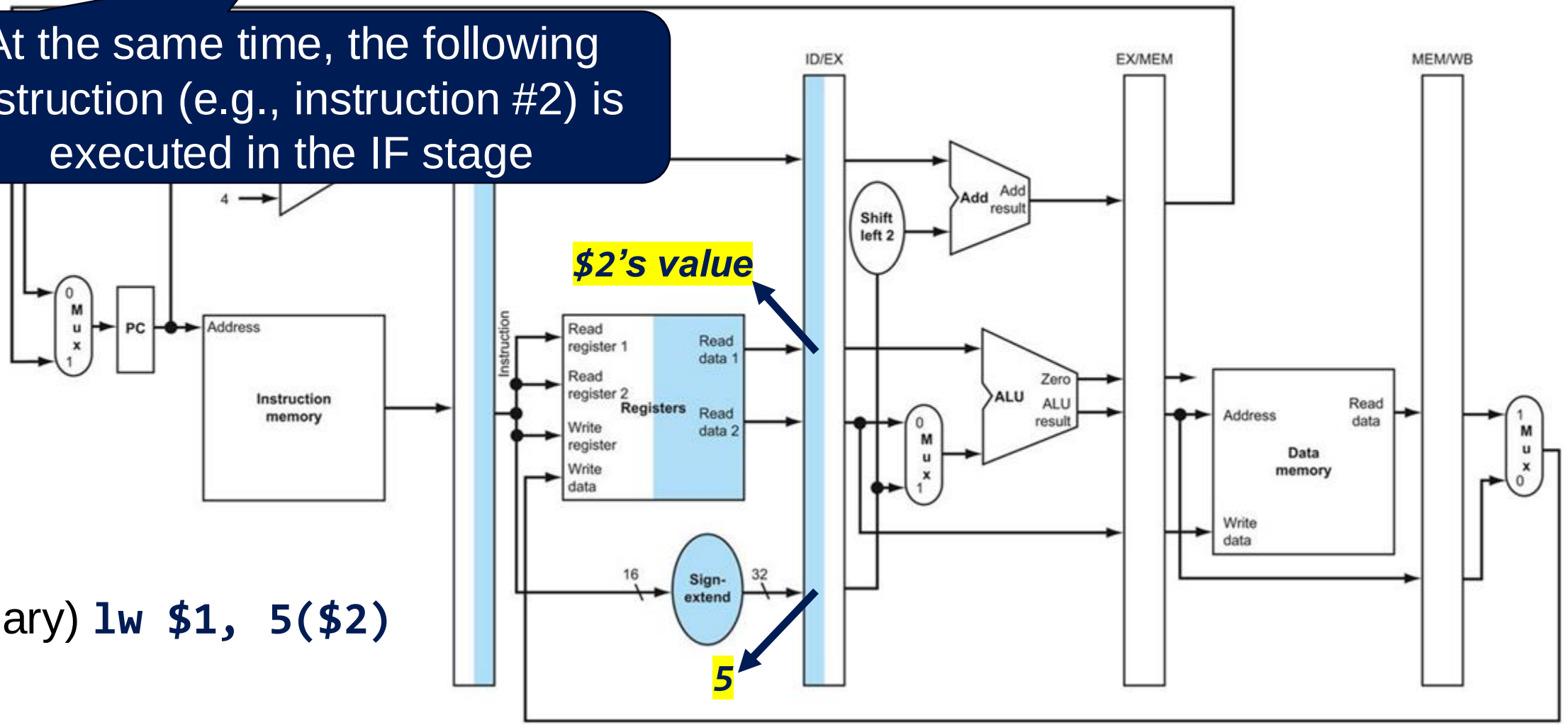


(binary) `lw $1, 5($2)`

The Pipelined Version of the Datapath

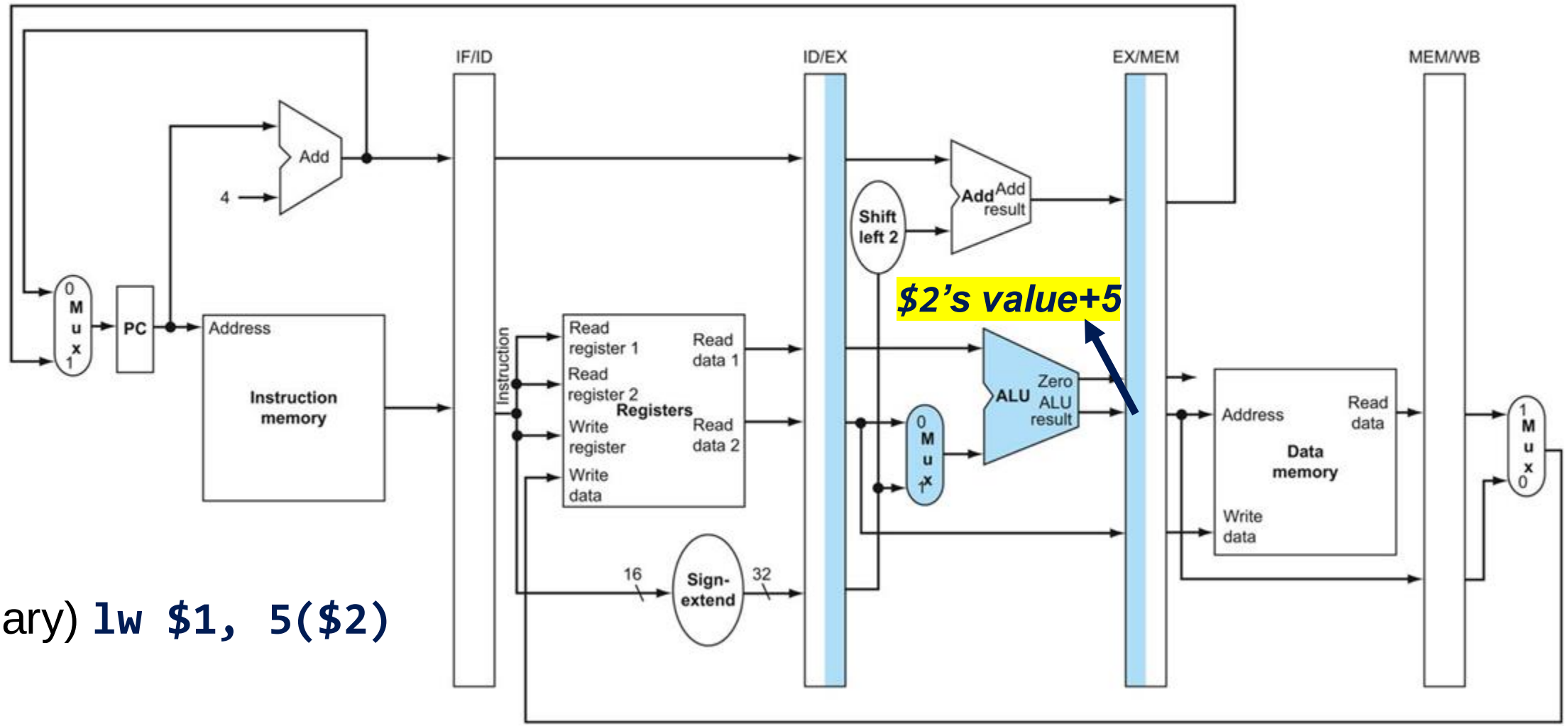


At the same time, the following instruction (e.g., instruction #2) is executed in the IF stage



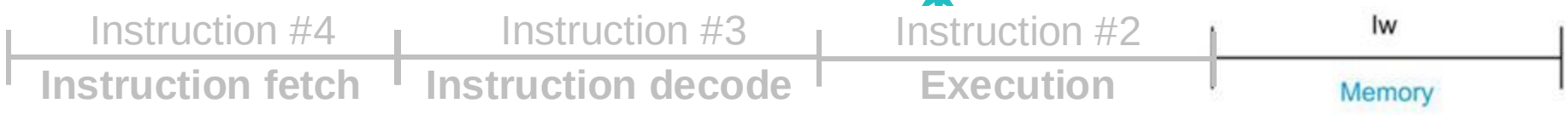
(binary) `lw $1, 5($2)`

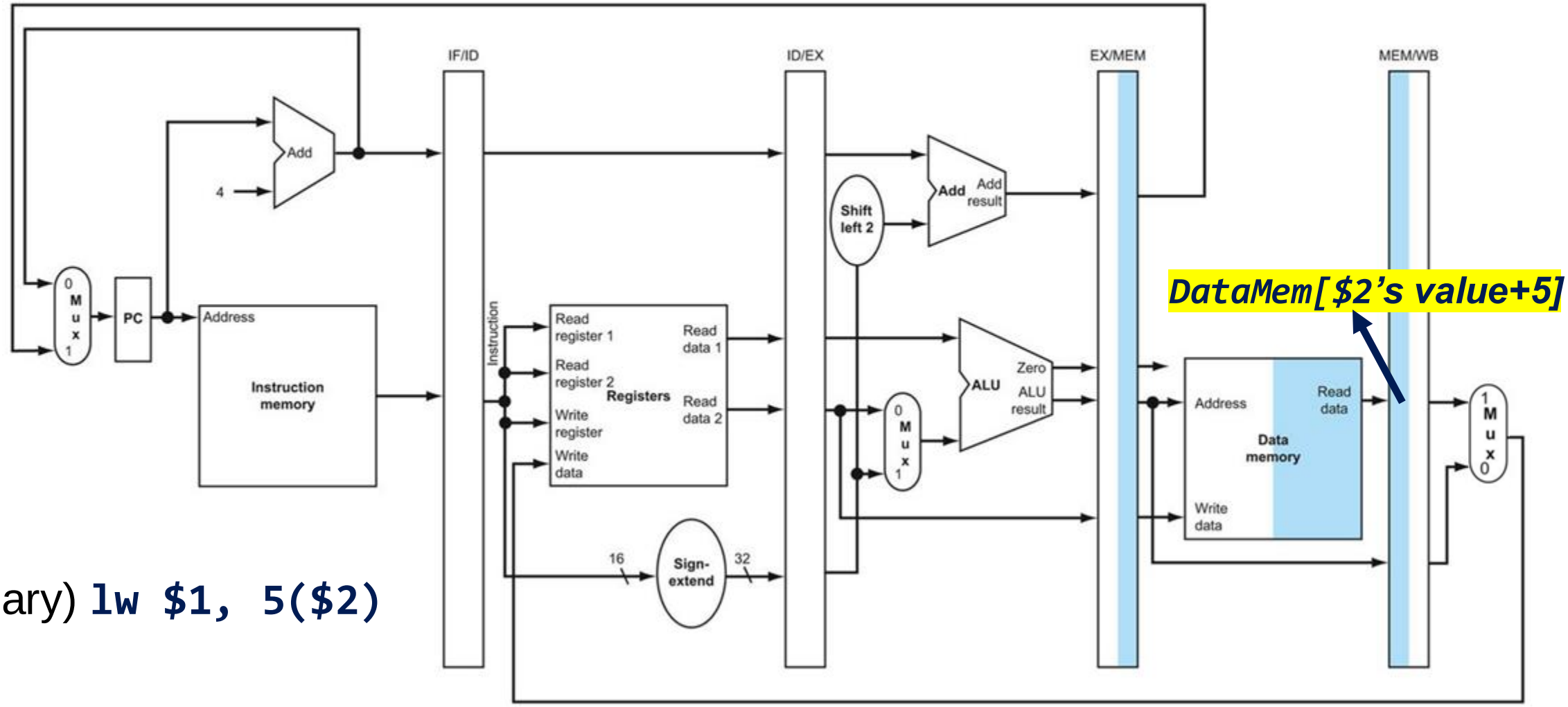
The Pipelined Version of the Datapath



(binary) `lw $1, 5($2)`

The Pipelined Version of the Datapath

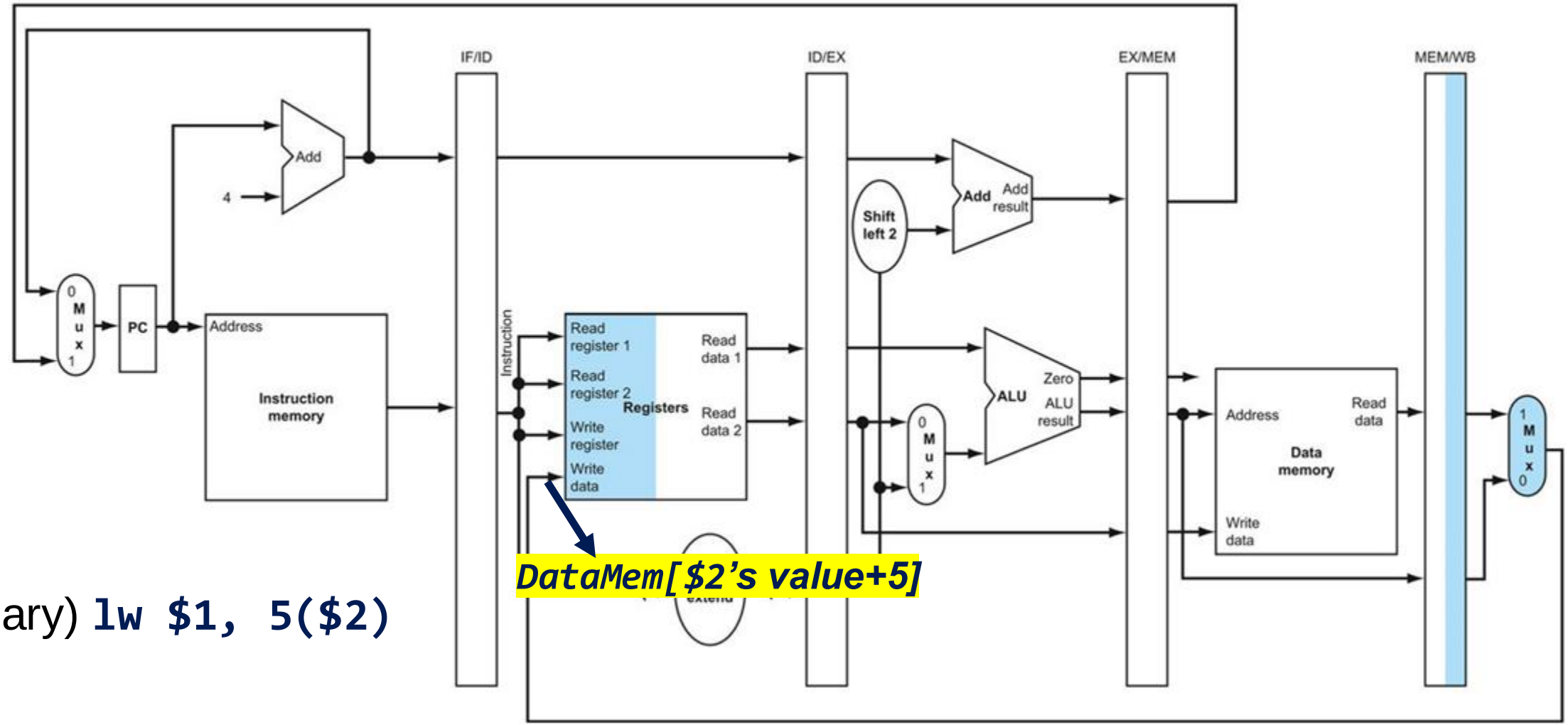




(binary) `lw $1, 5($2)`

The Pipelined Version of the Datapath

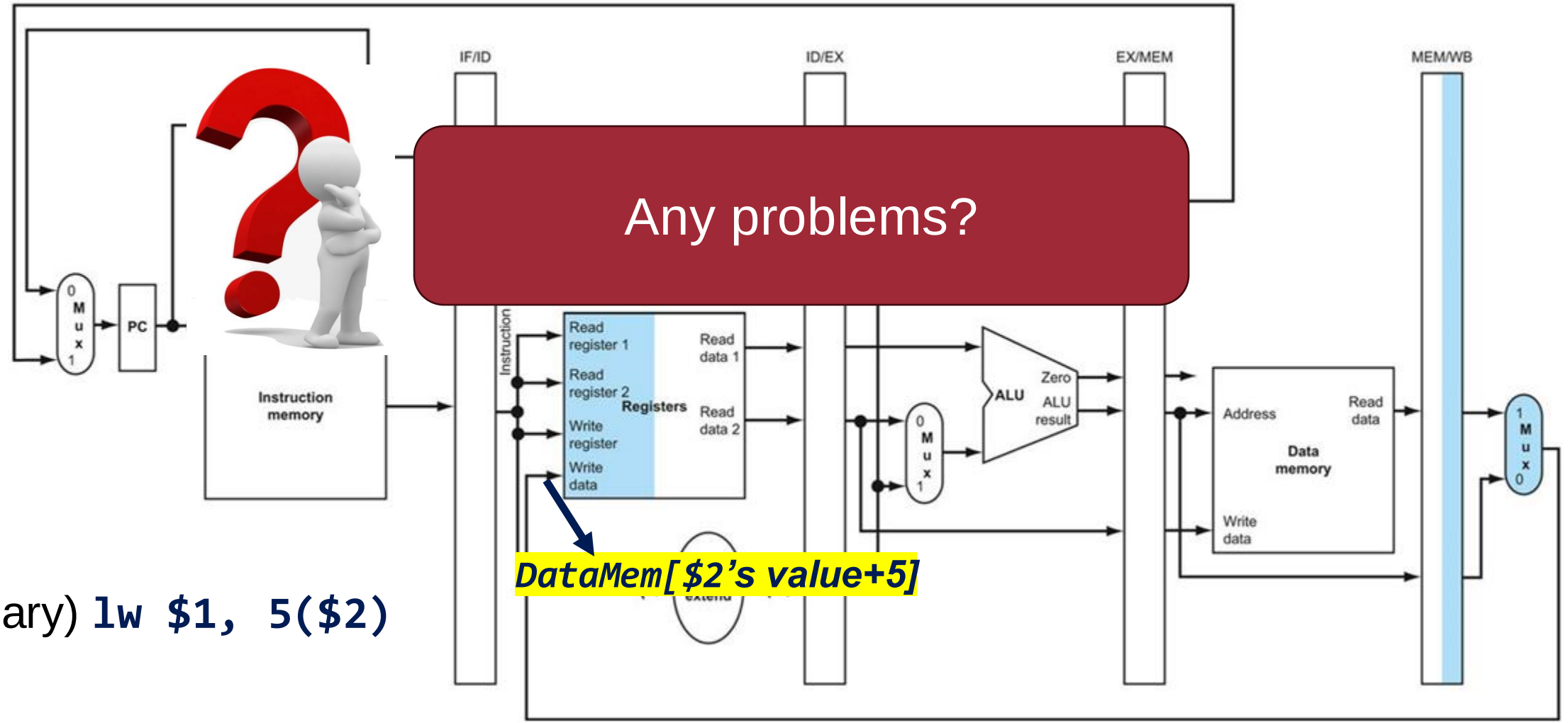
Instruction #5 Instruction fetch | Instruction #4 Instruction decode | Instruction #3 Execution | Instruction #2 Memory | lw Write-back



(binary) `lw $1, 5($2)`

The Pipelined Version of the Datapath

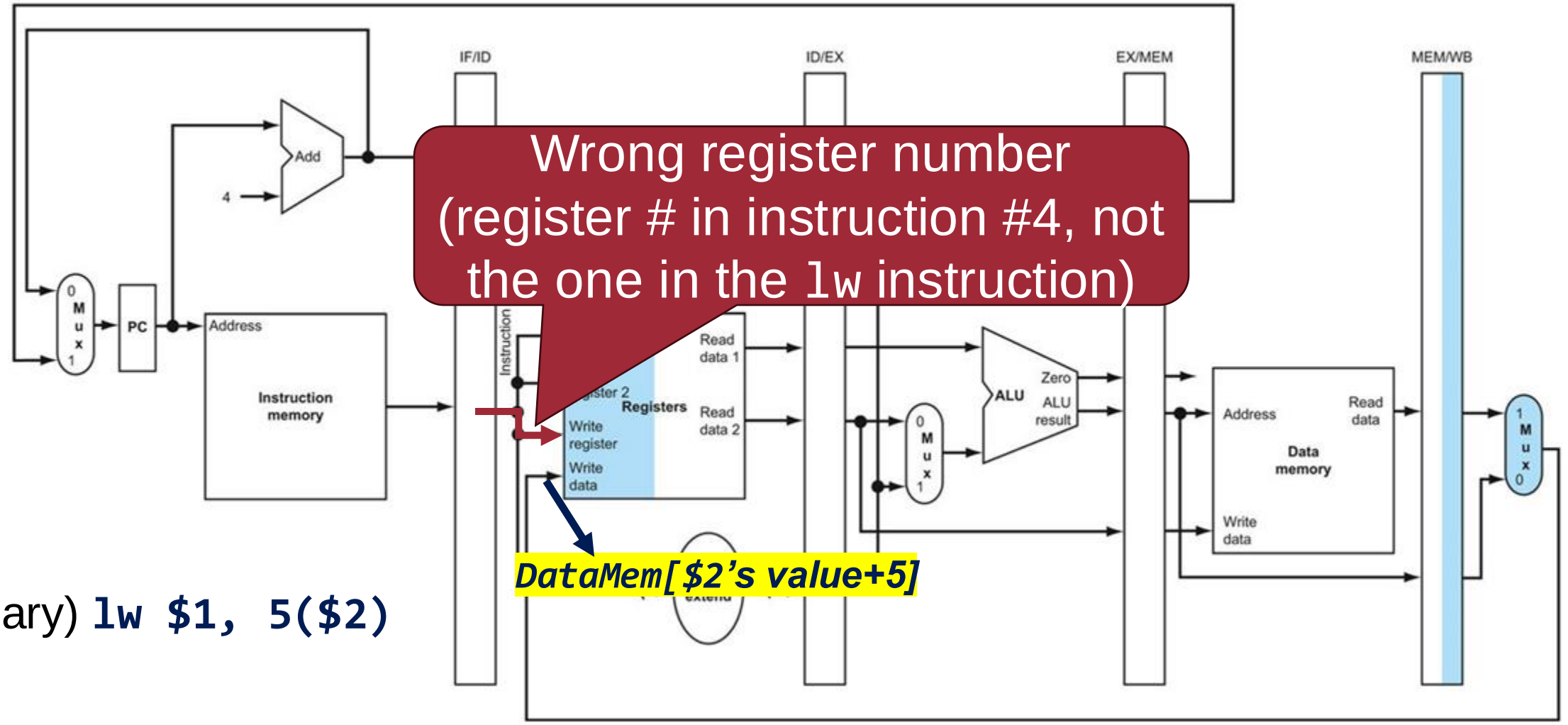
Instruction #5 Instruction fetch | Instruction #4 Instruction decode | Instruction #3 Execution | Instruction #2 Memory | lw Write-back



(binary) `lw $1, 5($2)`

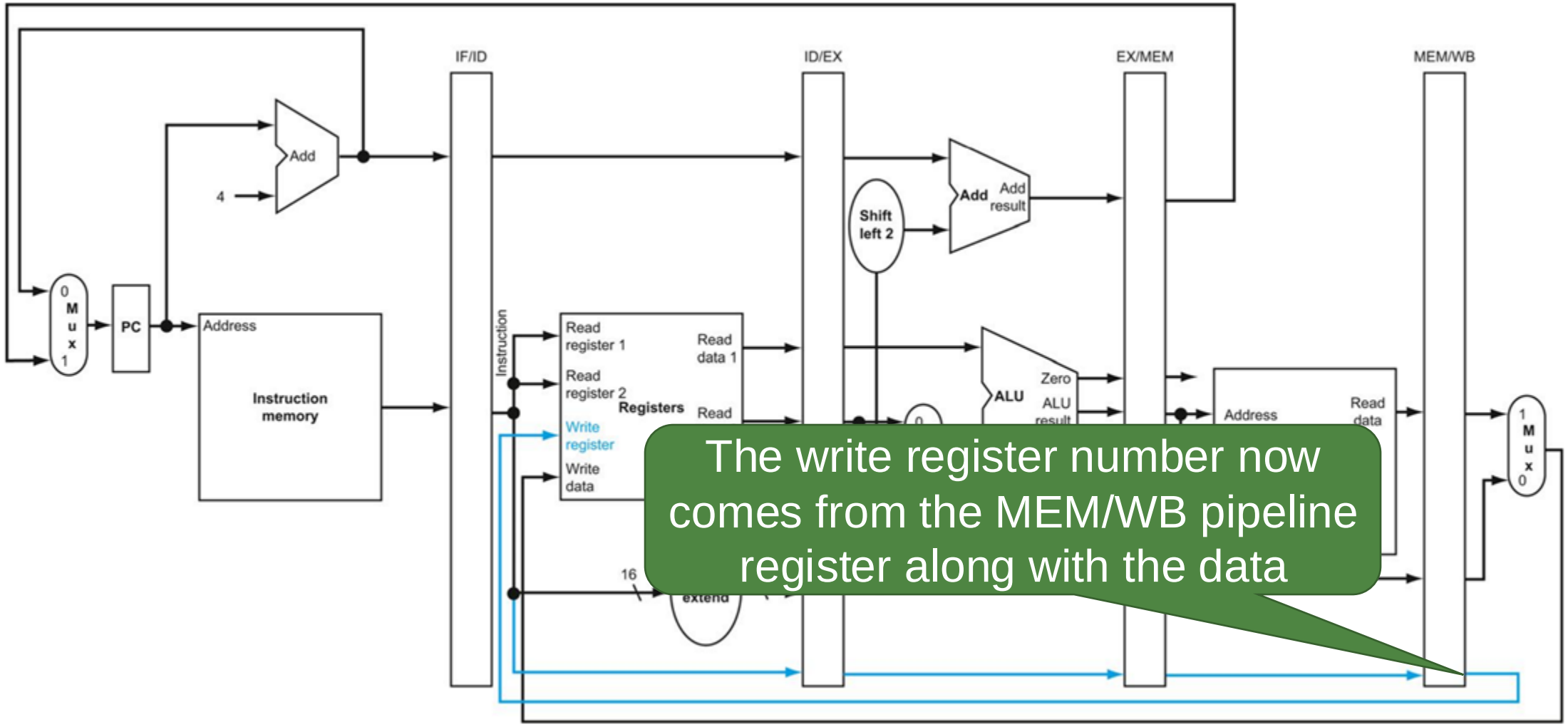
The Pipelined Version of the Datapath

Instruction #5 Instruction fetch | Instruction #4 Instruction decode | Instruction #3 Execution | Instruction #2 Memory | lw Write-back



(binary) `lw $1, 5($2)`

Solution: Corrected Datapath for Load



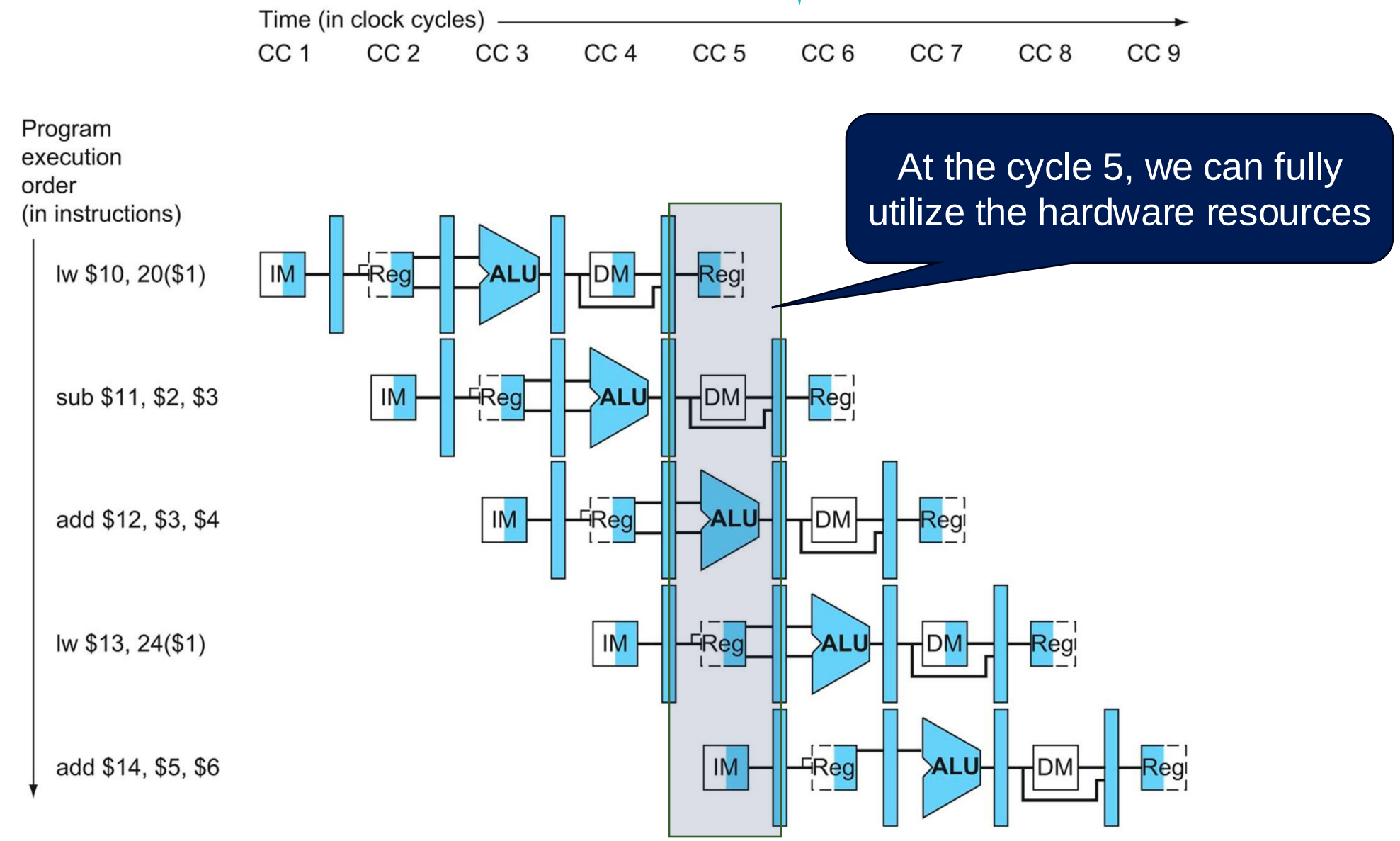
Pipelining Example


```
lw  $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw  $13, 24($1)
add $14, $5, $6
```

***Assumption*: there is no hazard**

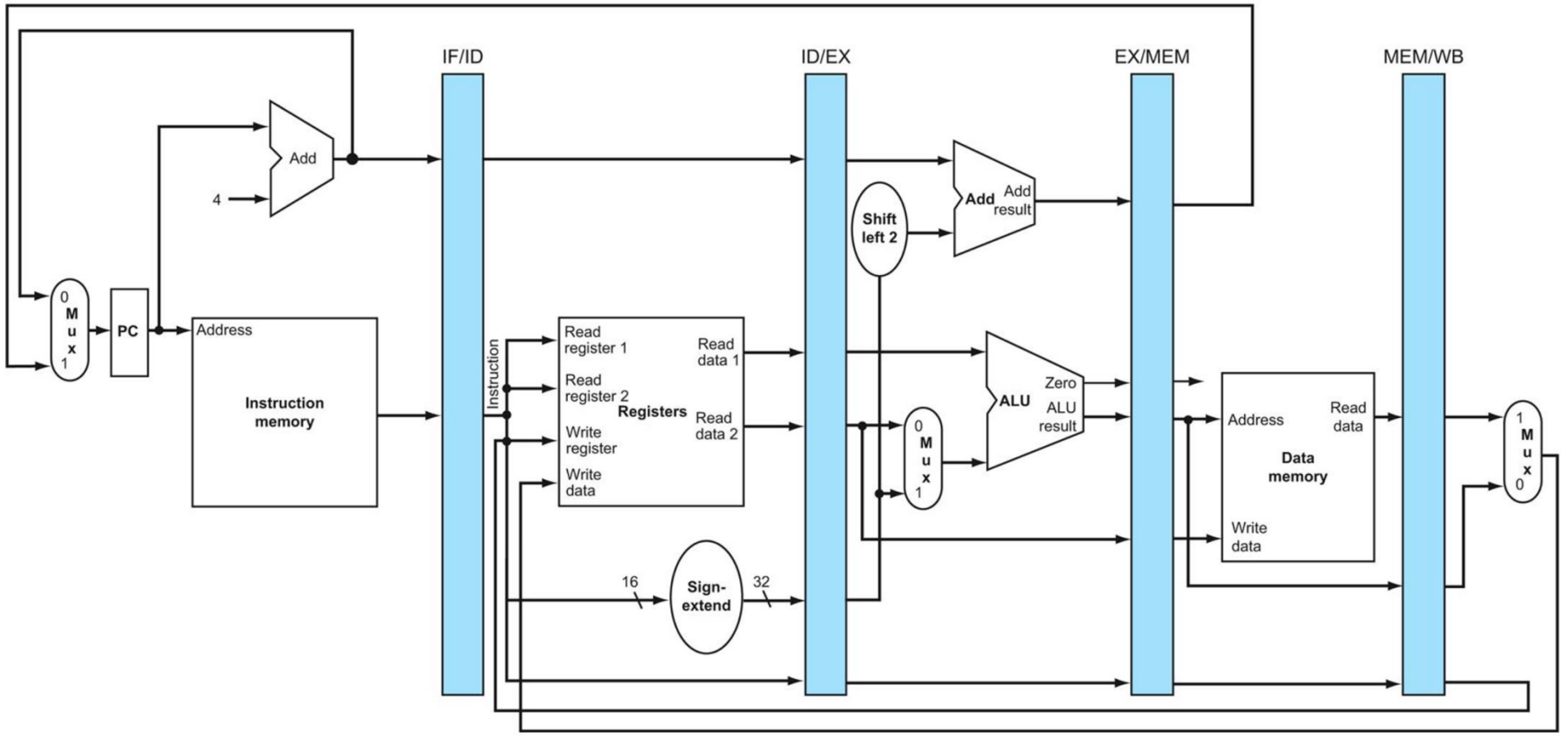
Later, we will cover about hazards!

Pipeline Diagram of Five Instructions



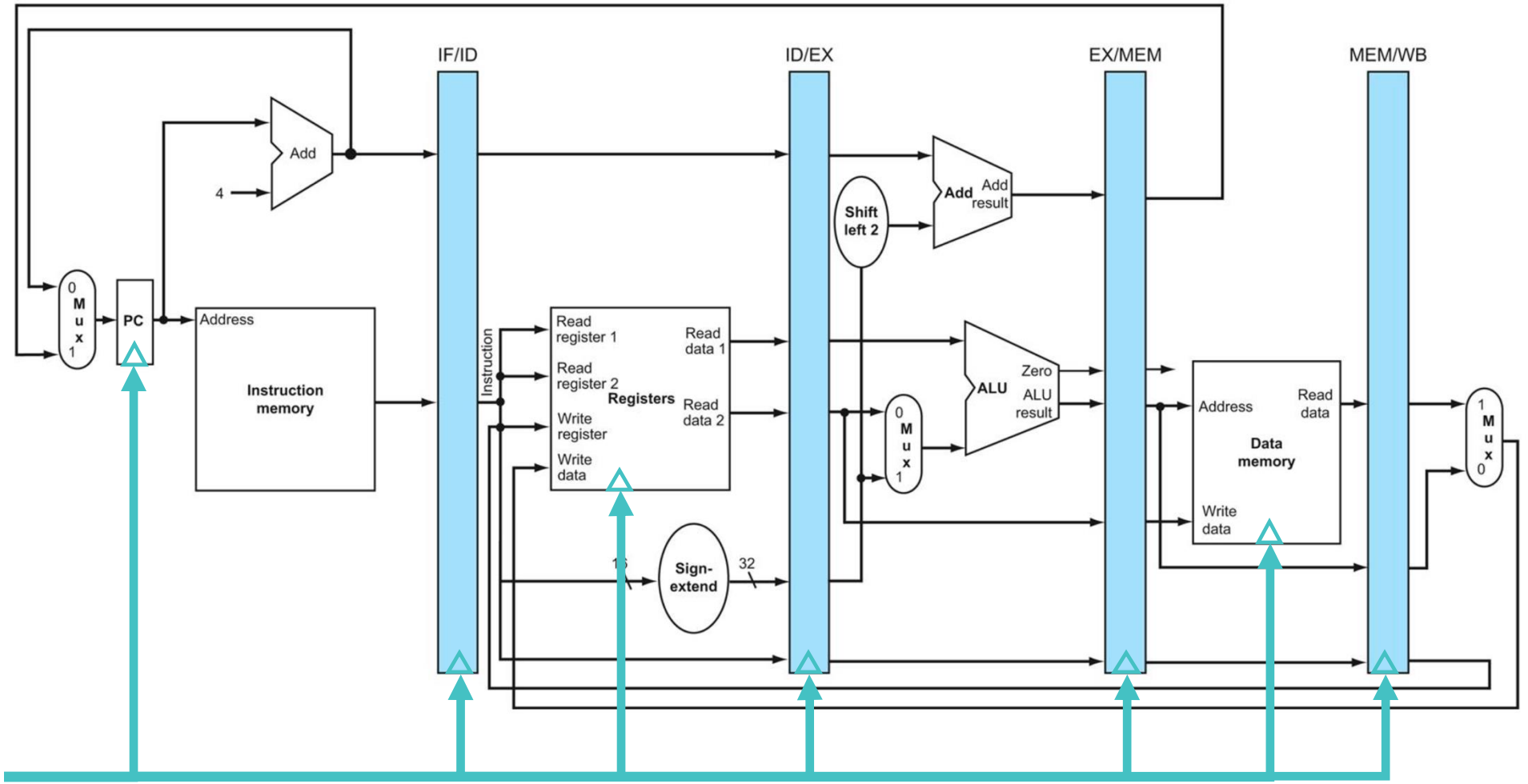
Clock Cycle 5 of the Pipeline

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



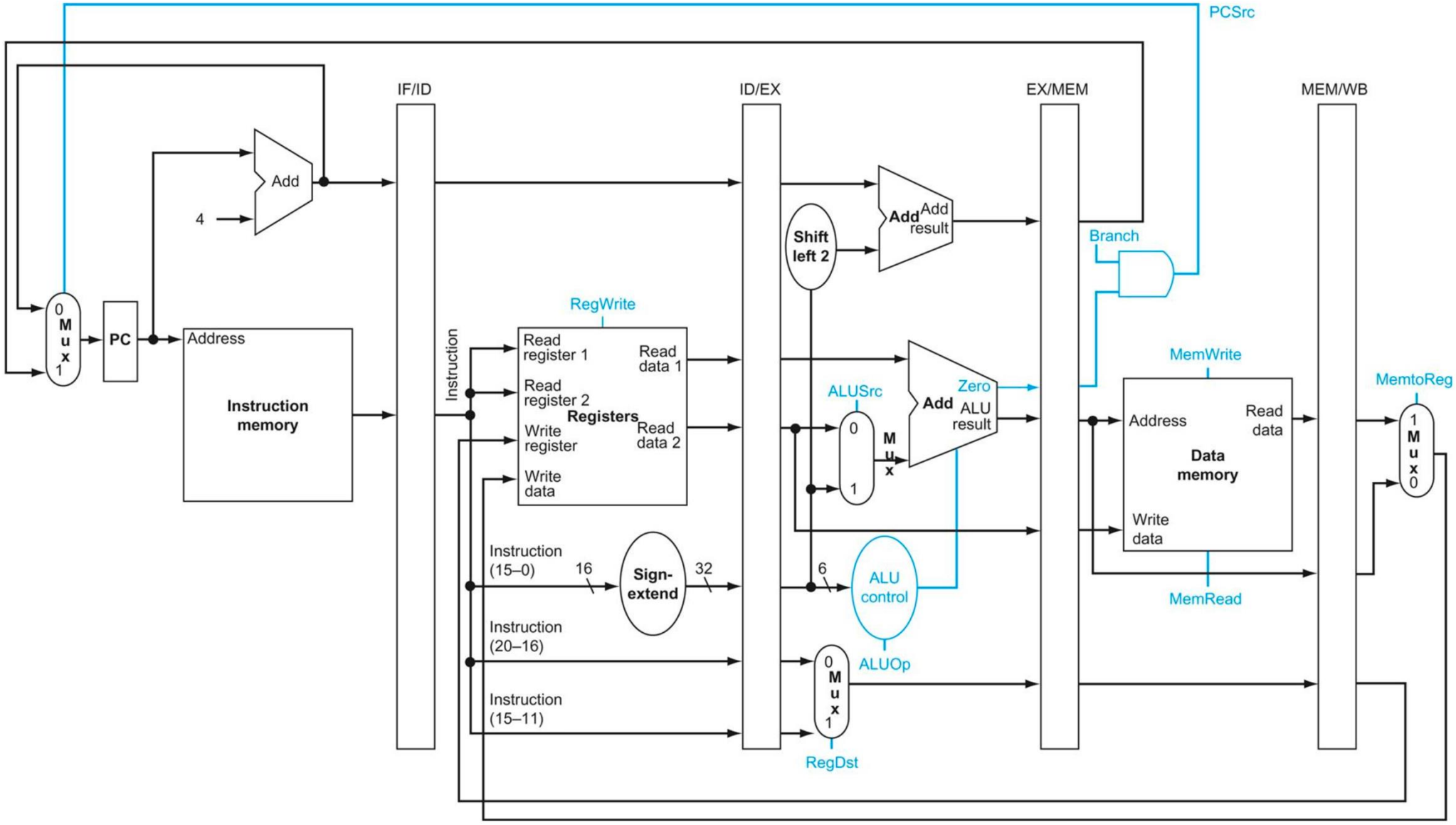
Clock Cycle 5 of the Pipeline

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

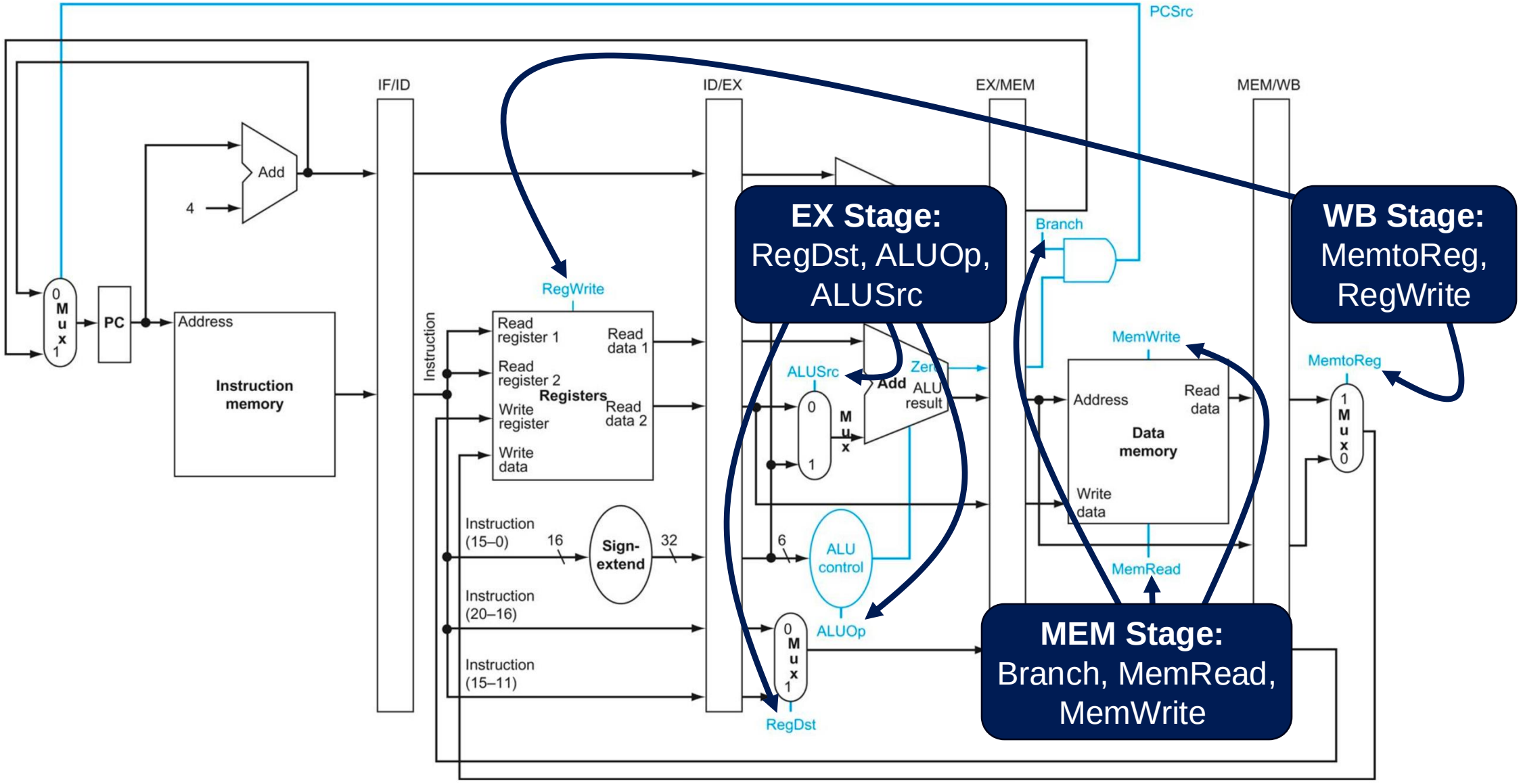


Pipelined Control

Pipeline Control Signals



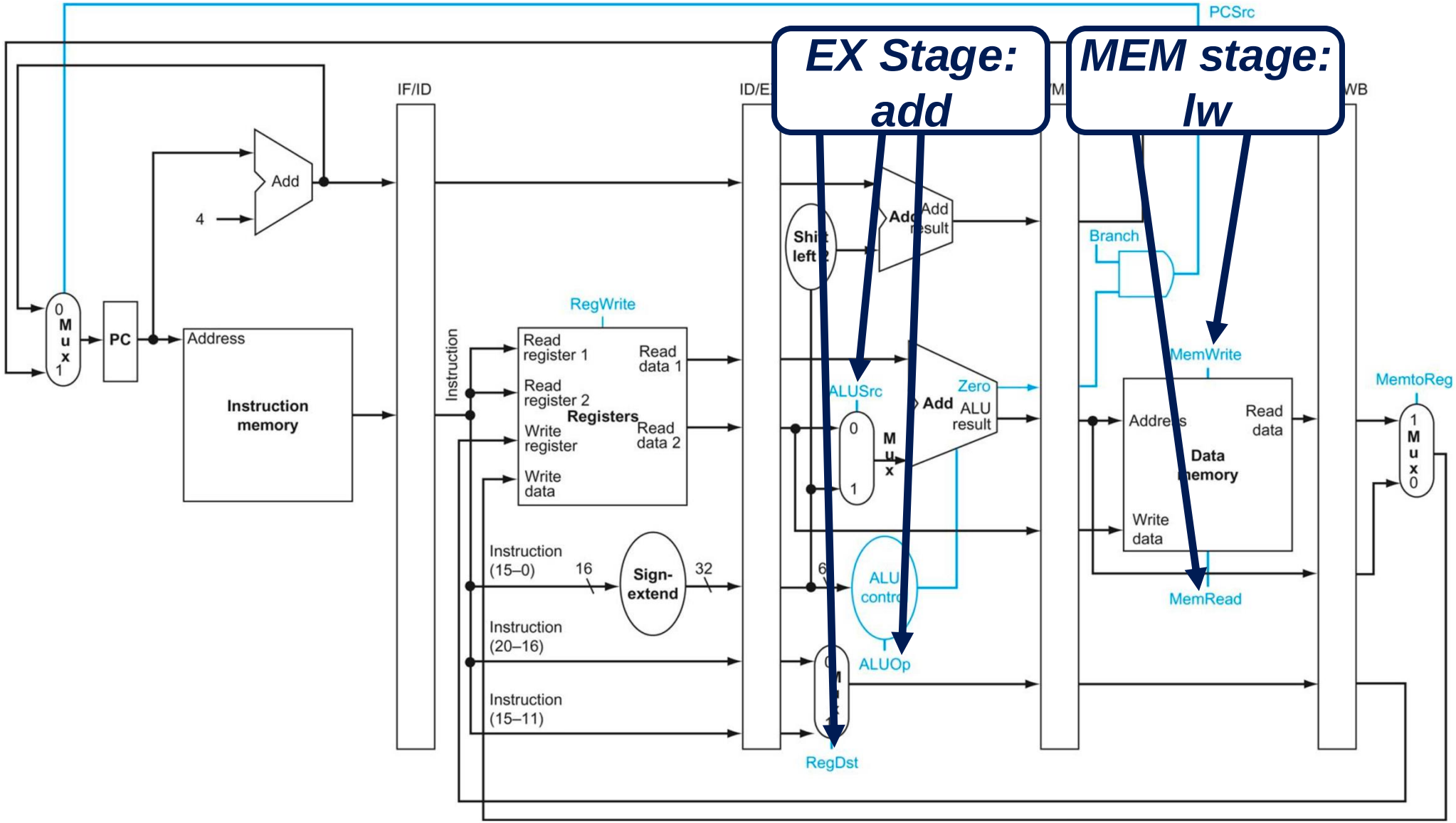
Pipeline Control Signals



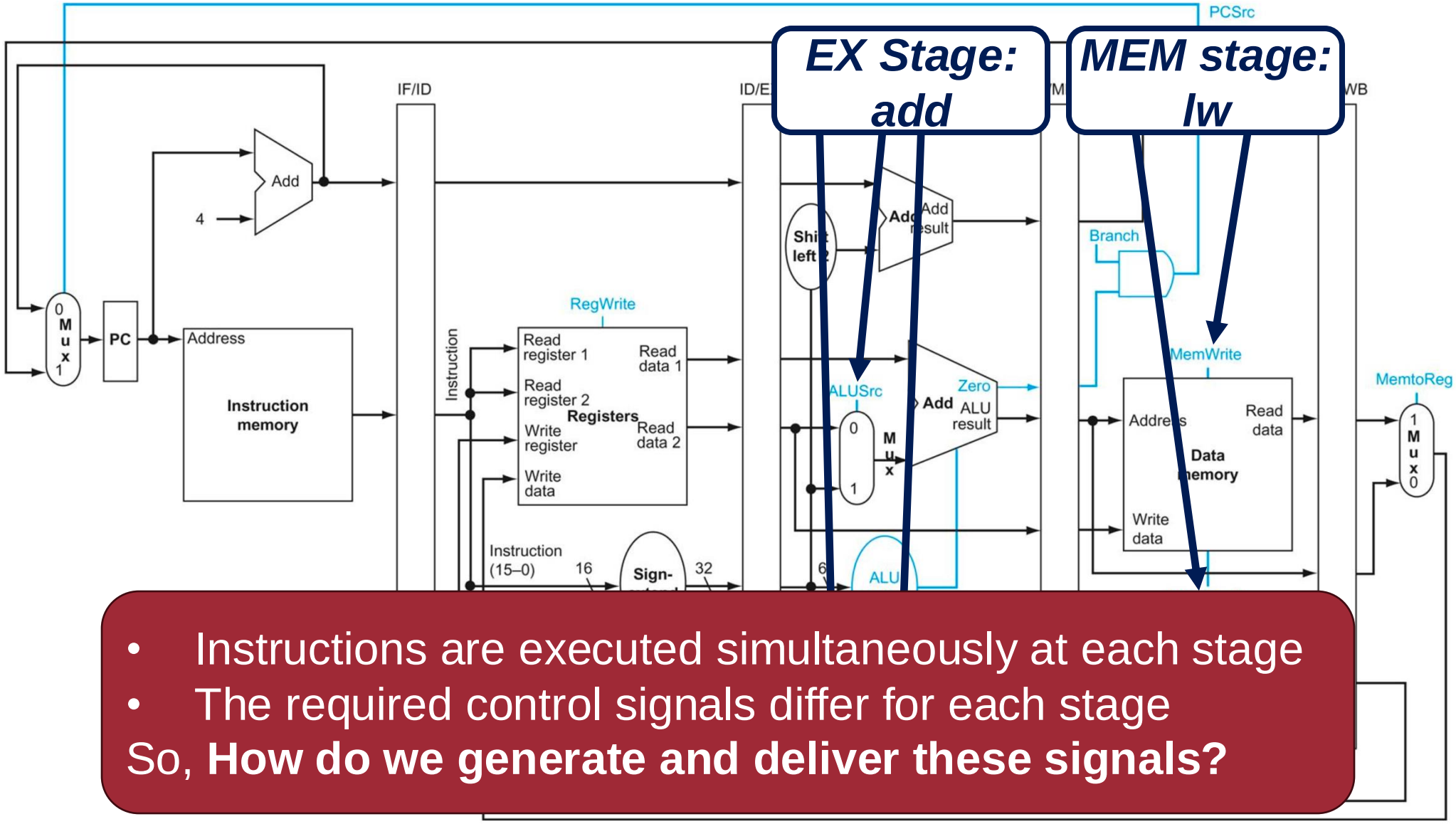
Pipeline Control Signals

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Motivation: Pipelined Control



Motivation: Pipelined Control



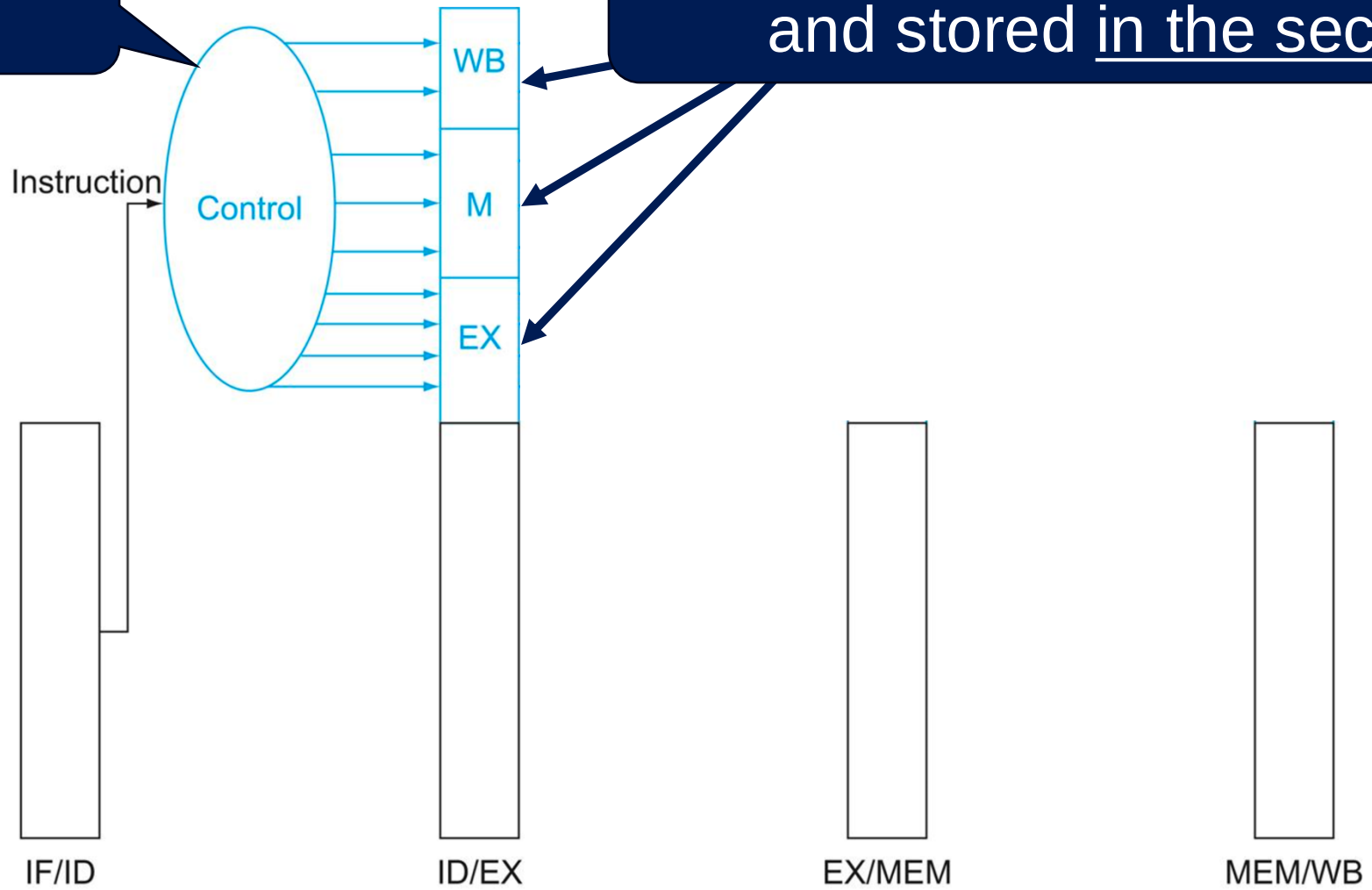
- Instructions are executed simultaneously at each stage
- The required control signals differ for each stage

So, **How do we generate and deliver these signals?**

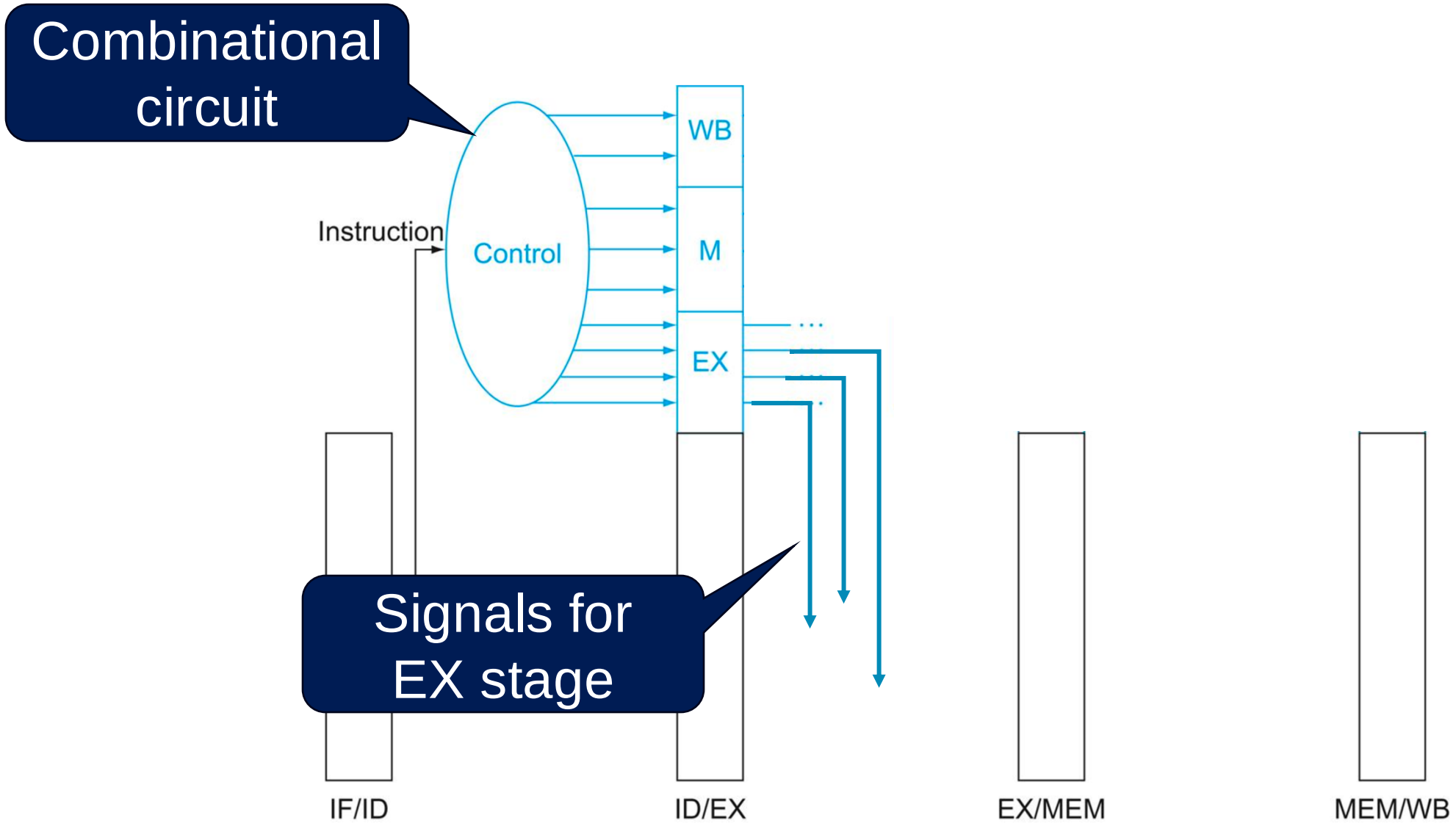
Idea: Pipelined Control

Combinational circuit

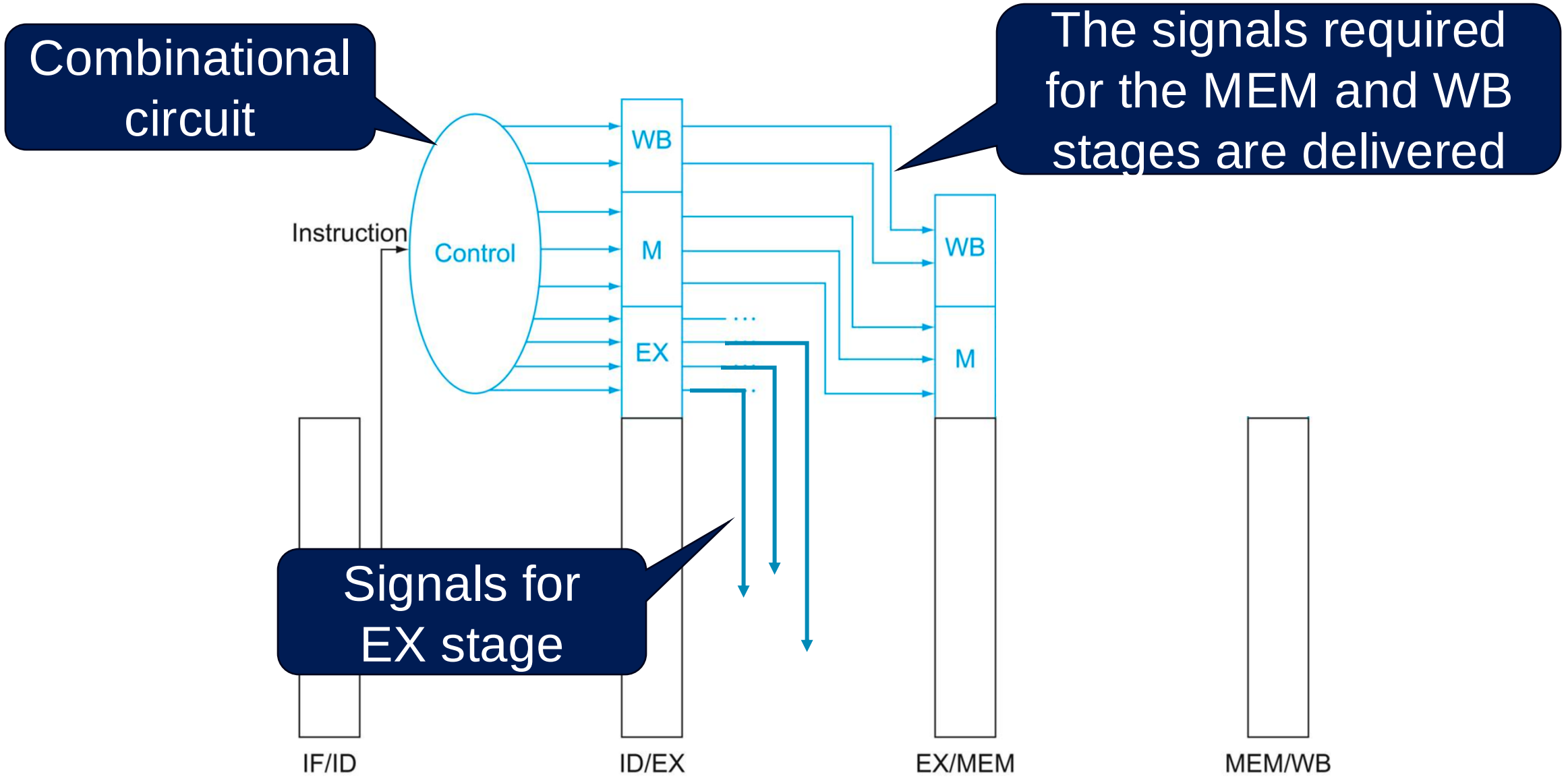
The signals for all stages are generated and stored in the second stage



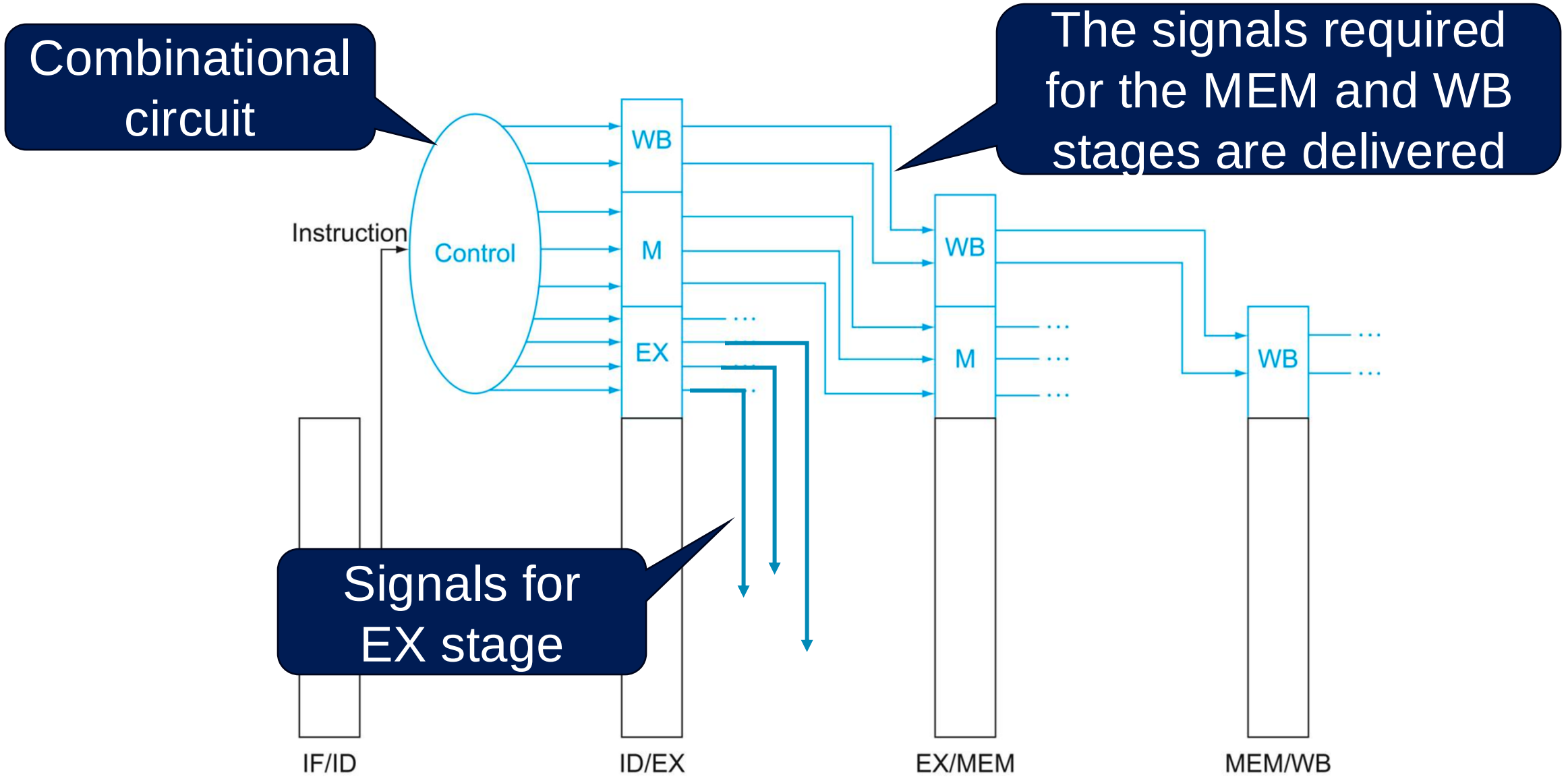
Idea: Pipelined Control



Idea: Pipelined Control



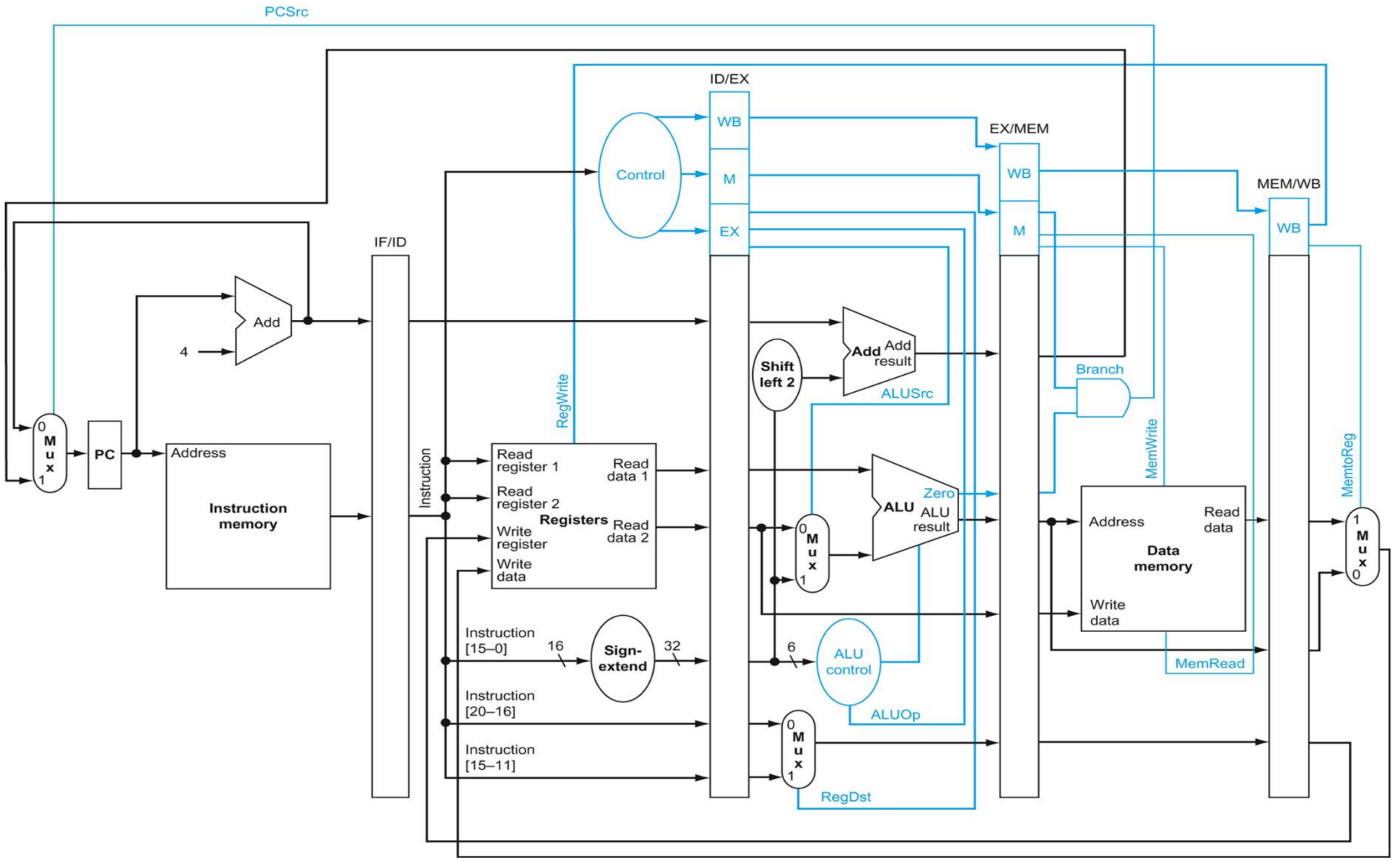
Idea: Pipelined Control



Pipelined Control Signals

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

The Pipelined Datapath with the Control



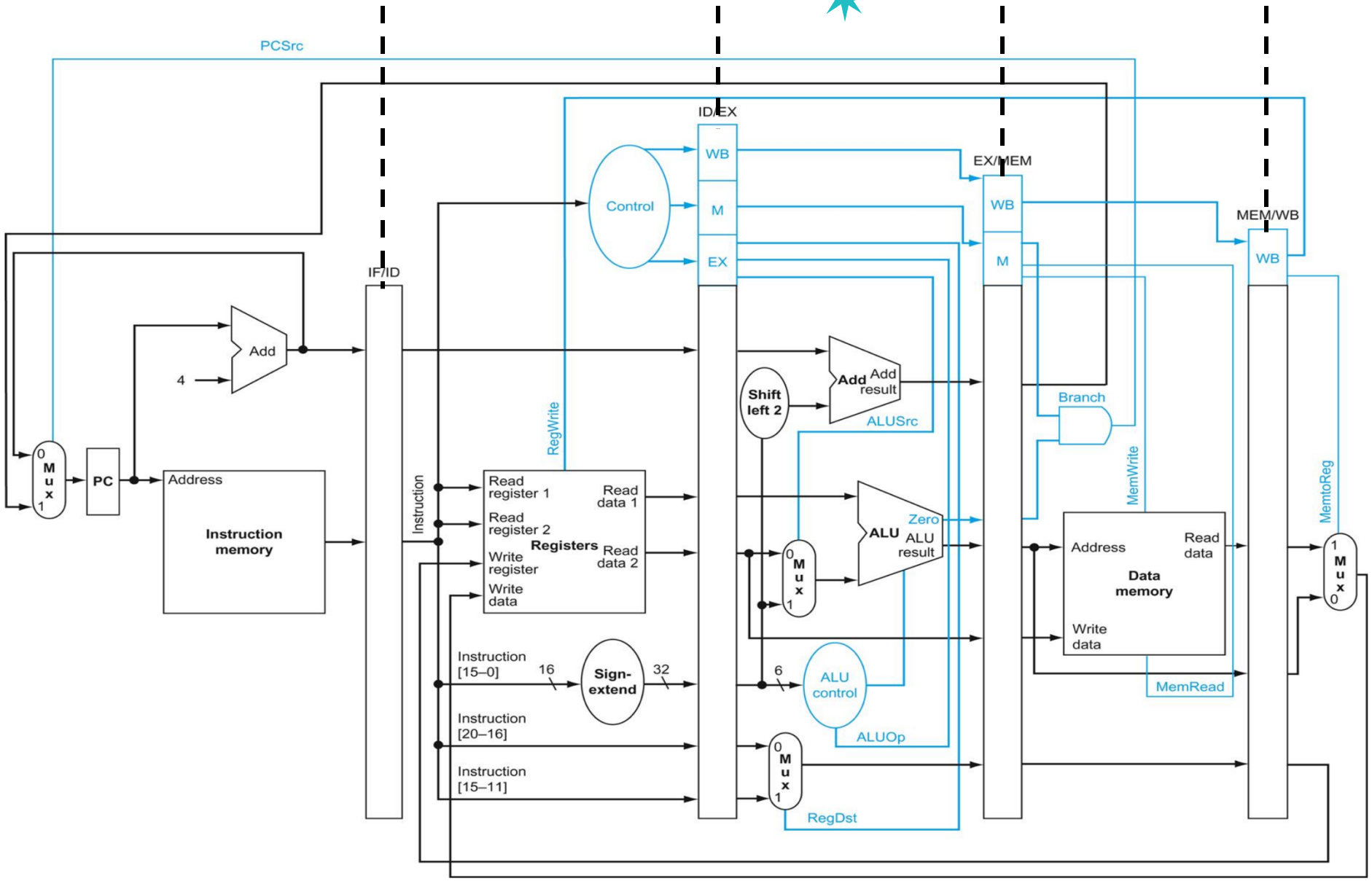
Pipelining Example


```
lw  $10, 20($1)
sub $11, $2, $3
add $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

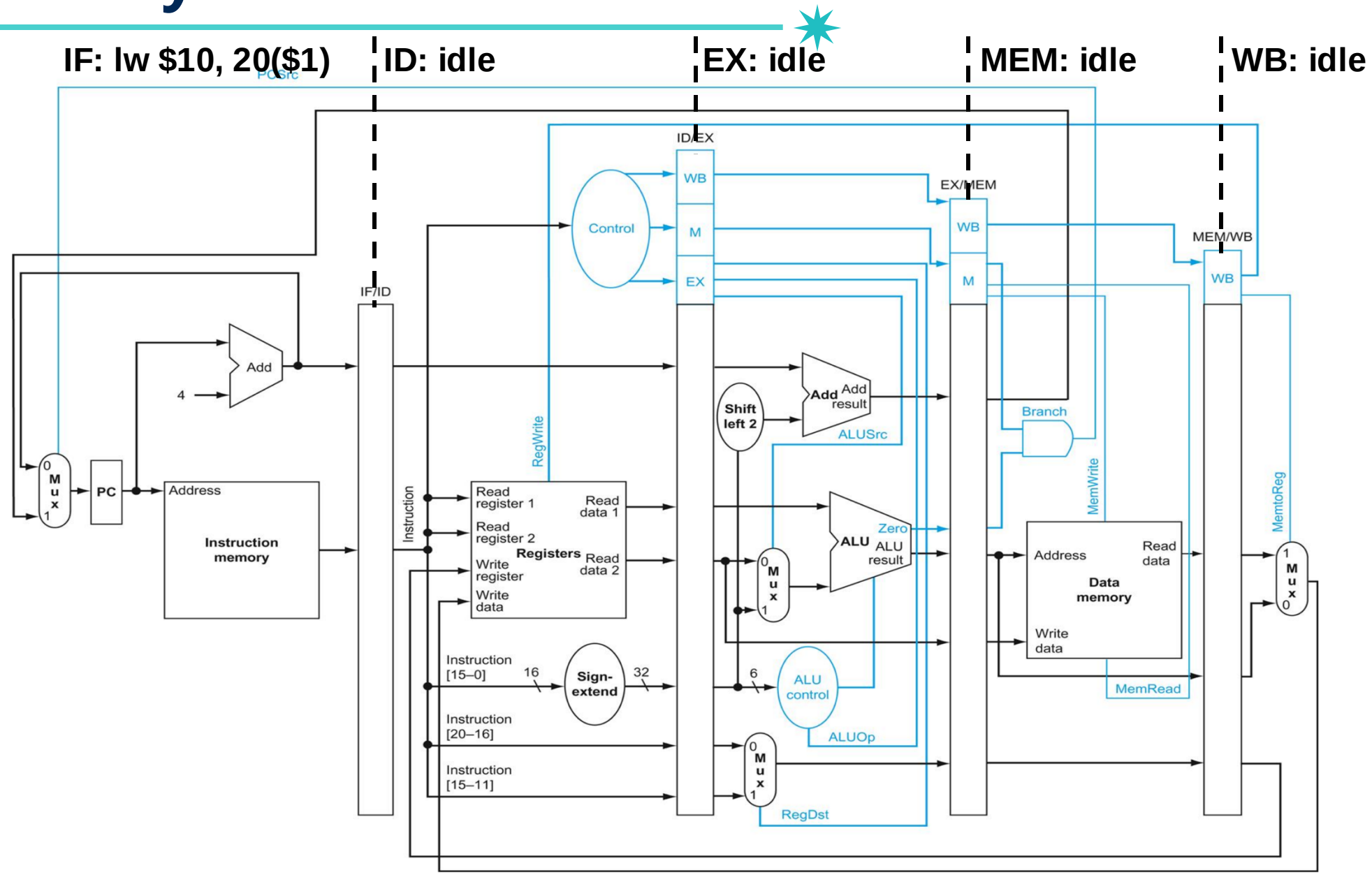
***Assumption*: there is no hazard**

Later, we will cover about hazards!

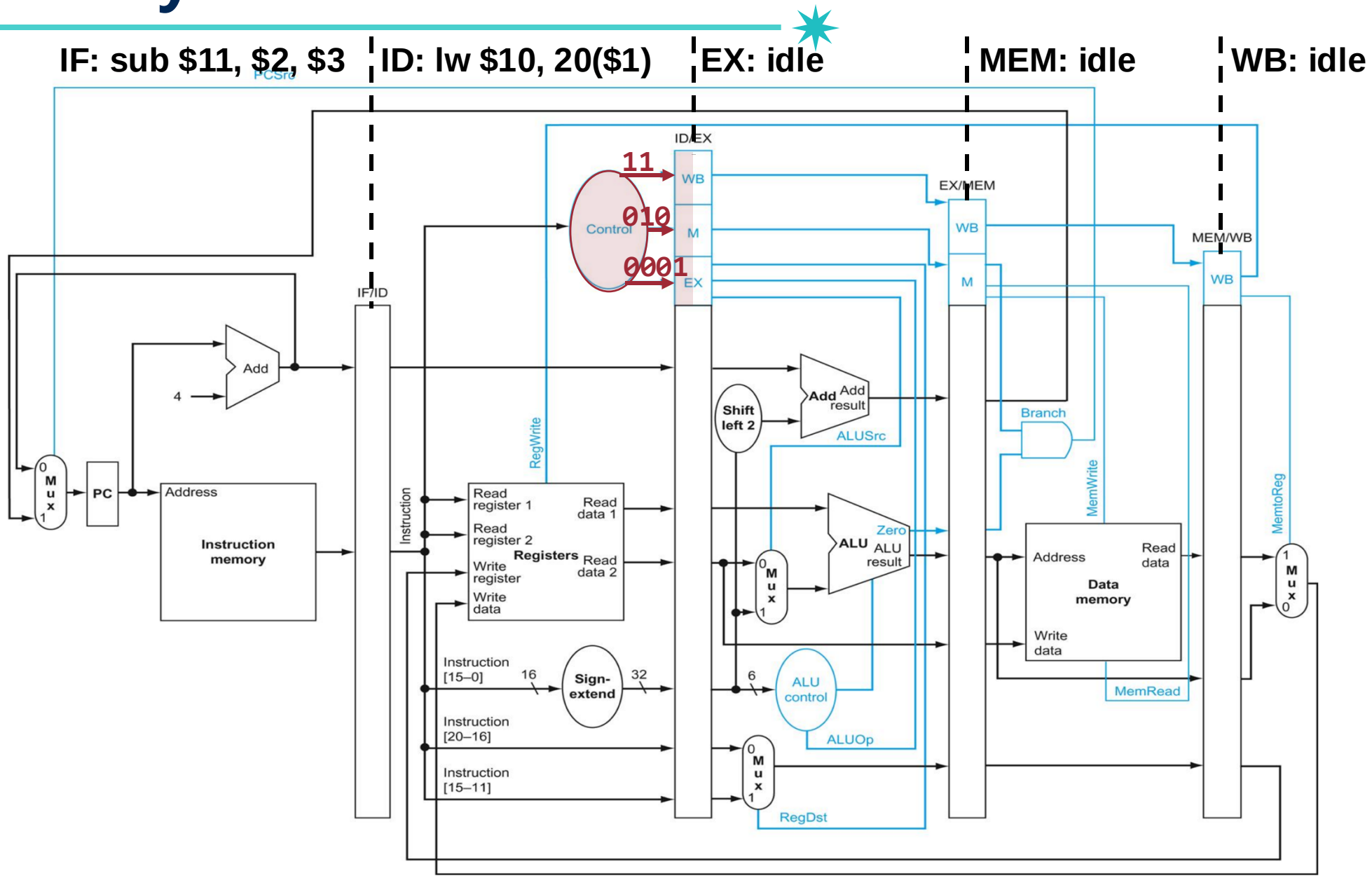
Example: Pipelined Datapath with the Control



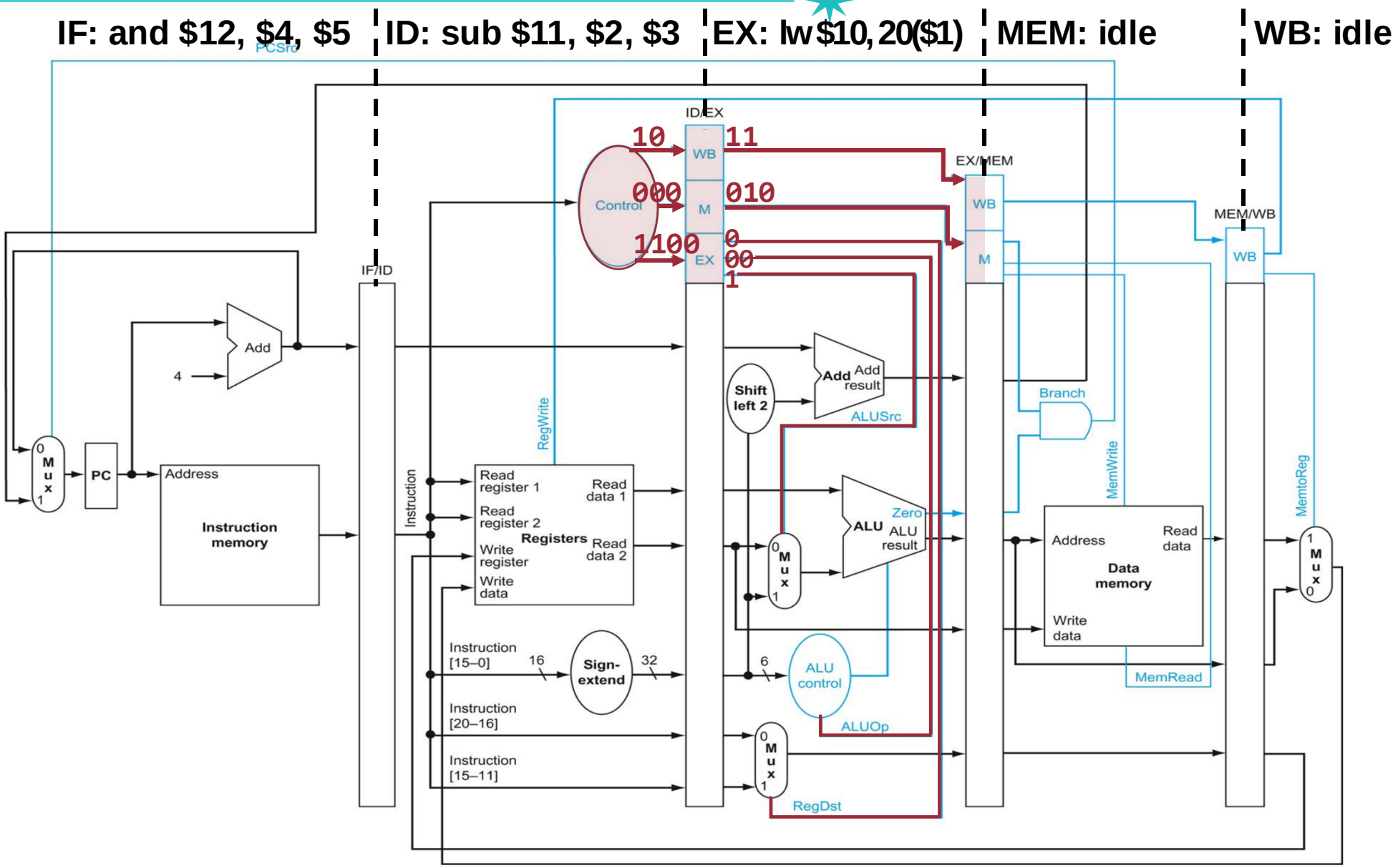
Clock Cycle #1



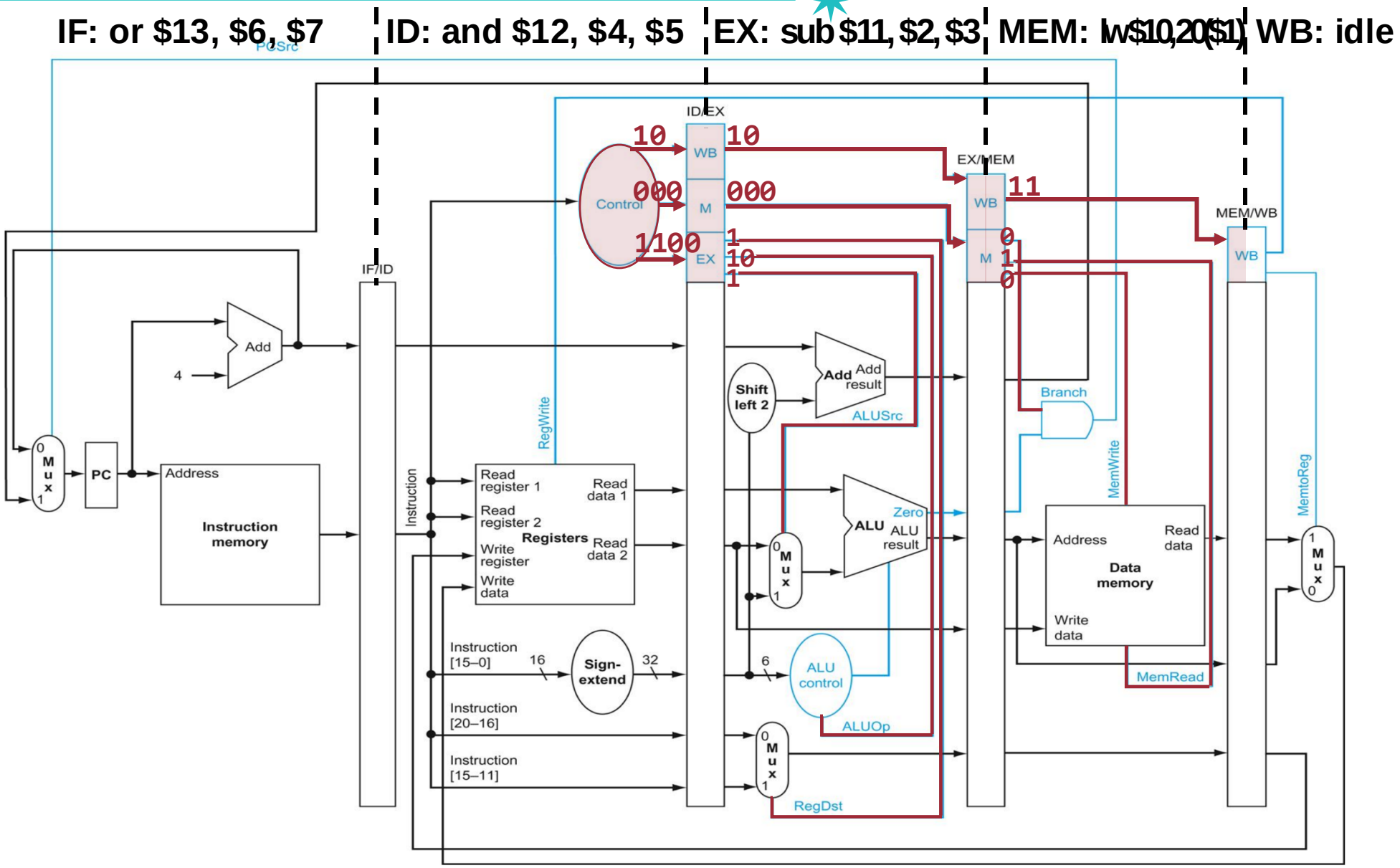
Clock Cycle #2



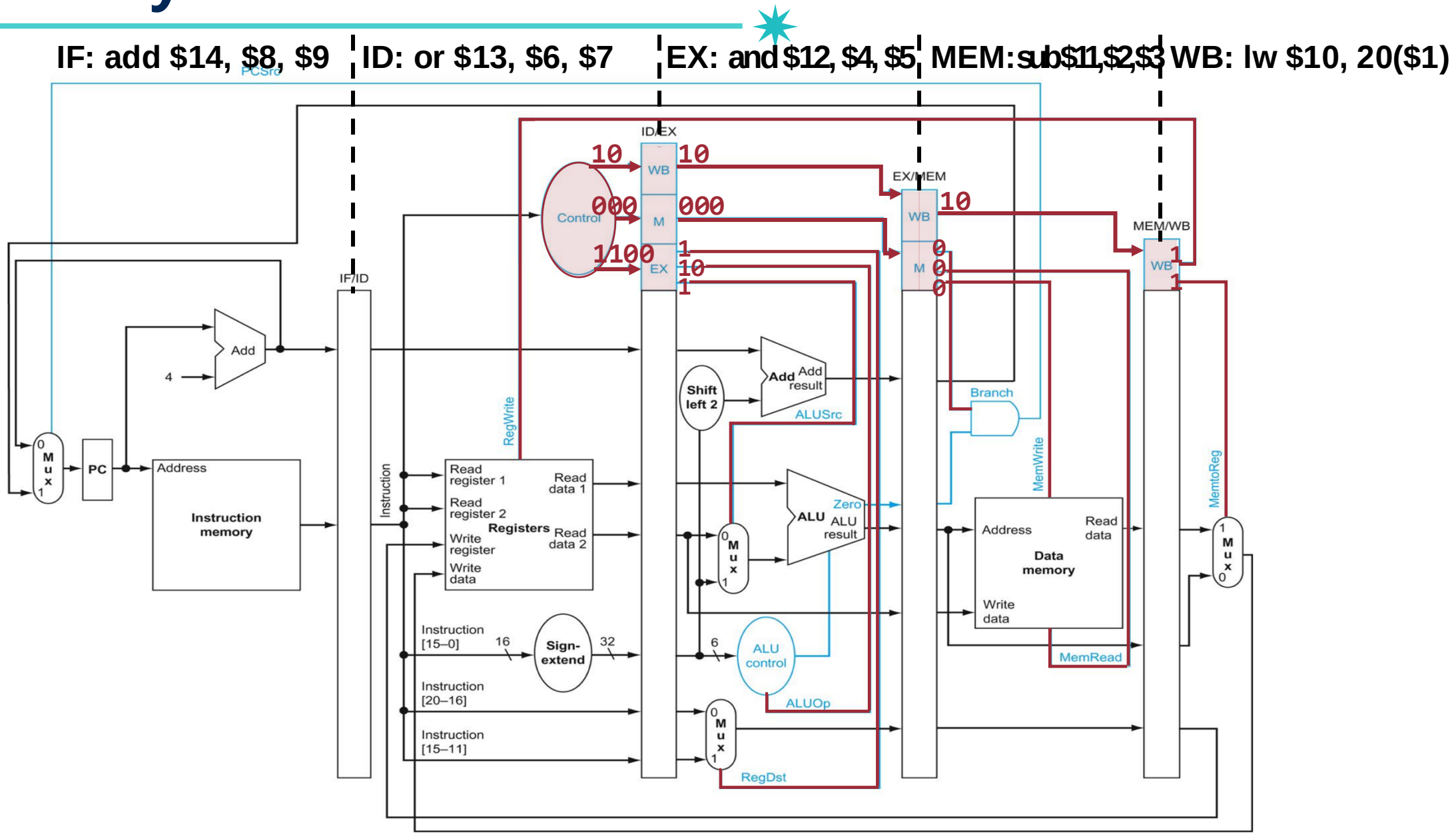
Clock Cycle #3



Clock Cycle #4



Clock Cycle #5



Question?