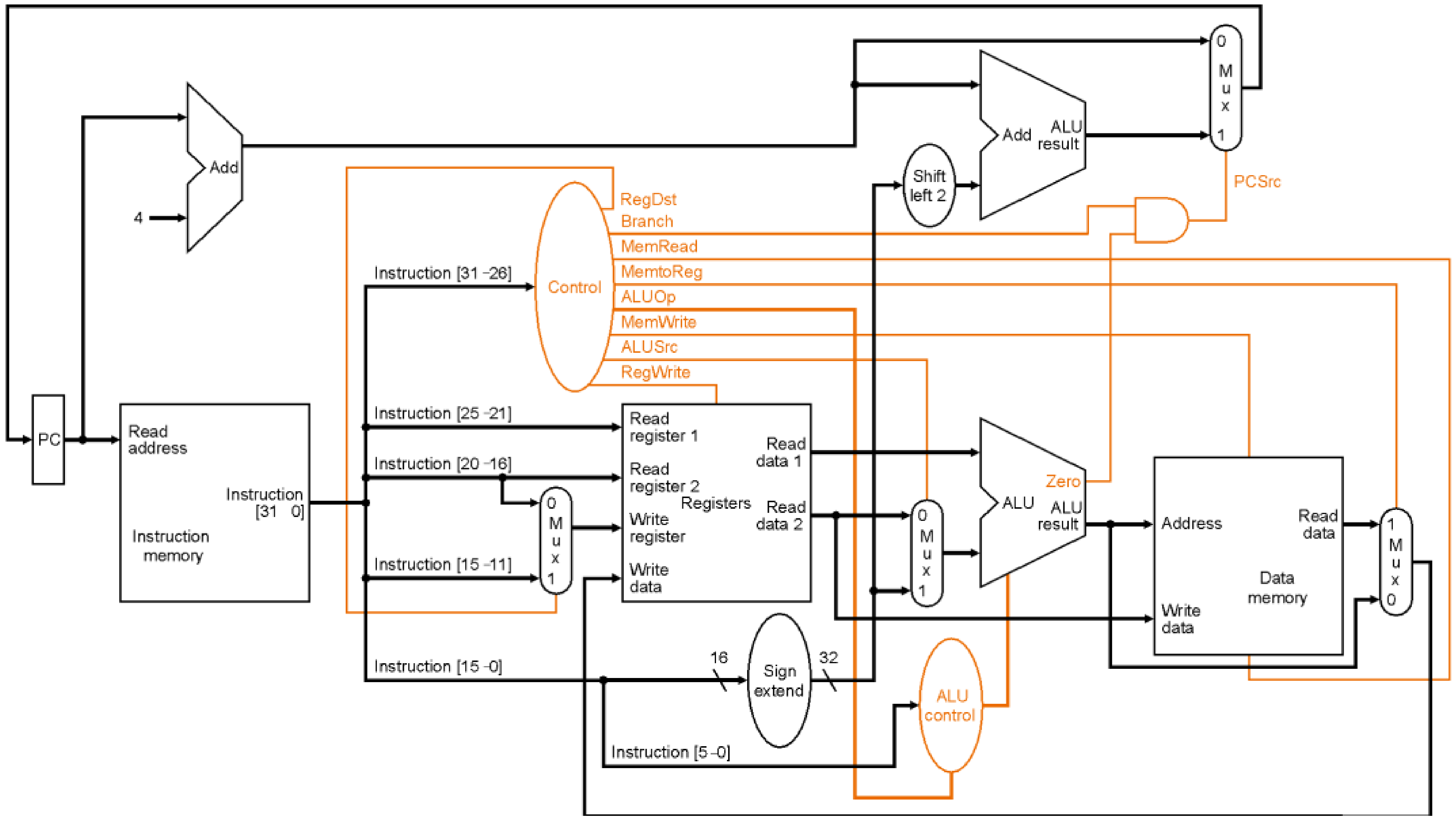
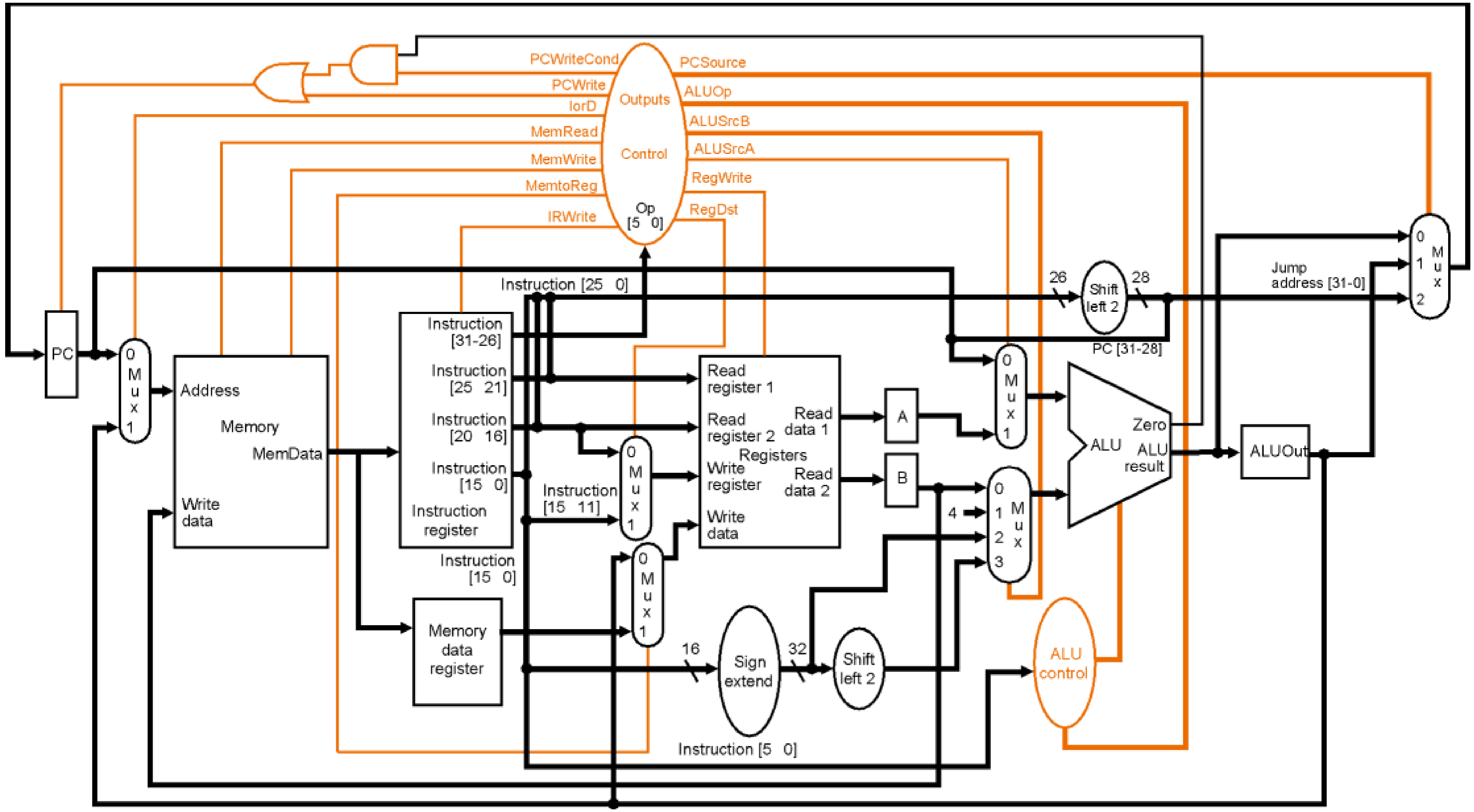
Pipelined CPU

Jason Mars

Evolution of Our CPU: Single Cycle



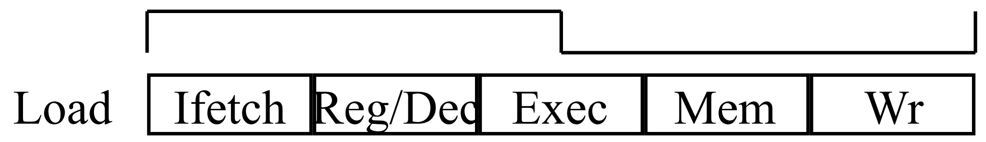
Evolution of Our CPU: Multi Cycle



Instruction Latencies and Throughput

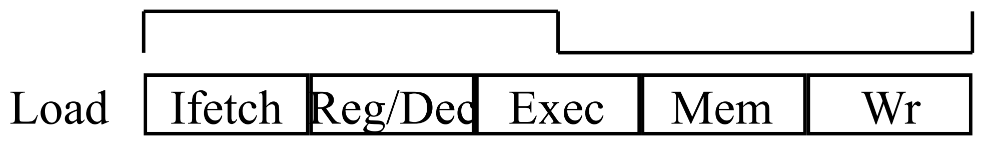
Instruction Latencies and Throughput

•Single-Cycle CPU

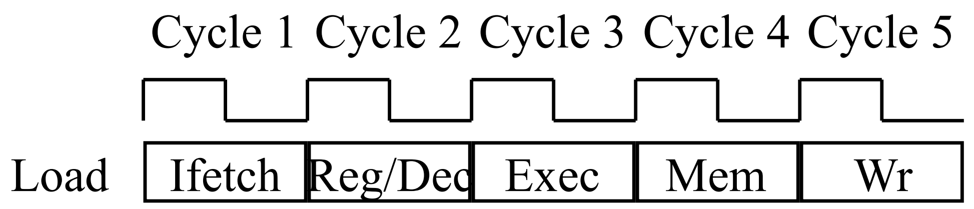


Instruction Latencies and Throughput

•Single-Cycle CPU

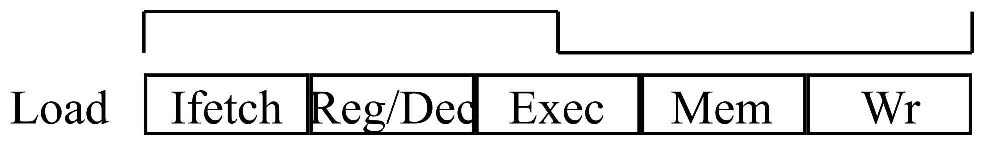


•Multiple Cycle CPU

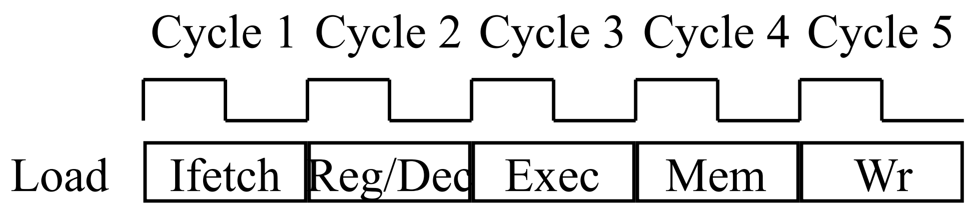


Instruction Latencies and Throughput

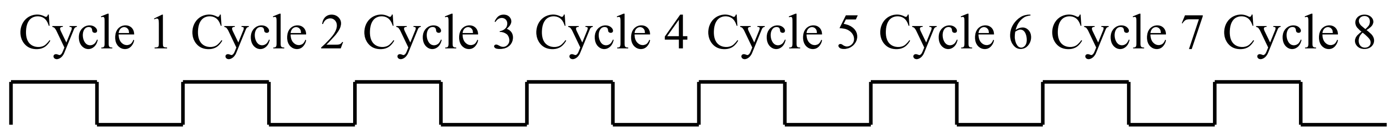
•Single-Cycle CPU



•Multiple Cycle CPU

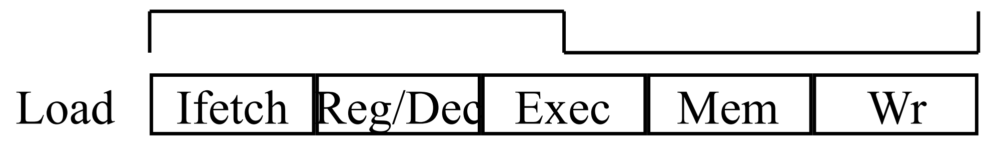


•Pipelined CPU

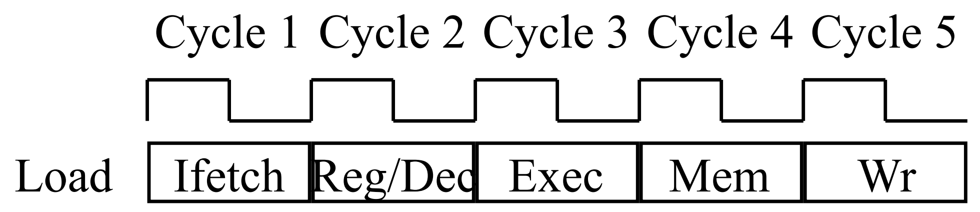


Instruction Latencies and Throughput

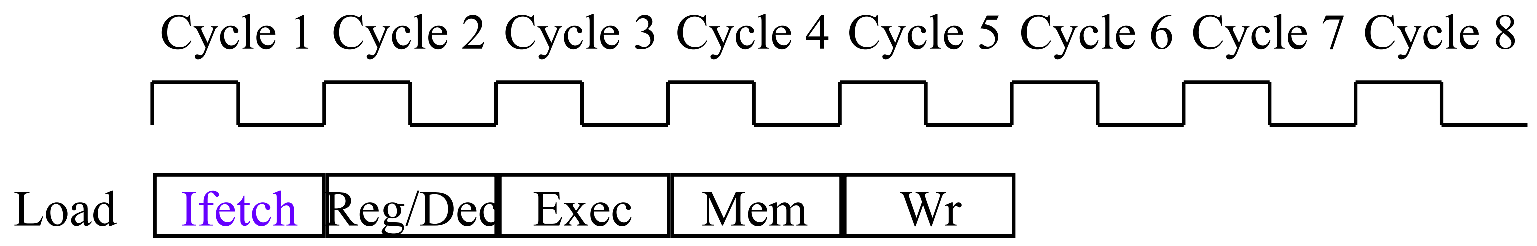
- **Single-Cycle CPU**



- **Multiple Cycle CPU**

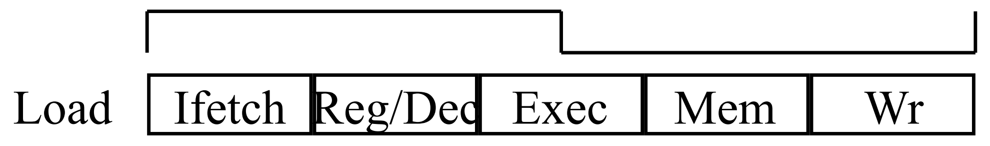


- **Pipelined CPU**

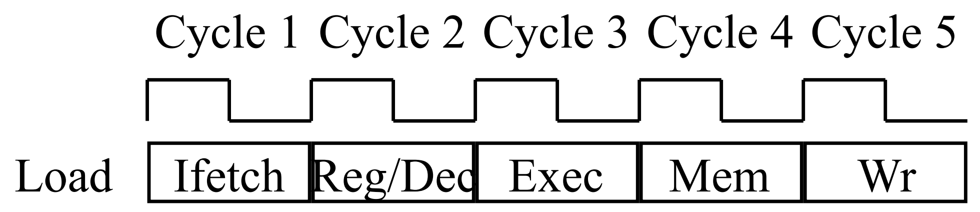


Instruction Latencies and Throughput

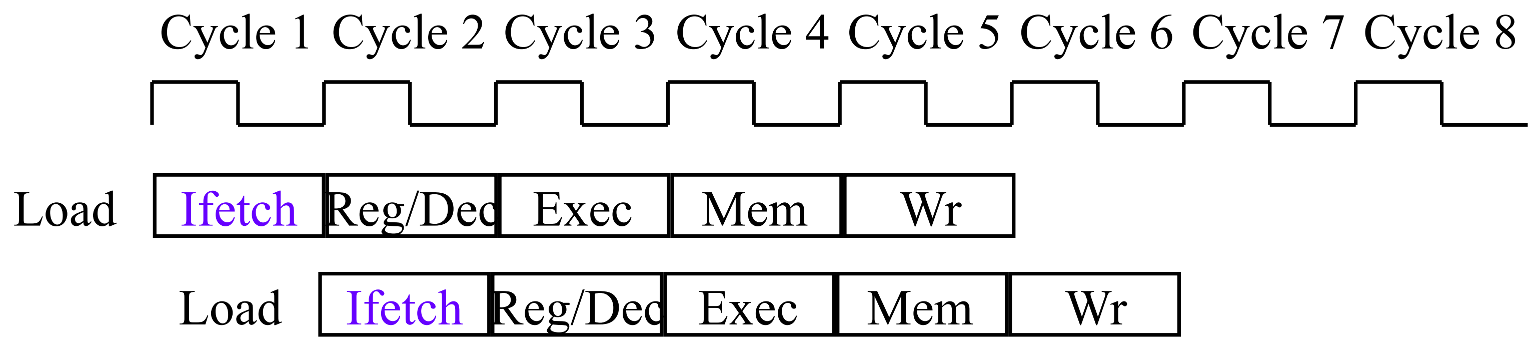
•Single-Cycle CPU



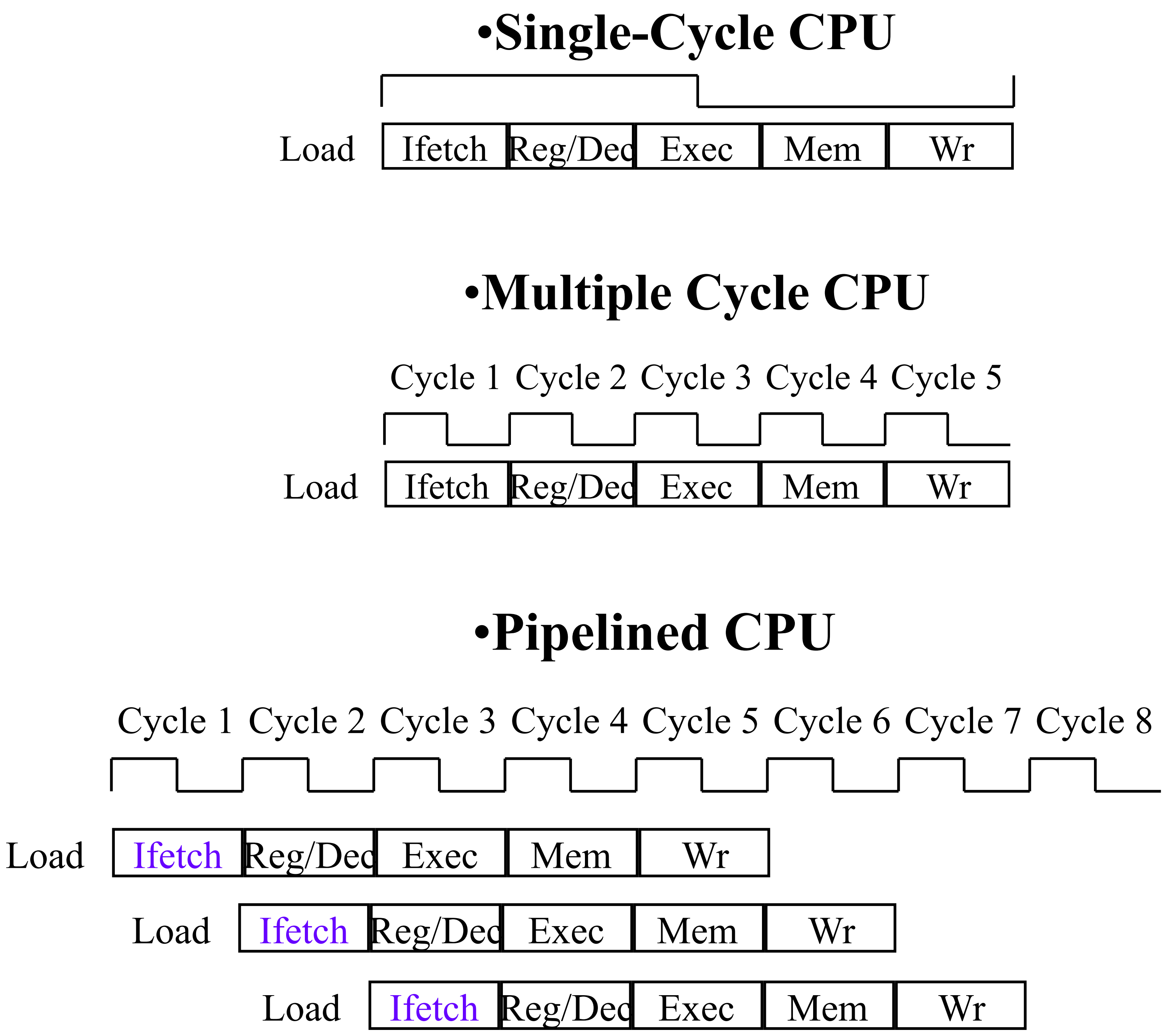
•Multiple Cycle CPU



•Pipelined CPU

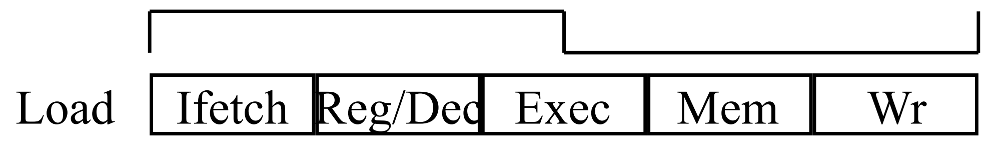


Instruction Latencies and Throughput

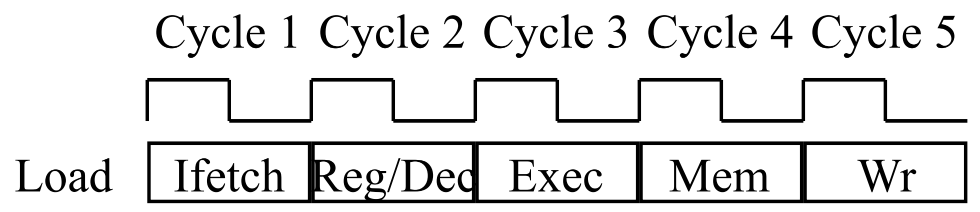


Instruction Latencies and Throughput

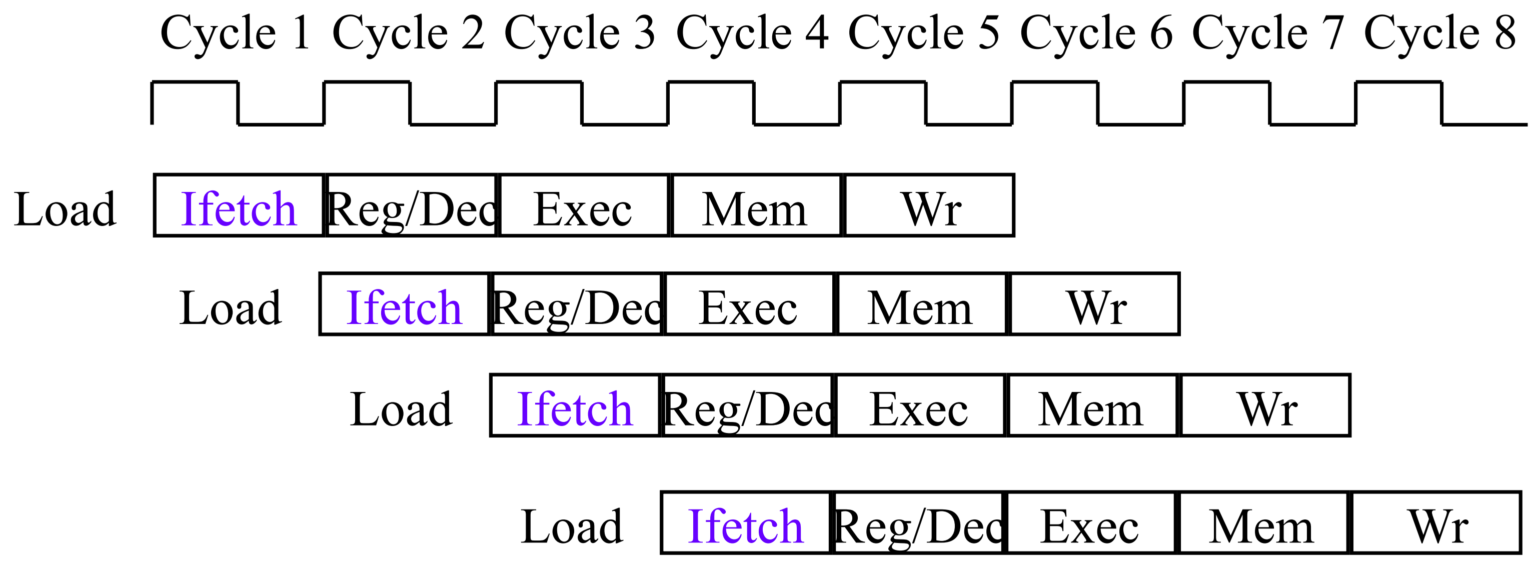
•Single-Cycle CPU



•Multiple Cycle CPU



•Pipelined CPU



Pipelining Advantages

Pipelining Advantages

- Higher maximum throughput

Pipelining Advantages

- Higher maximum throughput
- Higher utilization of CPU resources

Pipelining Advantages

- Higher maximum throughput
- Higher utilization of CPU resources

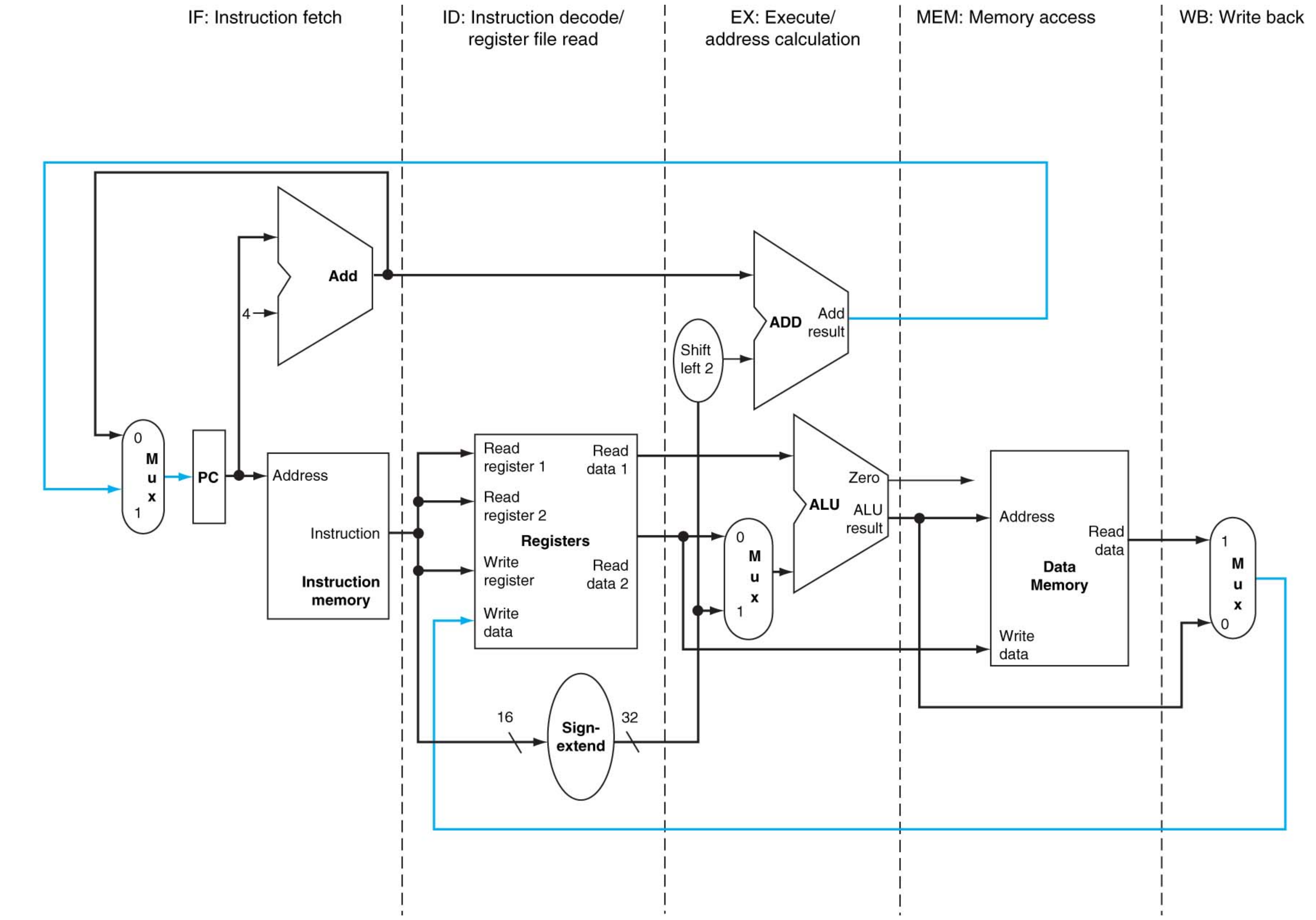
Pipelining Advantages

- Higher maximum throughput
- Higher utilization of CPU resources
- But, more complicated datapath, more complex control

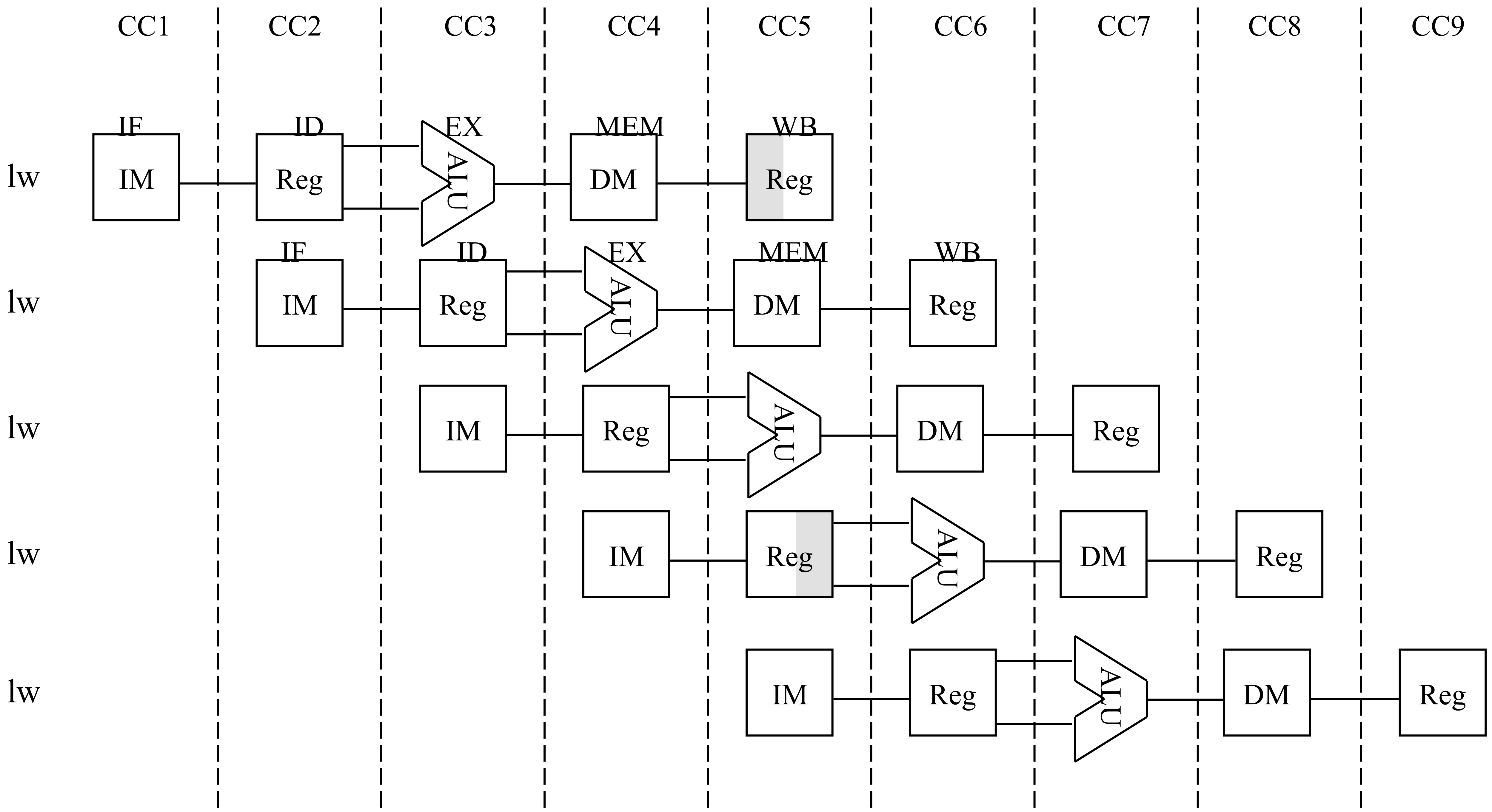
A Pipelined Datapath

- IF: Instruction fetch
- ID: Instruction decode and register fetch
- EX: Execution and effective address calculation
- MEM: Memory access
- WB: Write back

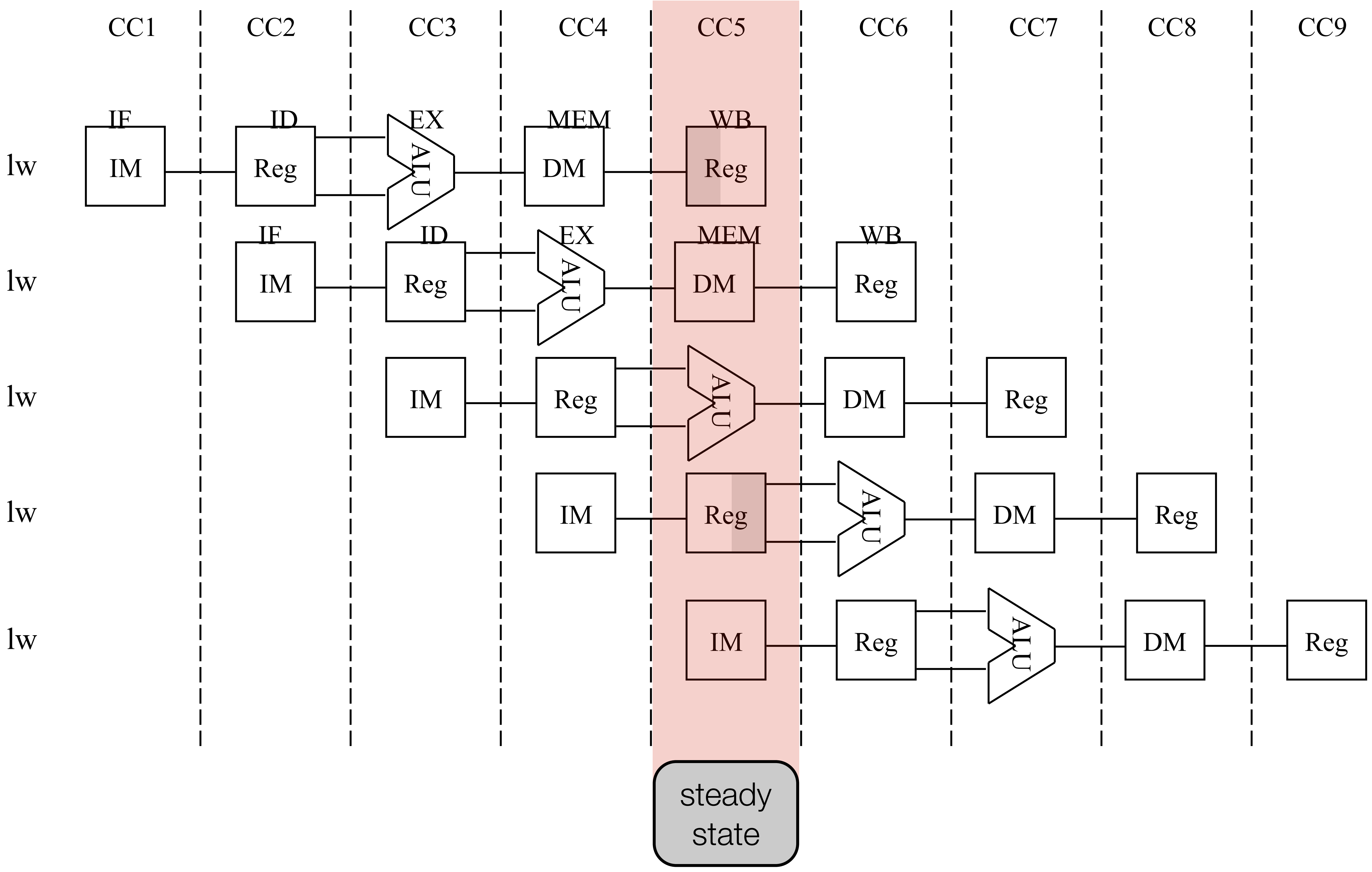
A Rough View of the Datapath



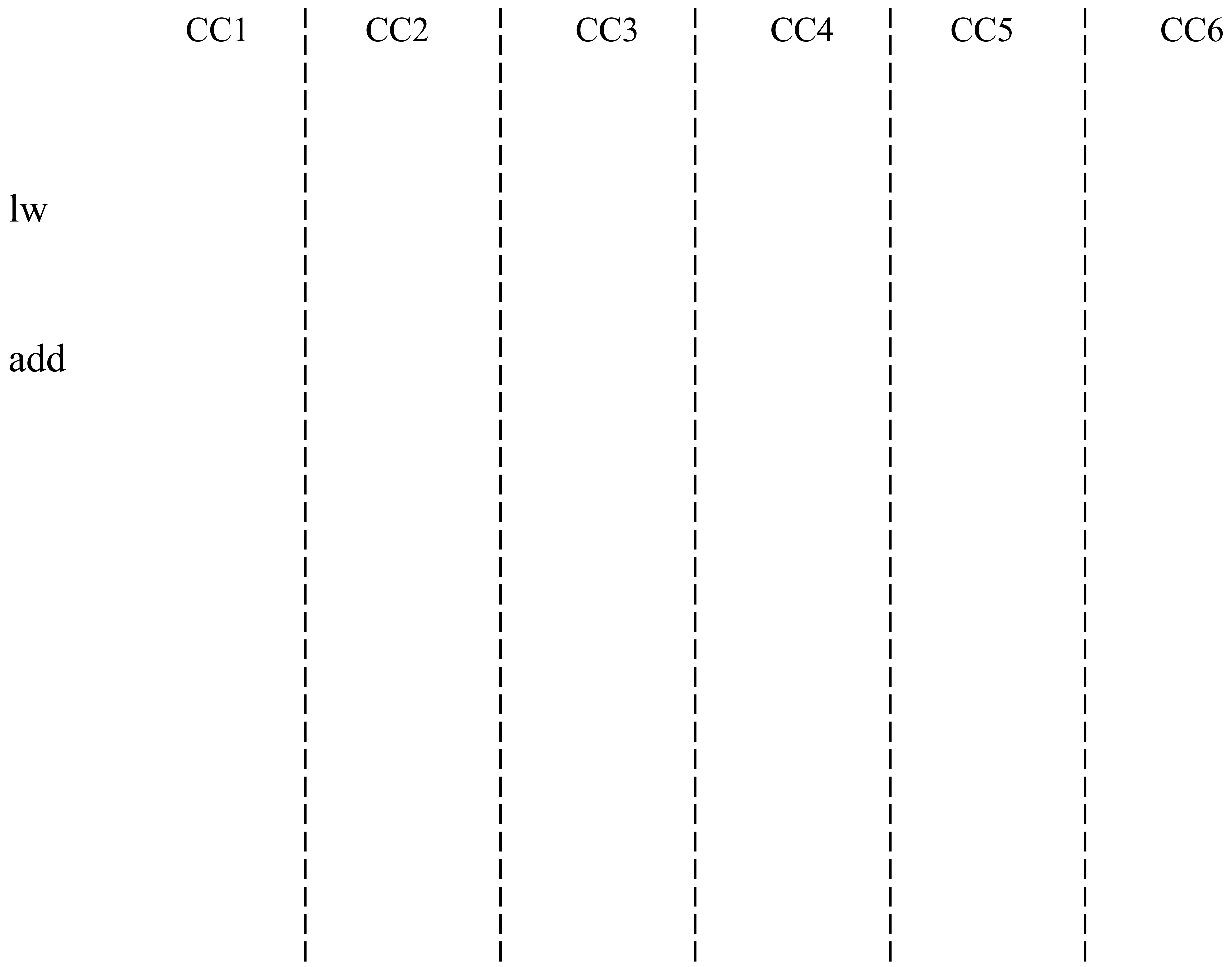
Execution in Pipelined Datapath



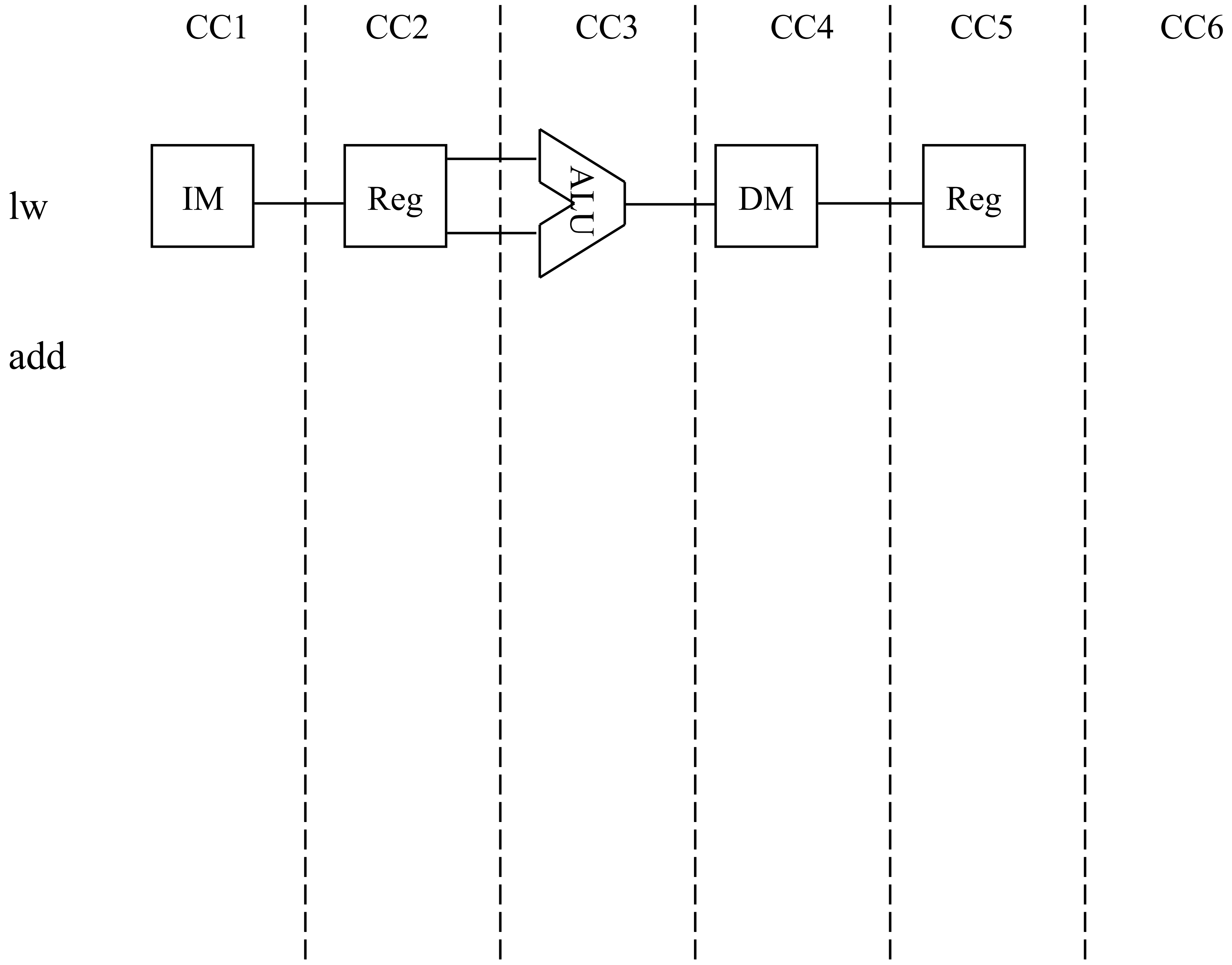
Execution in Pipelined Datapath



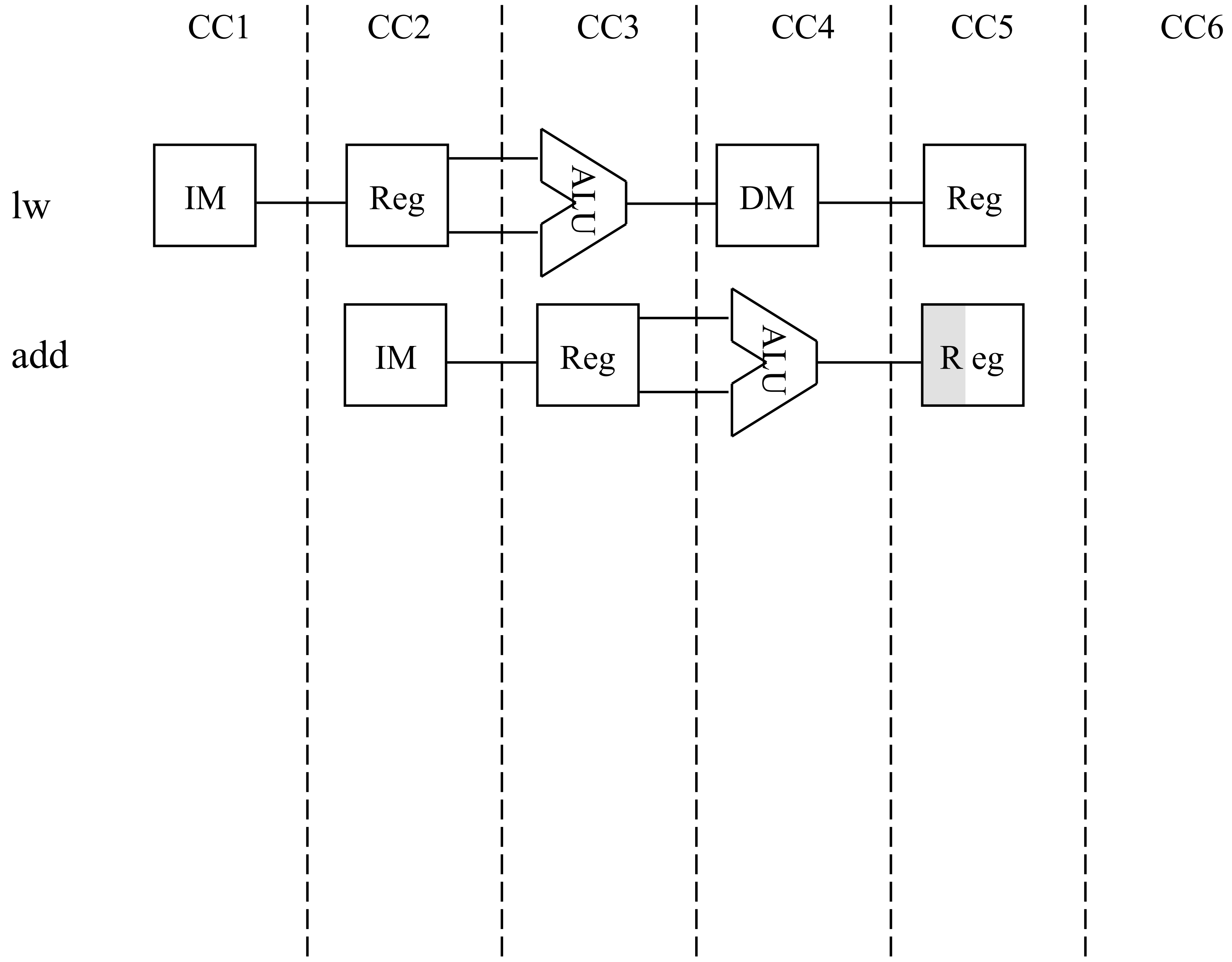
Mixed Instructions in the Pipeline



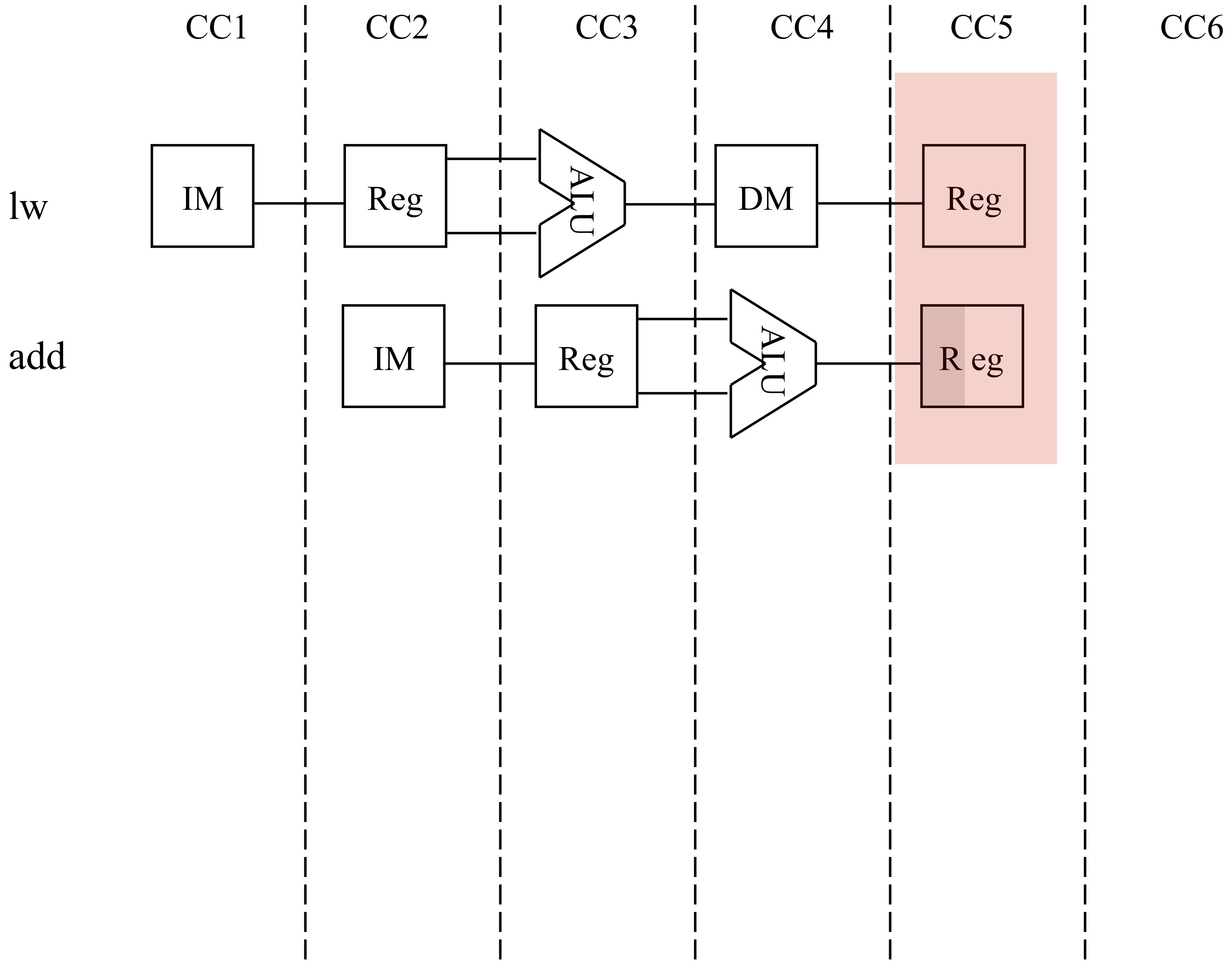
Mixed Instructions in the Pipeline



Mixed Instructions in the Pipeline

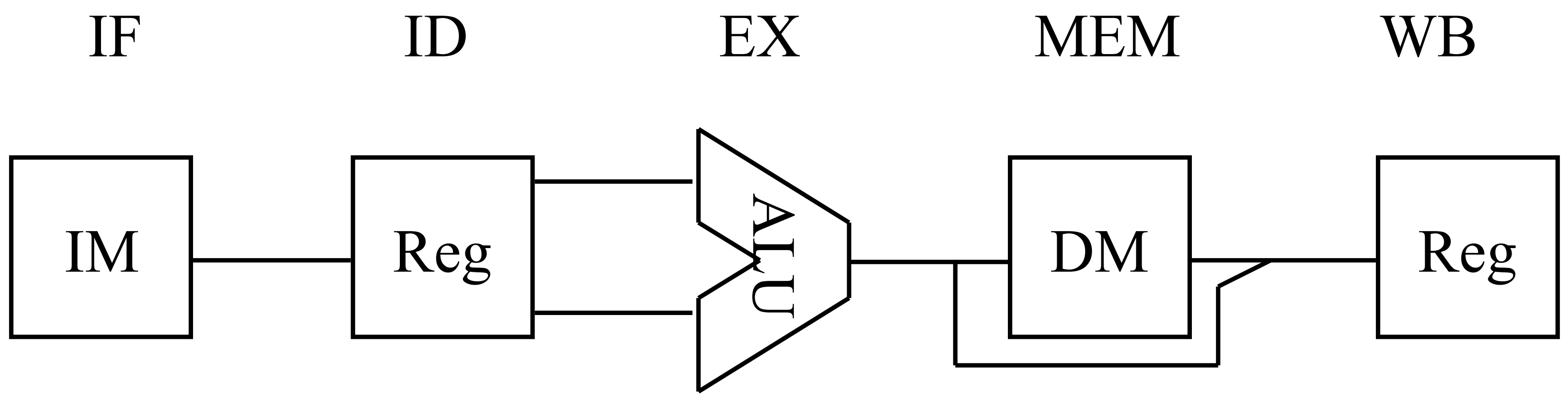


Mixed Instructions in the Pipeline



Pipeline Principles

- All instructions that share a pipeline should have the same stages in the same order.
    - therefore, add does nothing during Mem stage
    - sw does nothing during WB stage
- All intermediate values must be latched each cycle.
- There is no functional block reuse



Pipelined Datapath

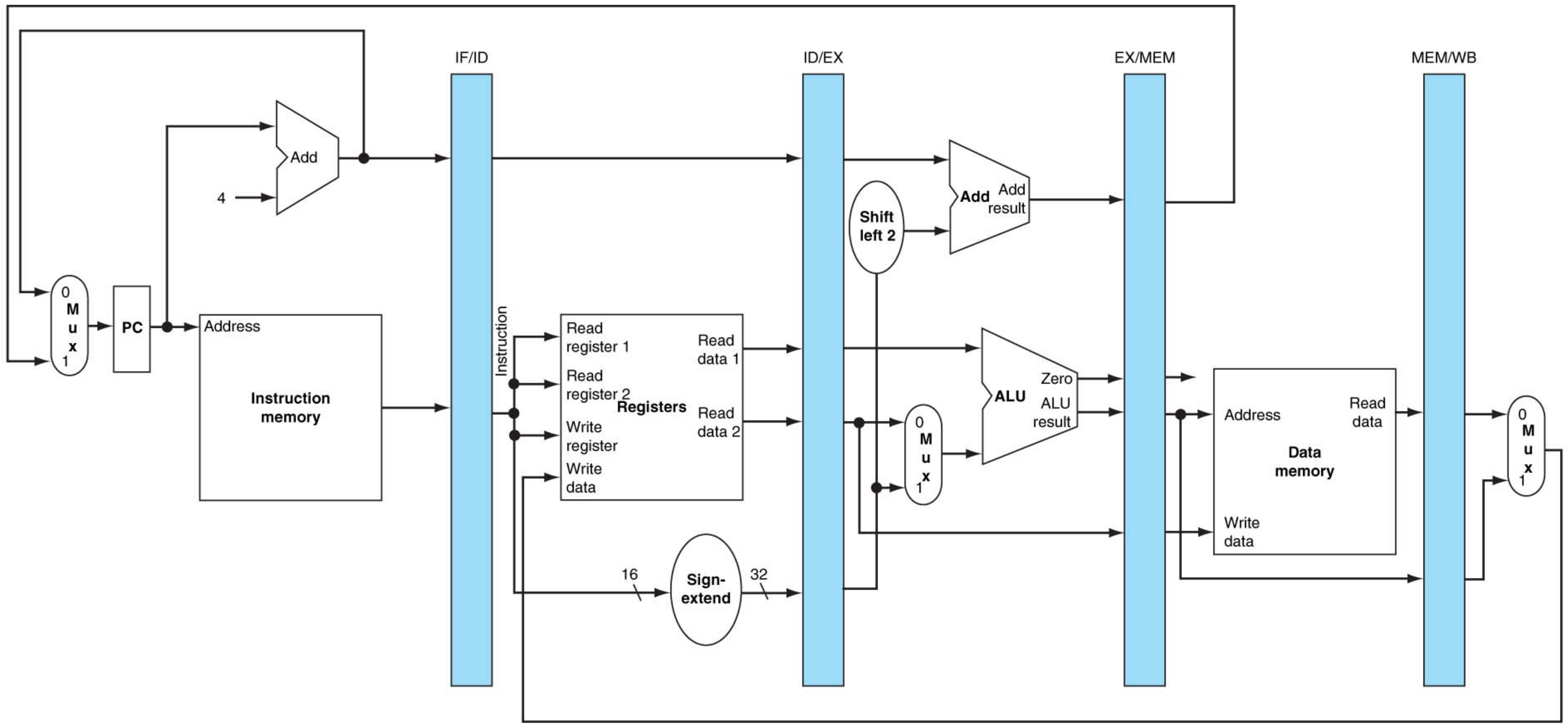
Instruction Fetch

Decode /
Reg. Fetch

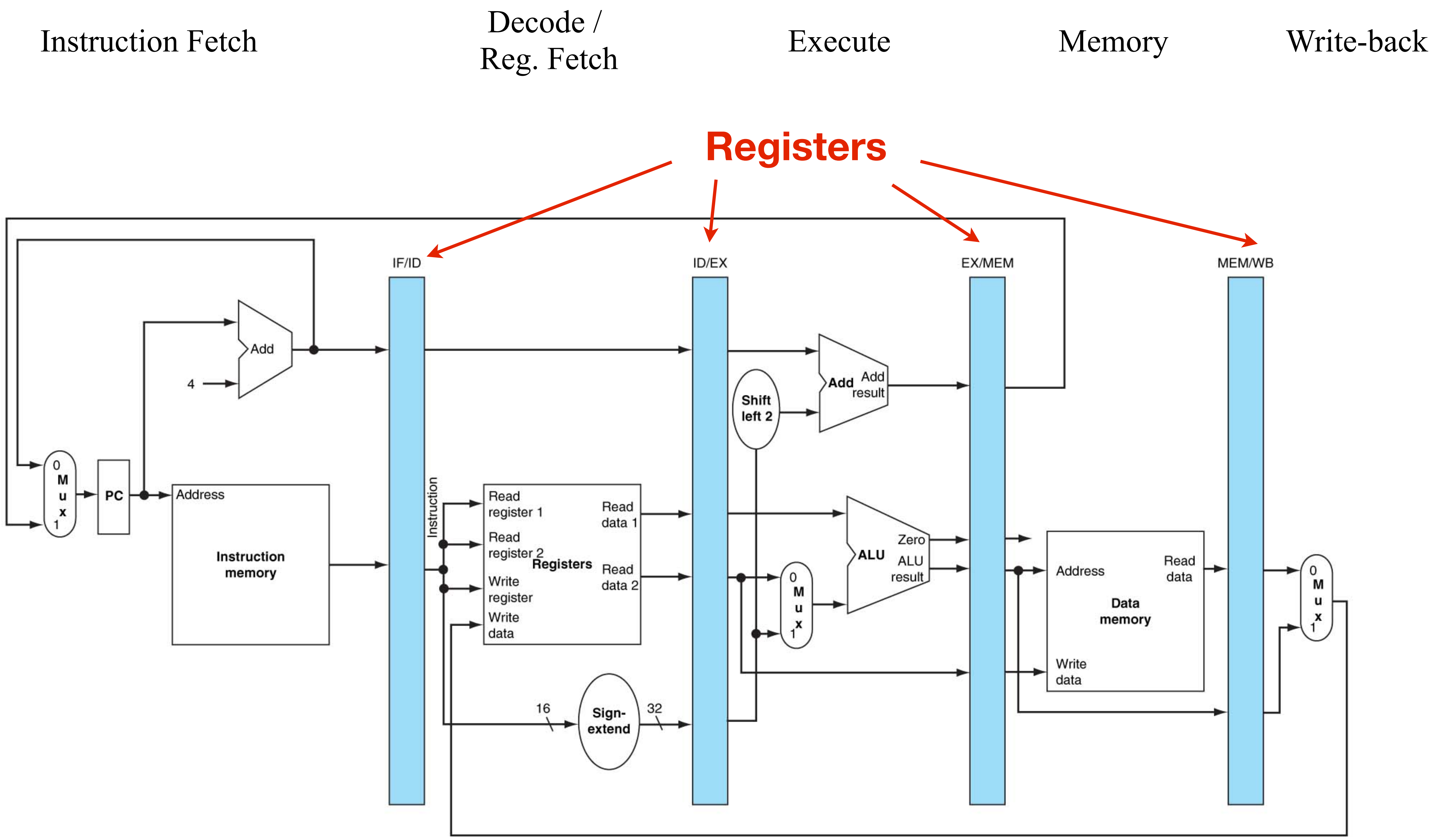
Execute

Memory

Write-back



Pipelined Datapath



Pipeline In Execution

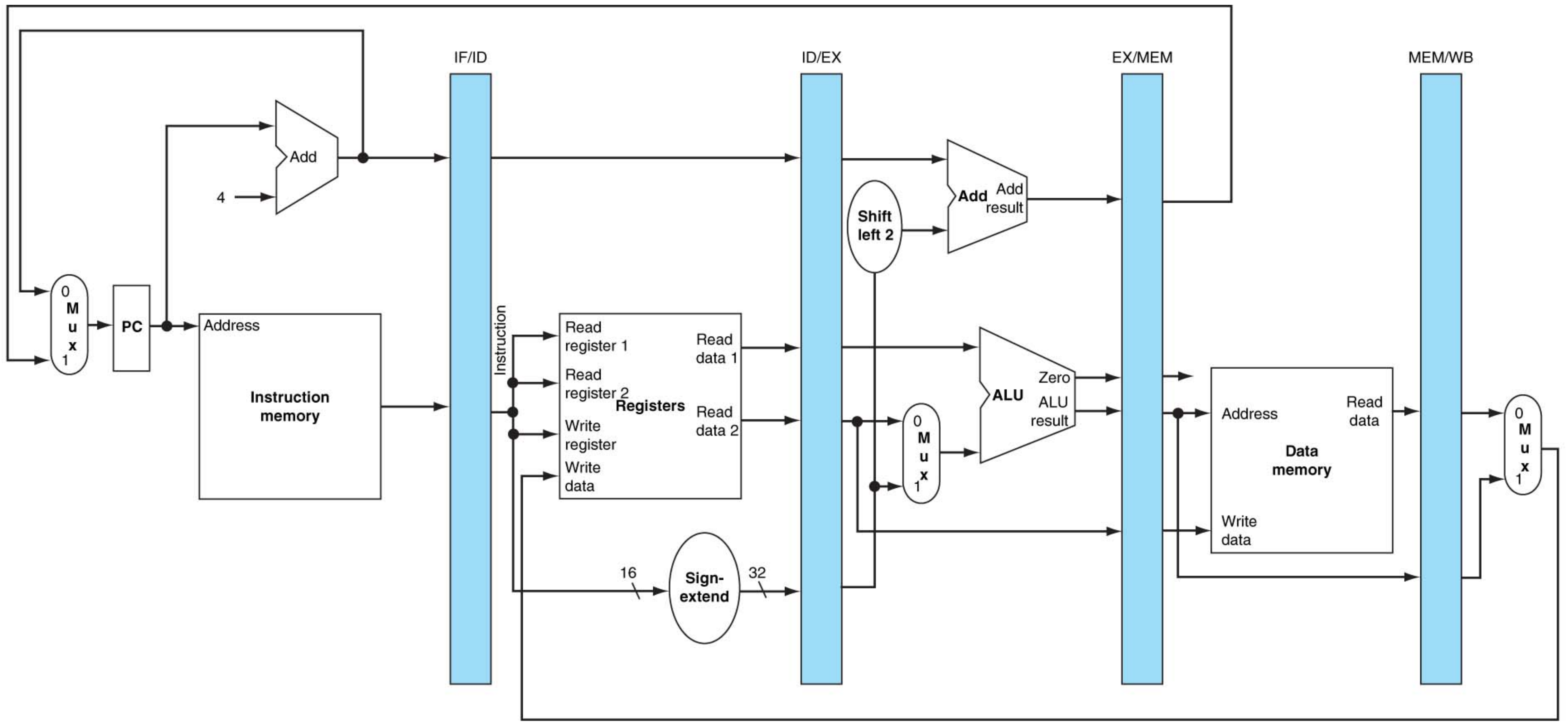
Instruction Fetch

Decode /
Reg. Fetch

Execute

Memory

Write-back



Pipeline In Execution

Instruction Fetch

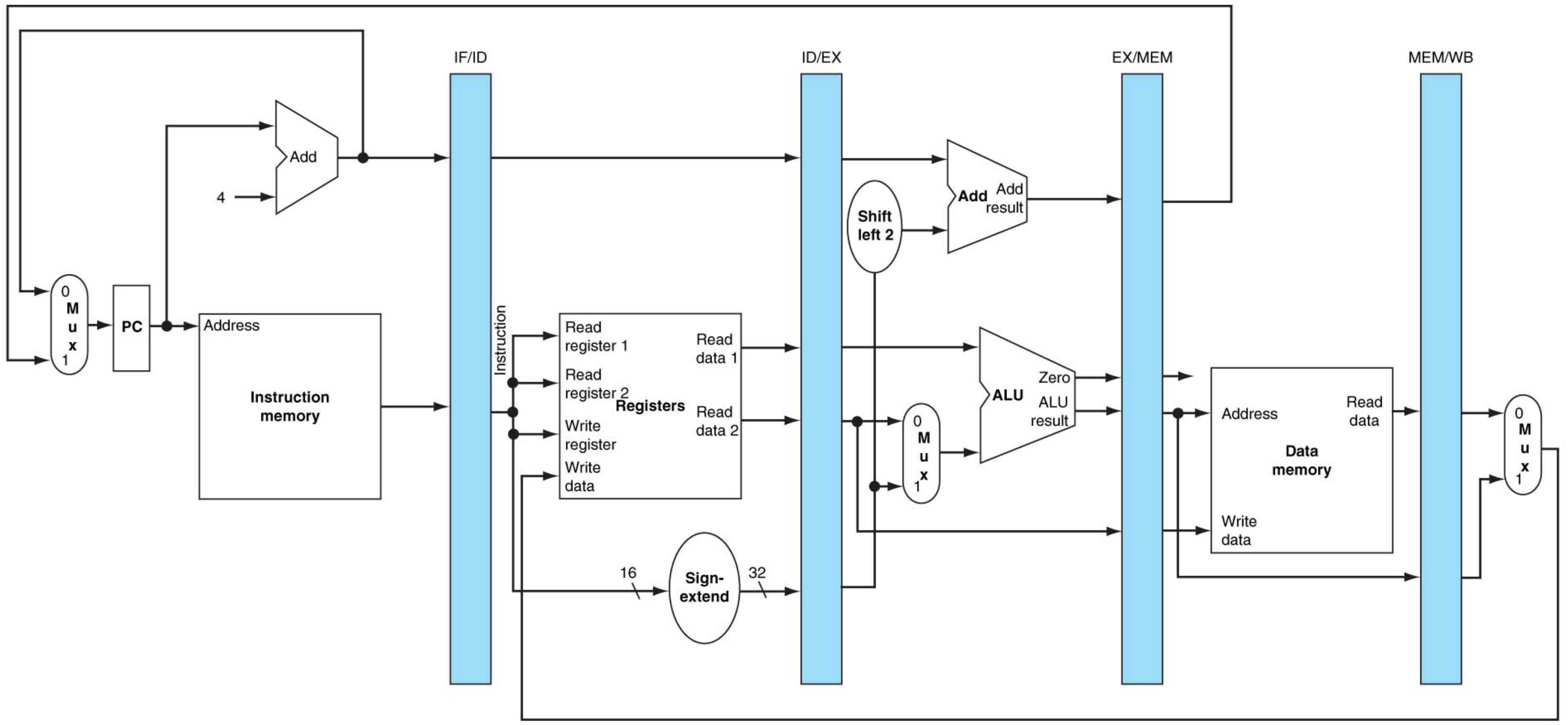
Decode /
Reg. Fetch

Execute

Memory

Write-back

add $10, $1, $2



Pipeline In Execution

Instruction Fetch

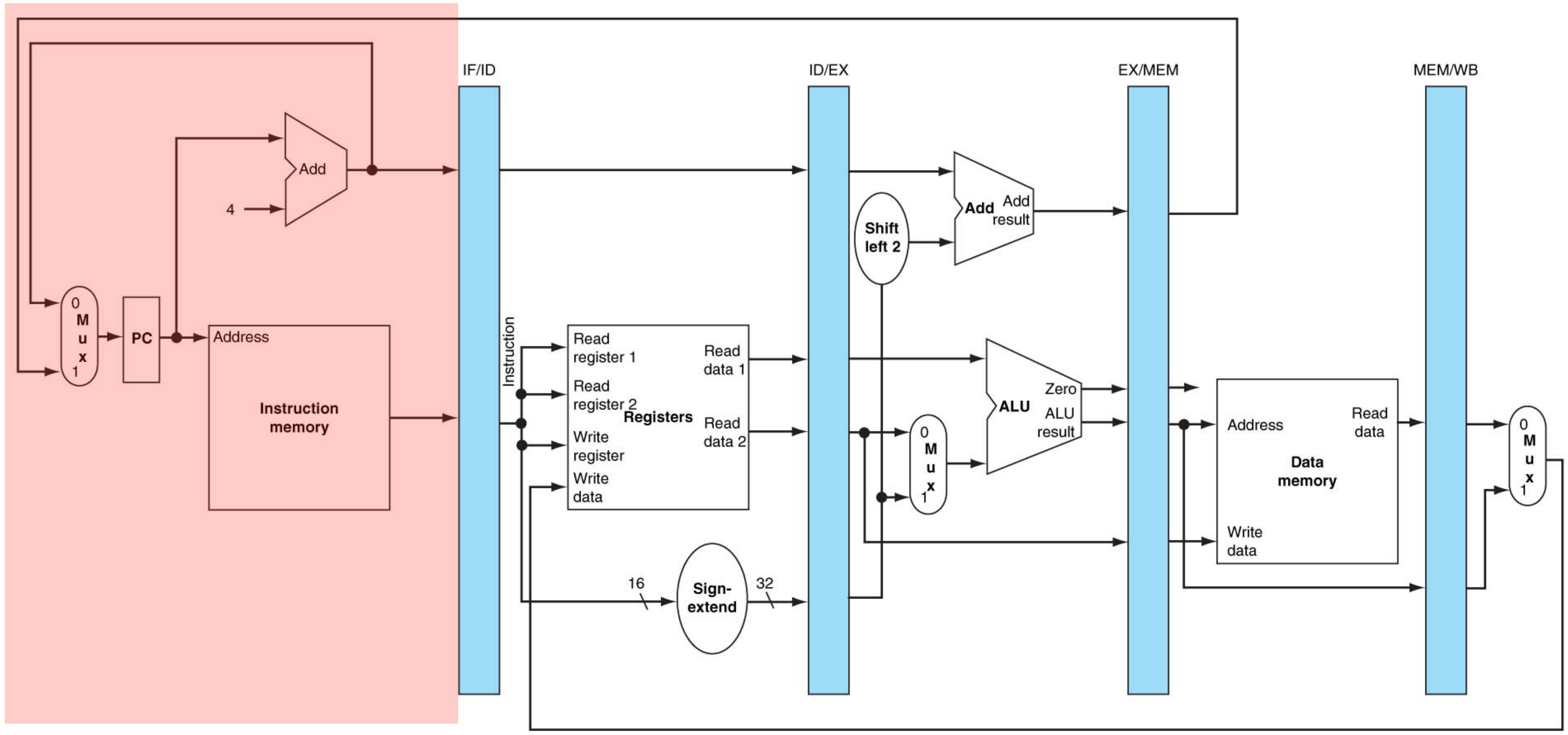
Decode / Reg. Fetch

Execute

Memory

Write-back

add $10, $1, $2



Pipeline In Execution

Instruction Fetch

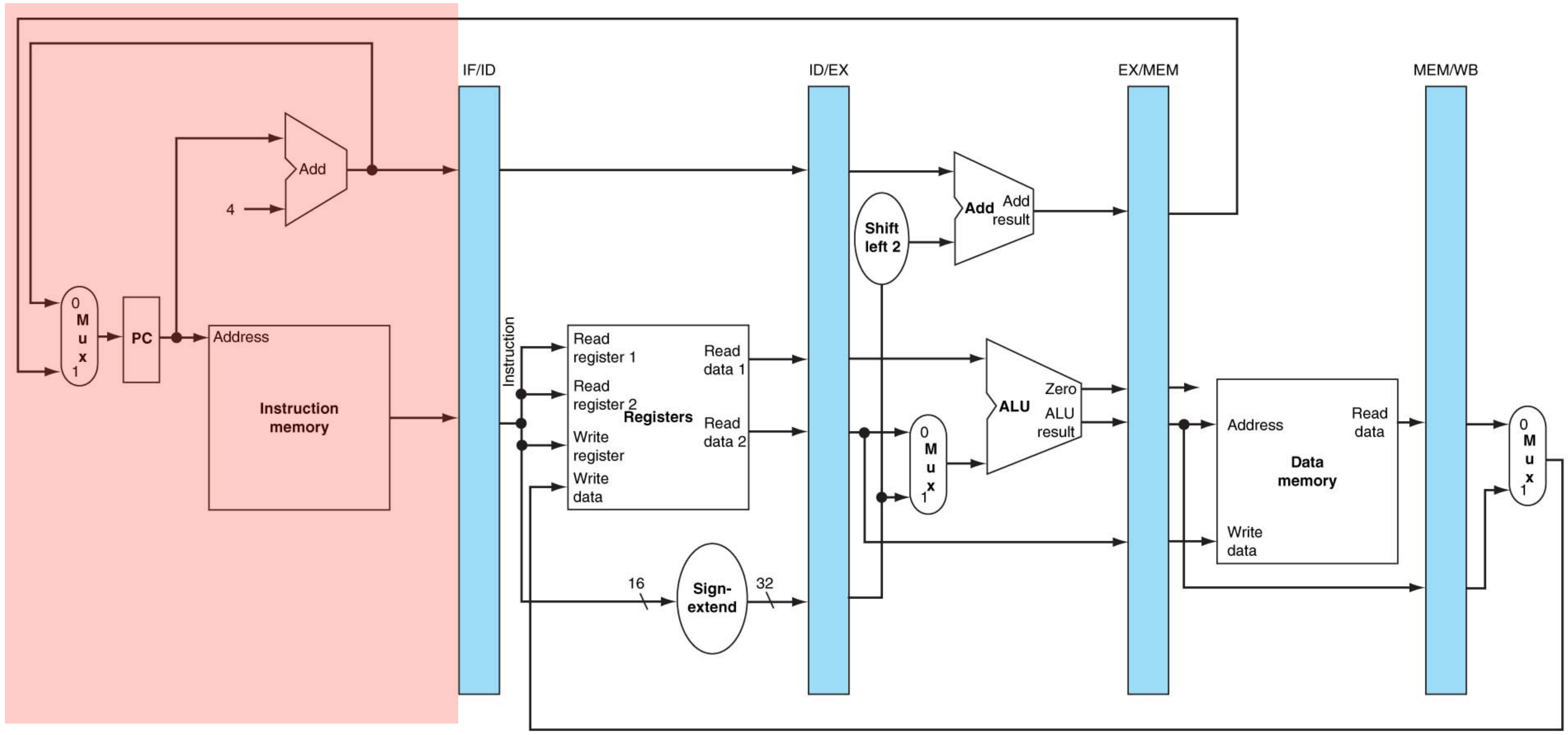
Decode / Reg. Fetch

Execute

Memory

Write-back

add $10, $1, $2



Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

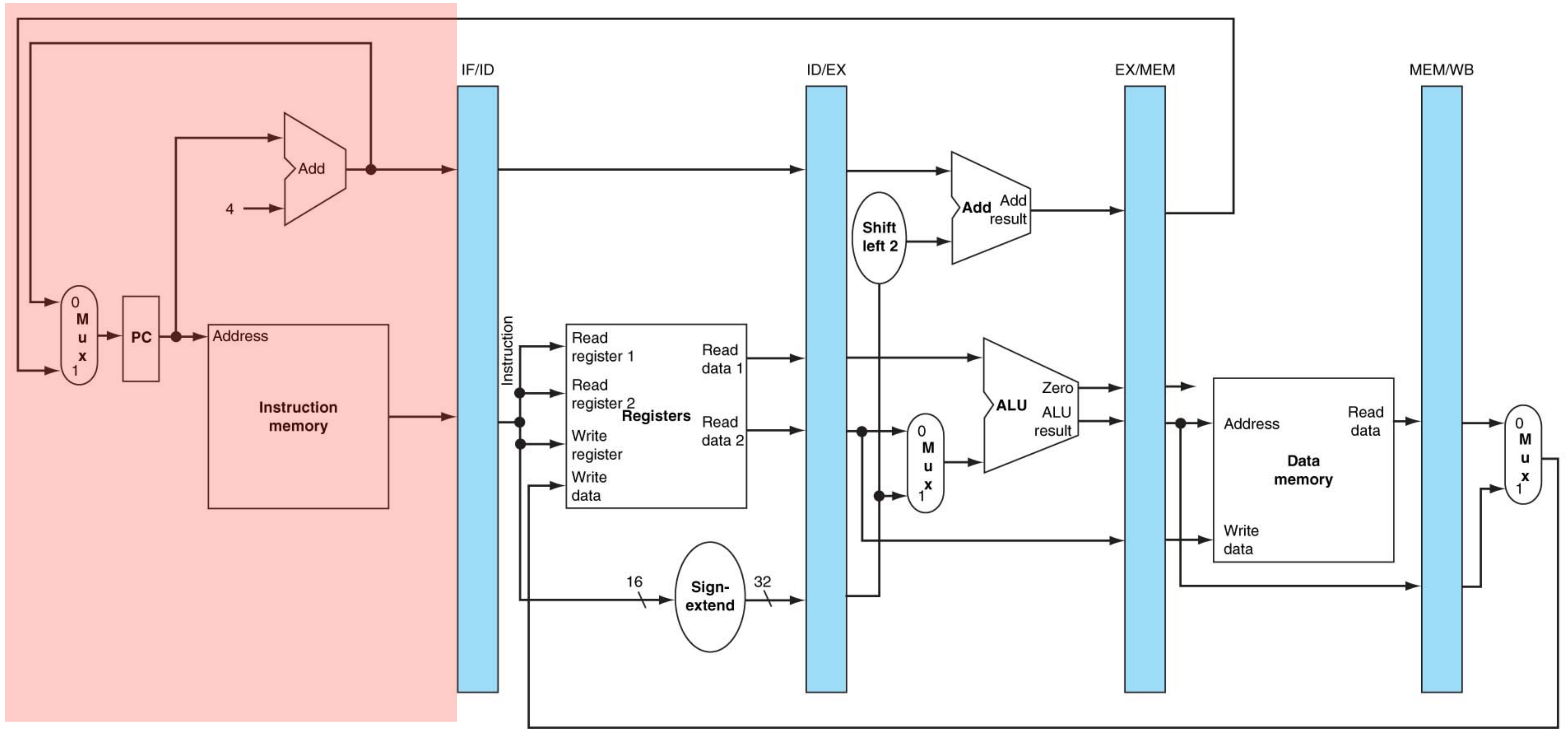
Execute

Memory

Write-back

lw $12, 1000($4)

add $10, $1, $2



Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

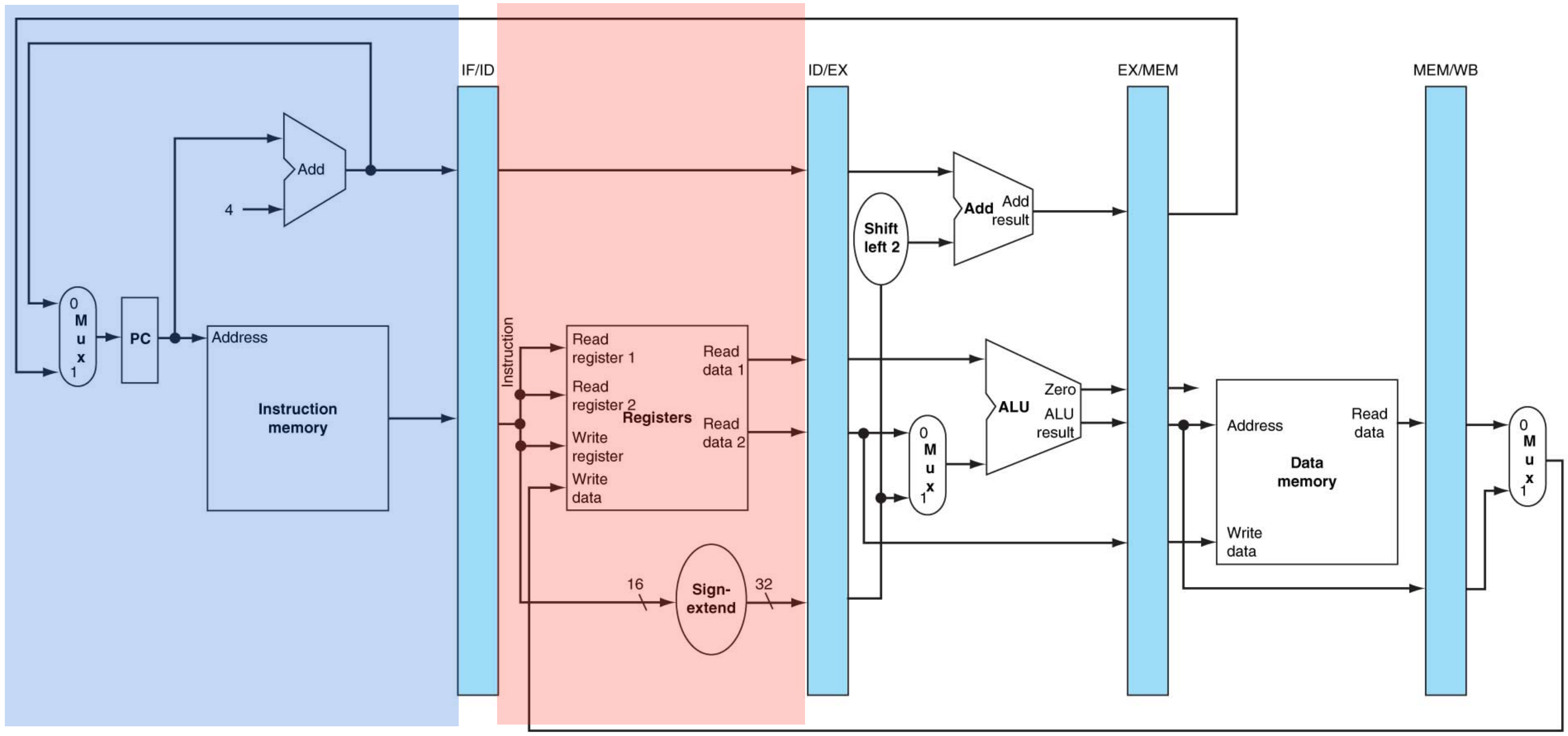
Execute

Memory

Write-back

lw $12, 1000($4)

add $10, $1, $2



Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

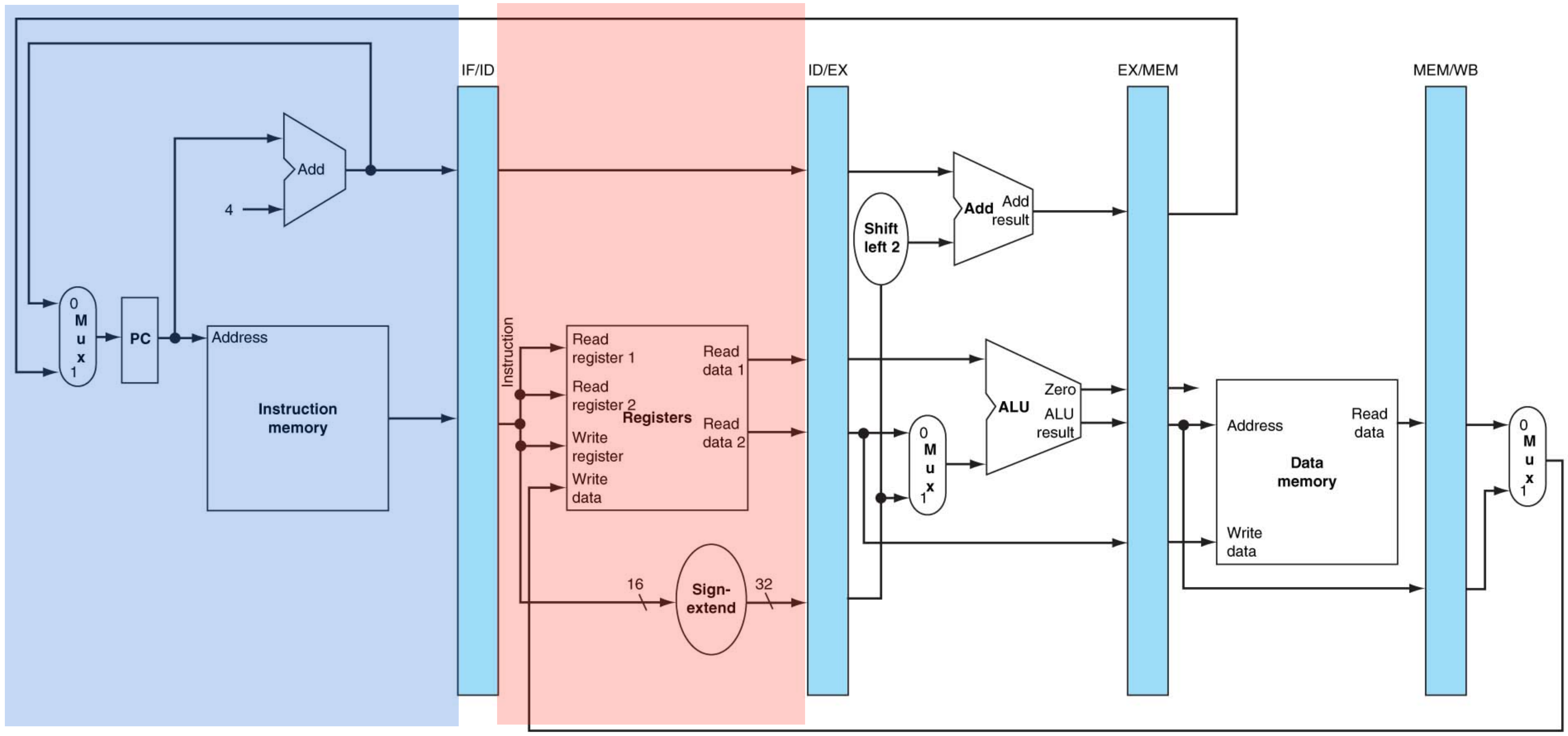
Execute

Memory

Write-back

lw $12, 1000($4)

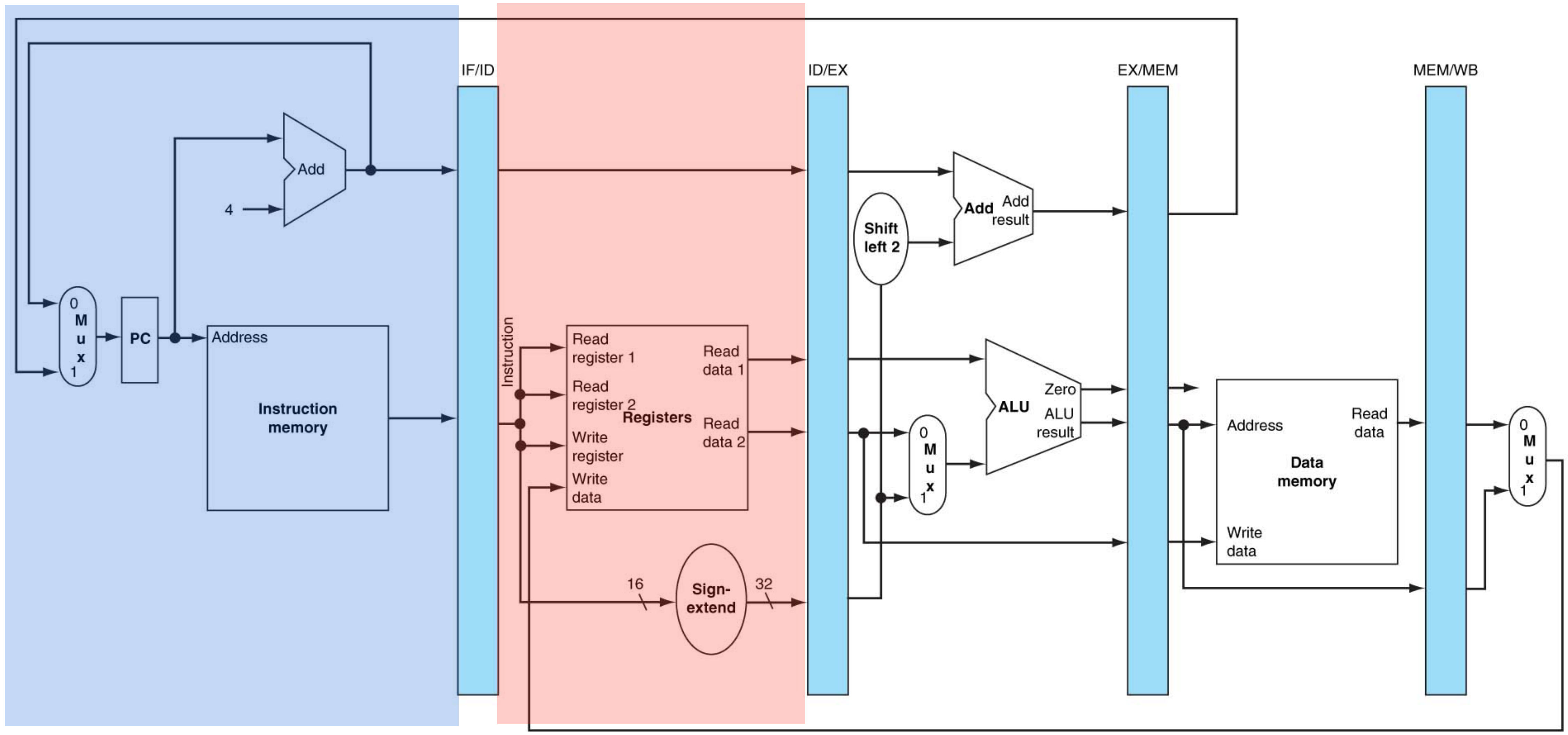
add $10, $1, $2



Pipeline In Execution

Instruction Fetch | Decode / Reg. Fetch | Execute | Memory | Write-back

sub $15, $4, $1 | lw $12, 1000($4) | add $10, $1, $2



Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

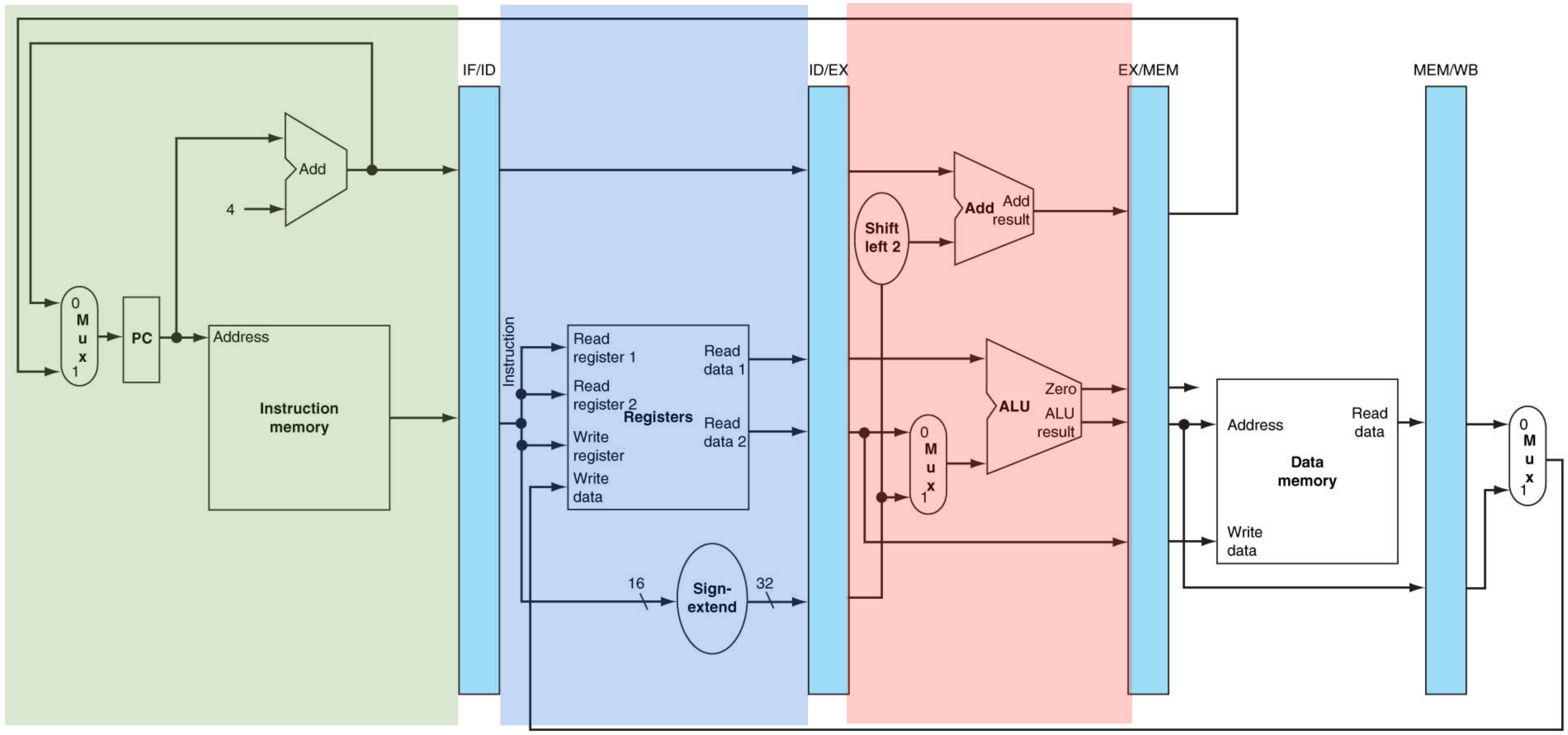
Memory

Write-back

sub $15, $4, $1

lw $12, 1000($4)

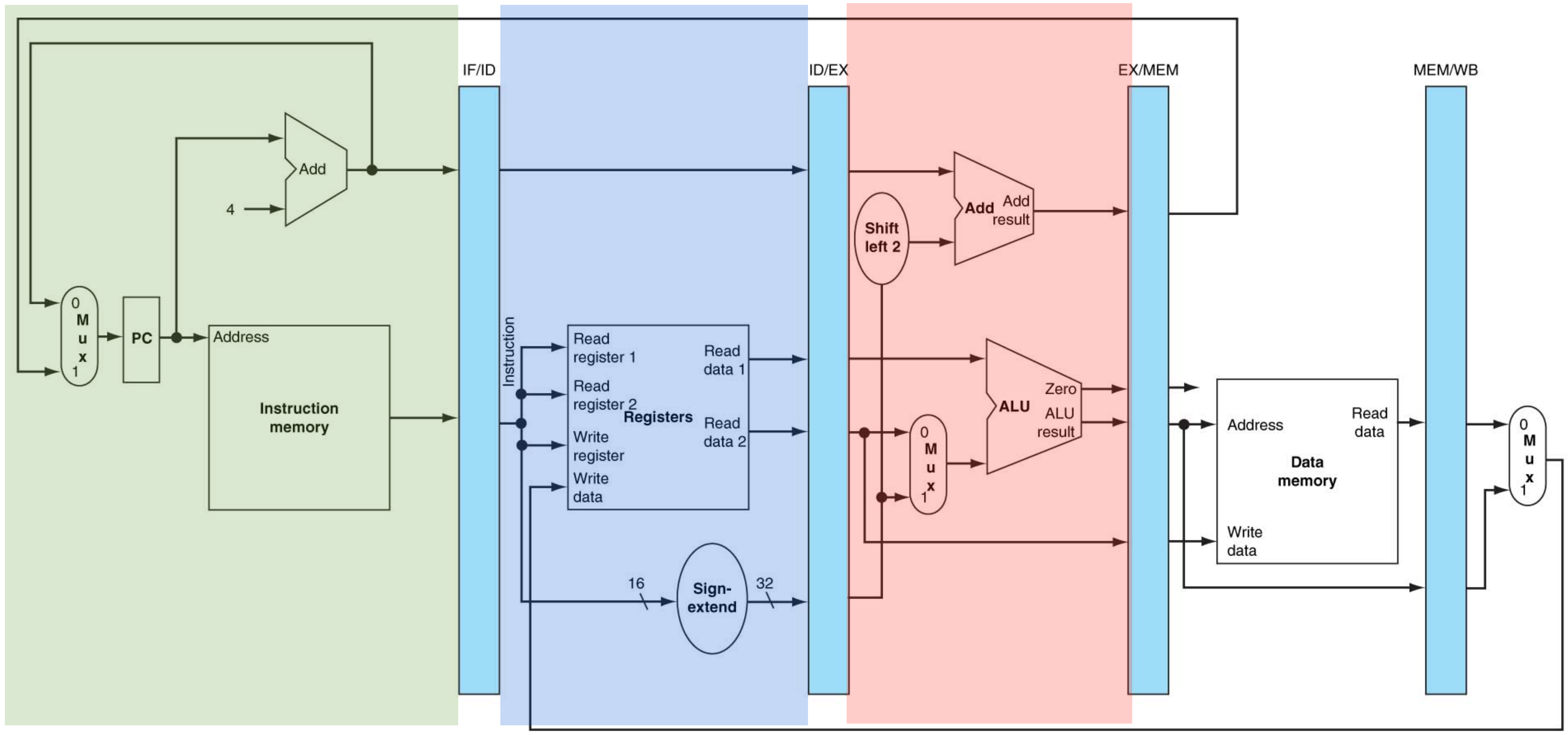
add $10, $1, $2



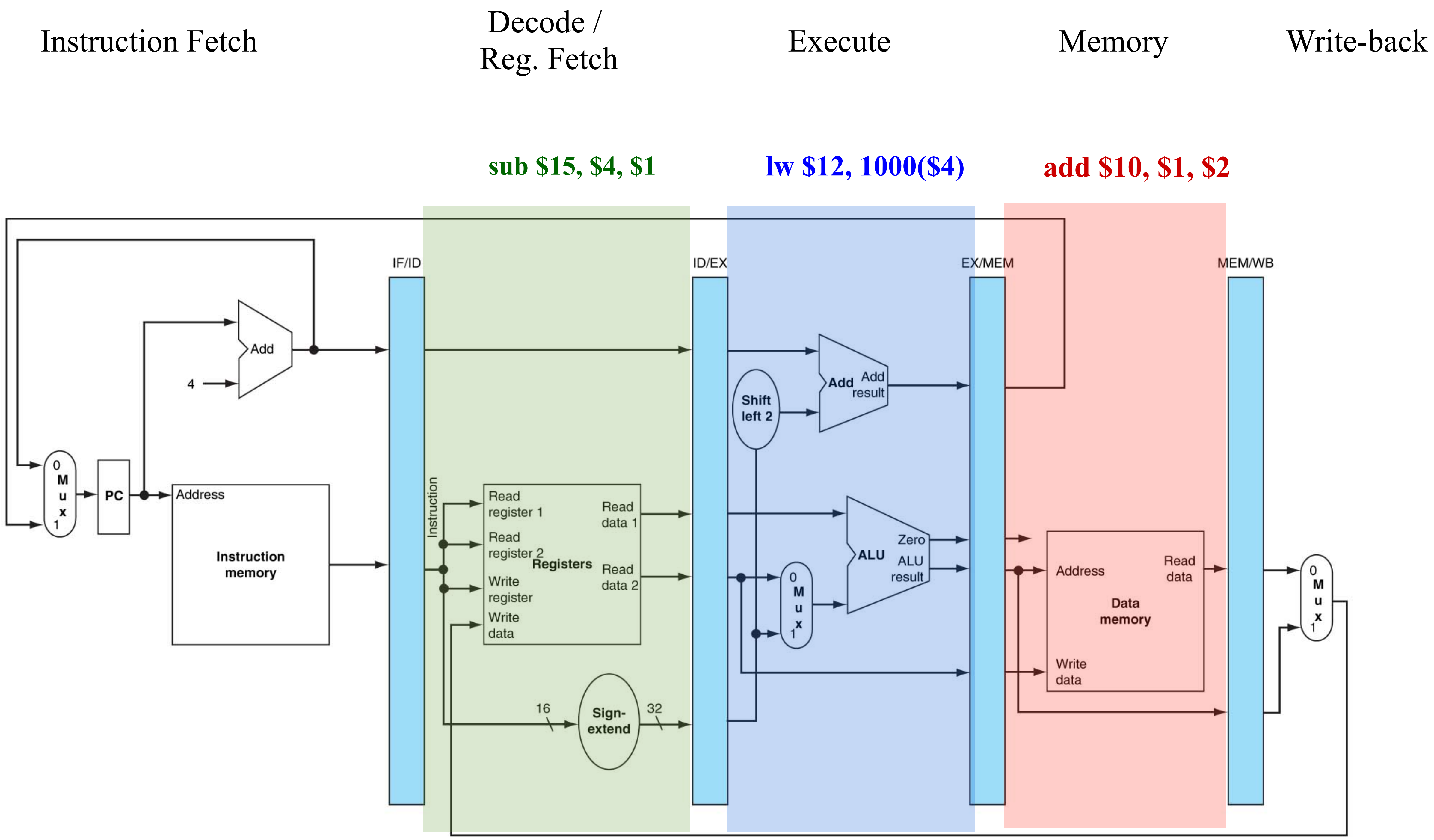
Pipeline In Execution

Instruction Fetch | Decode / Reg. Fetch | Execute | Memory | Write-back

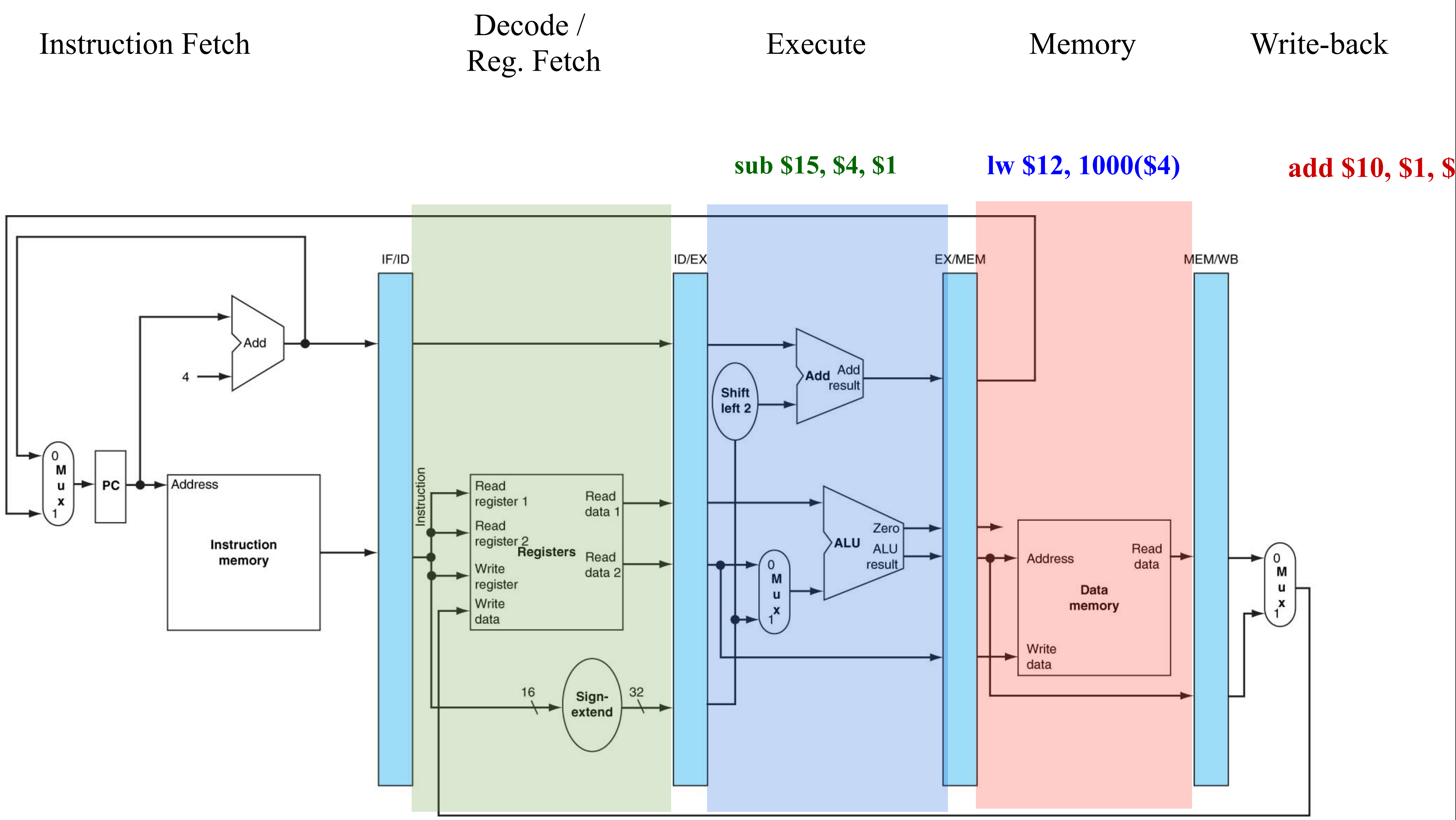
sub $15, $4, $1 | lw $12, 1000($4) | add $10, $1, $2



Pipeline In Execution



Pipeline In Execution



Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

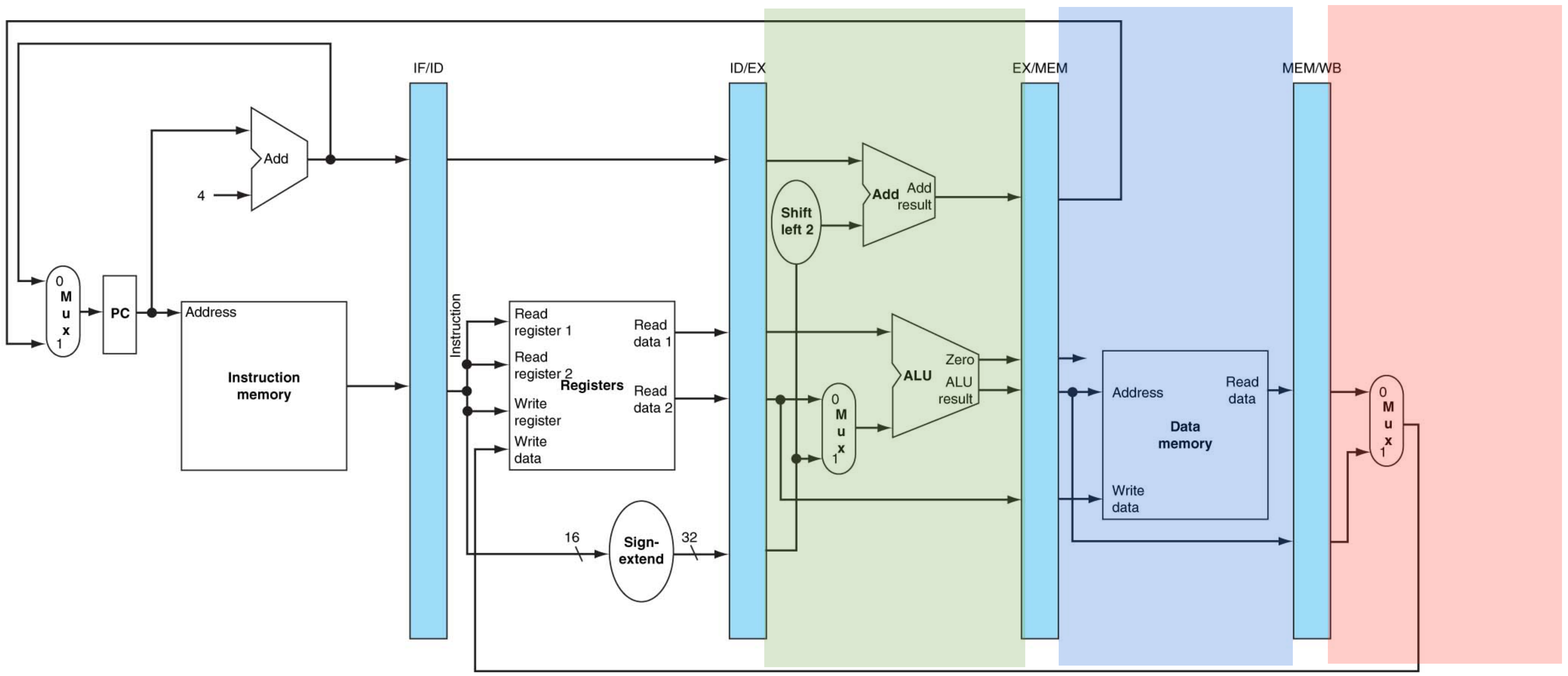
Memory

Write-back

sub $15, $4, $1

lw $12, 1000($4)

add $10, $1, $



Pipeline In Execution

Instruction Fetch

Decode /
Reg. Fetch

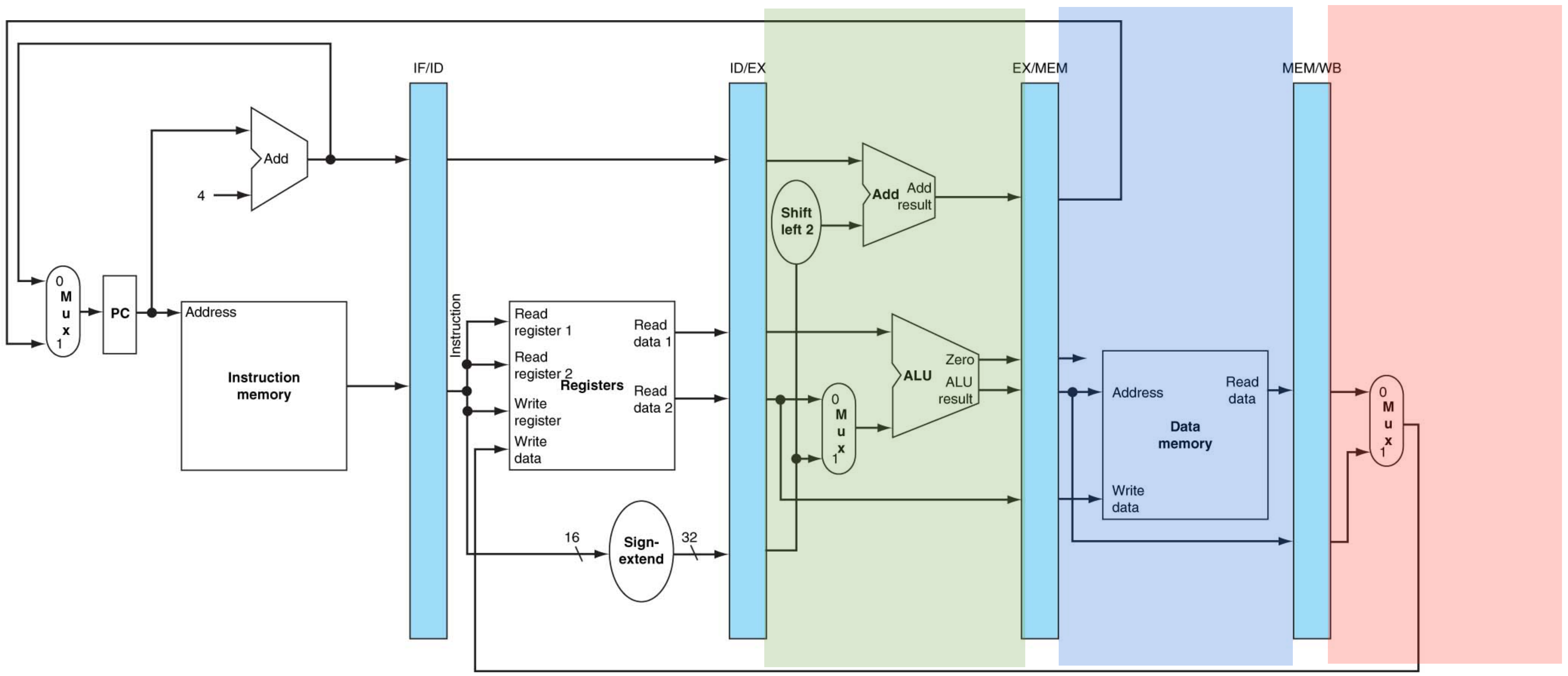
Execute

Memory

Write-back

sub $15, $4, $1

lw $12, 1000($4)



Pipeline In Execution

Instruction Fetch

Decode /
Reg. Fetch

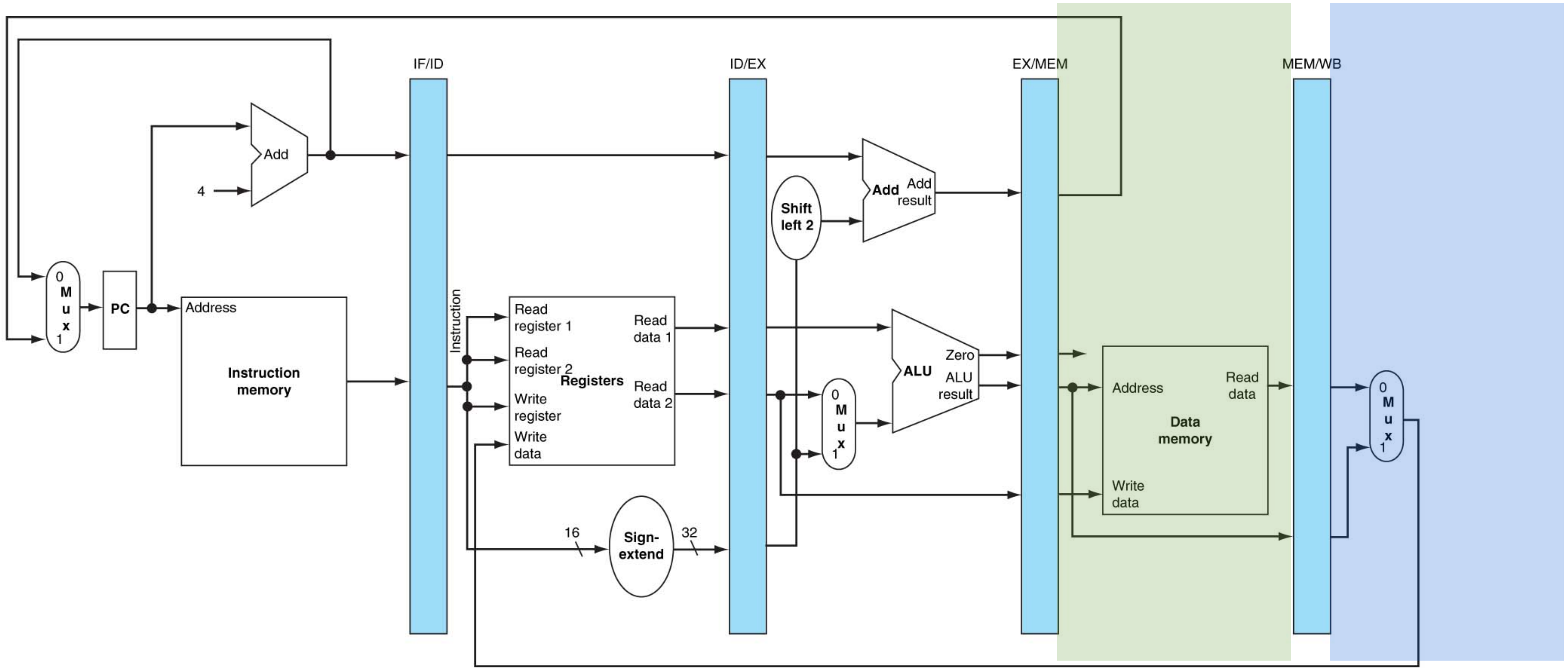
Execute

Memory

Write-back

sub $15, $4, $1

lw $12, 1000($4)



Pipeline In Execution

Instruction Fetch

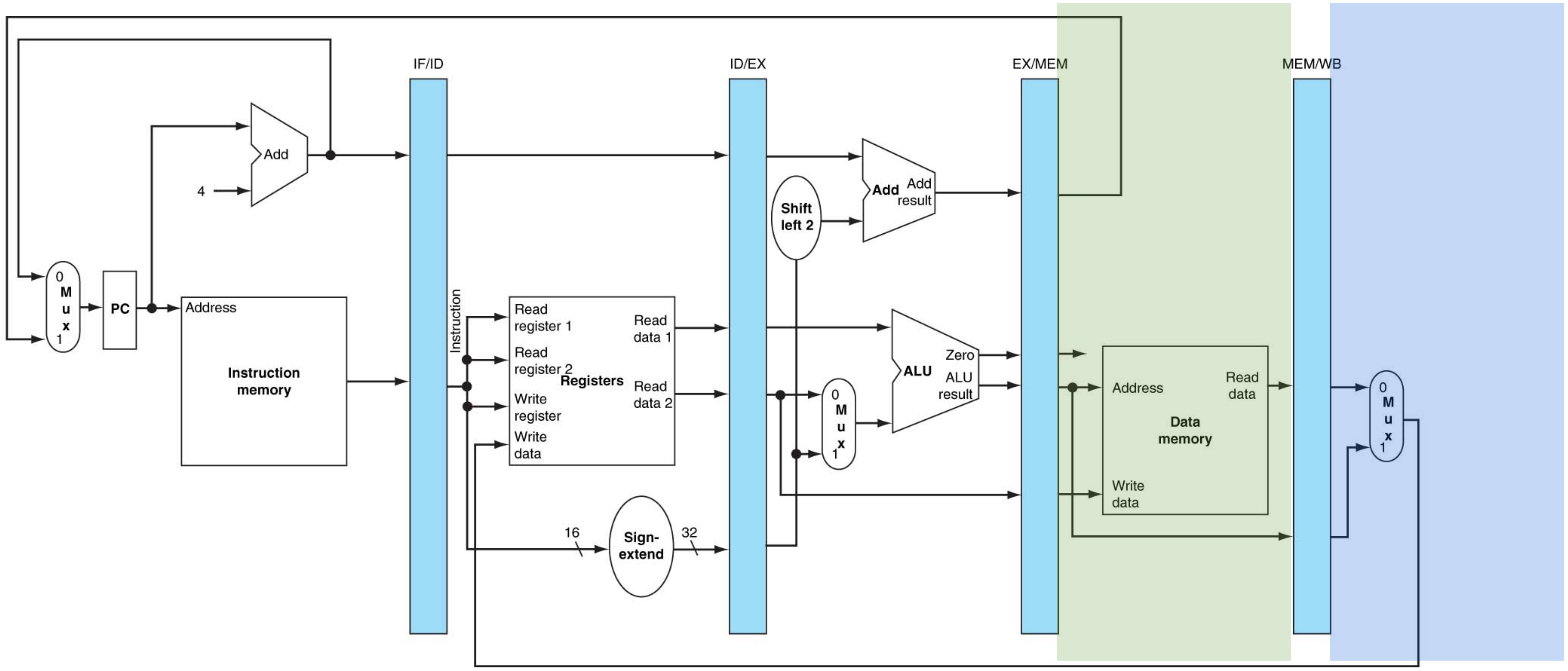
Decode /
Reg. Fetch

Execute

Memory

Write-back

sub $15, $4, $1



Pipeline In Execution

Instruction Fetch

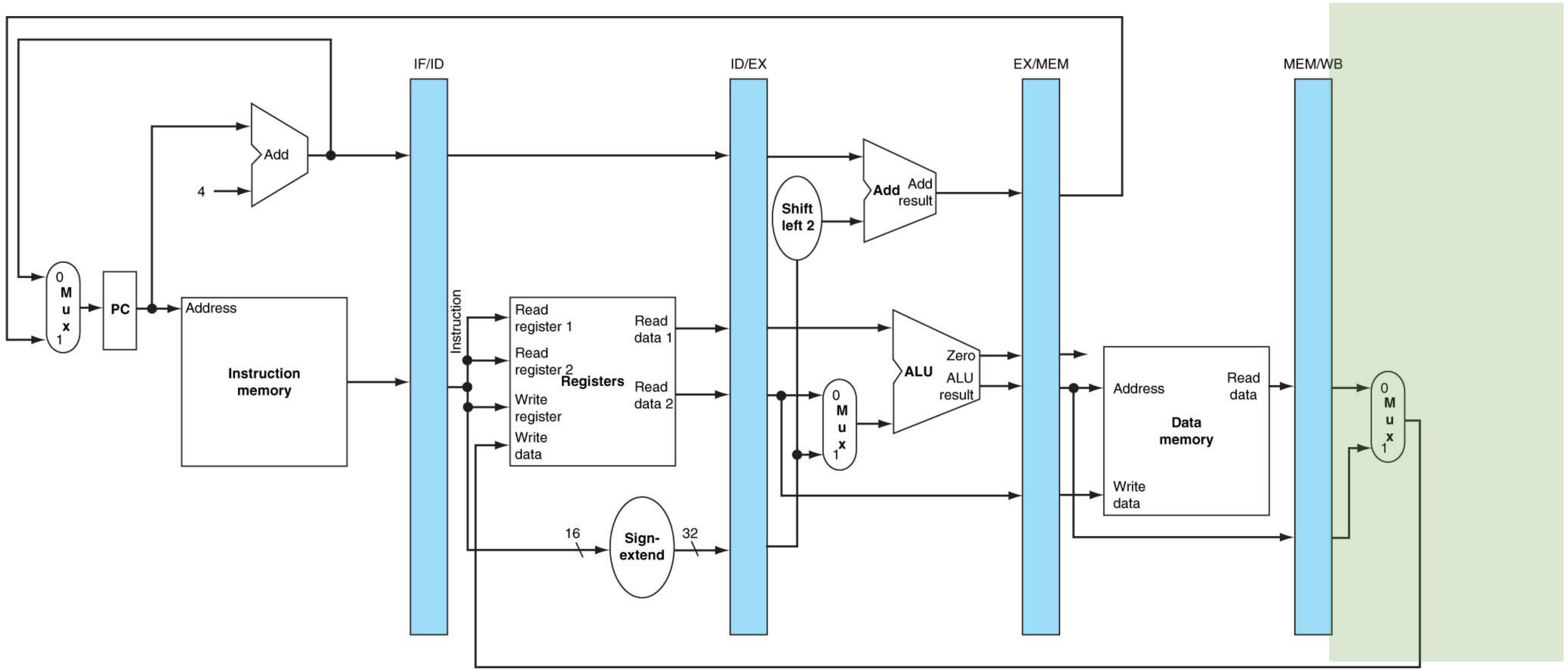
Decode / Reg. Fetch

Execute

Memory

Write-back

sub $15, $4, $1



Pipeline In Execution

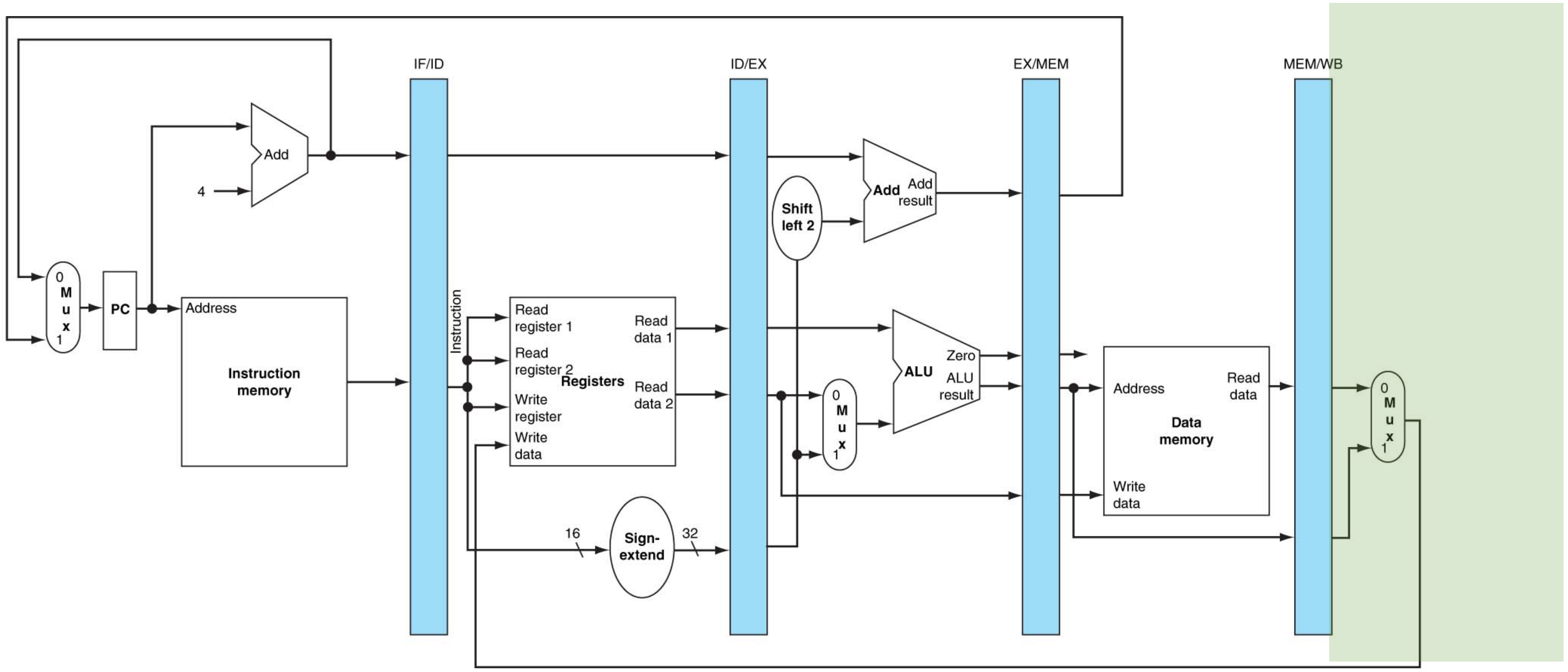
Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back



Pipeline In Execution

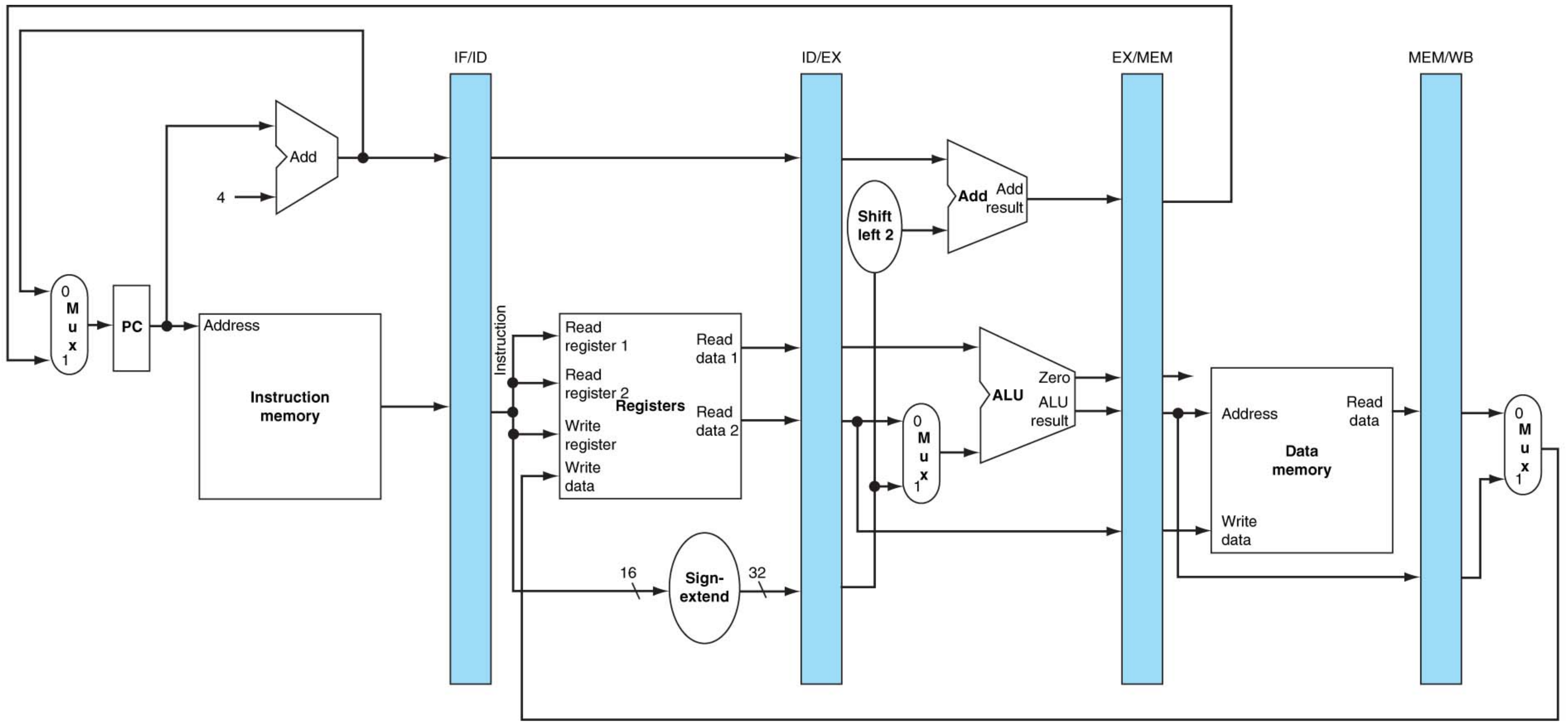
Instruction Fetch

Decode / Reg. Fetch

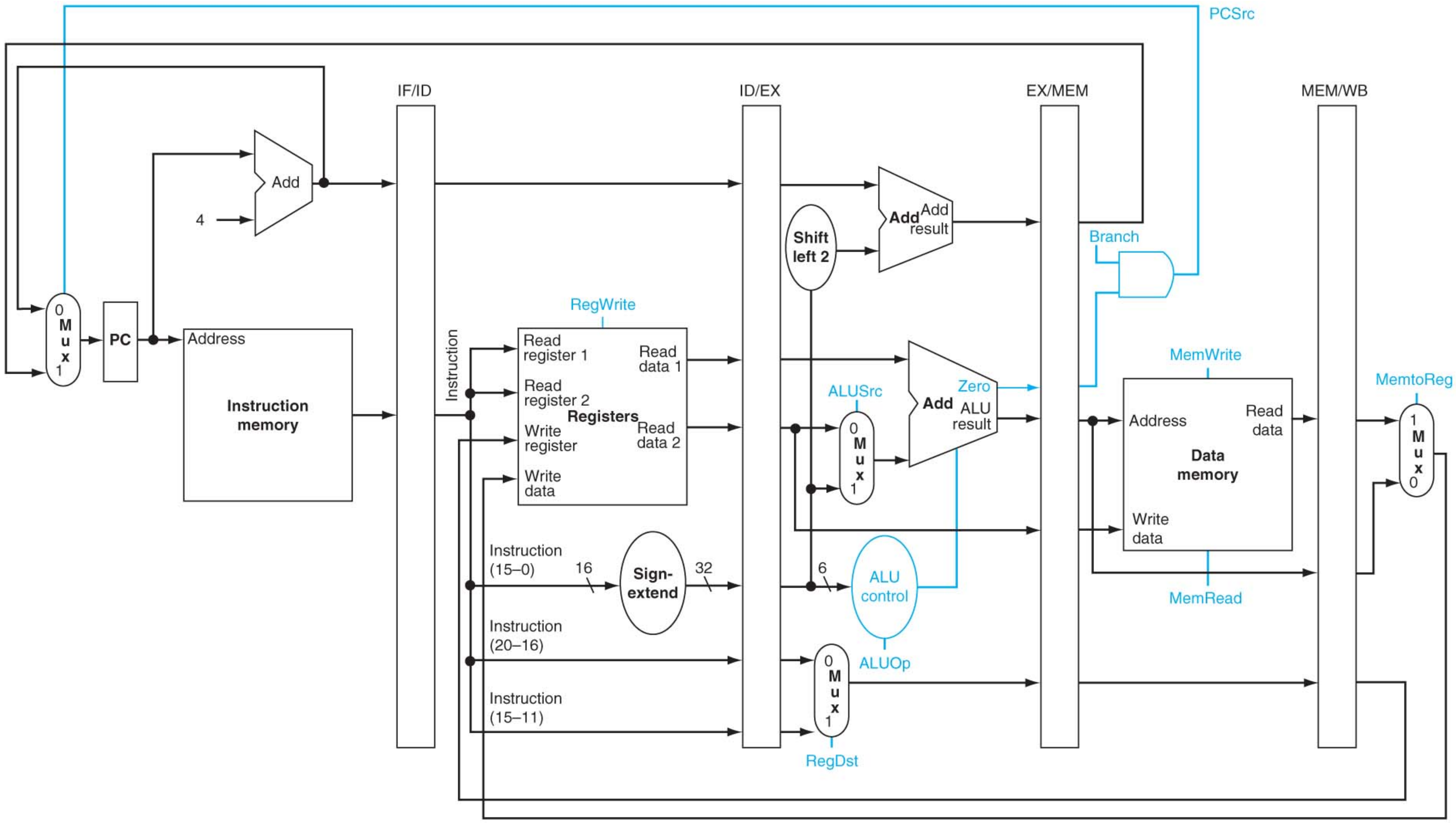
Execute

Memory

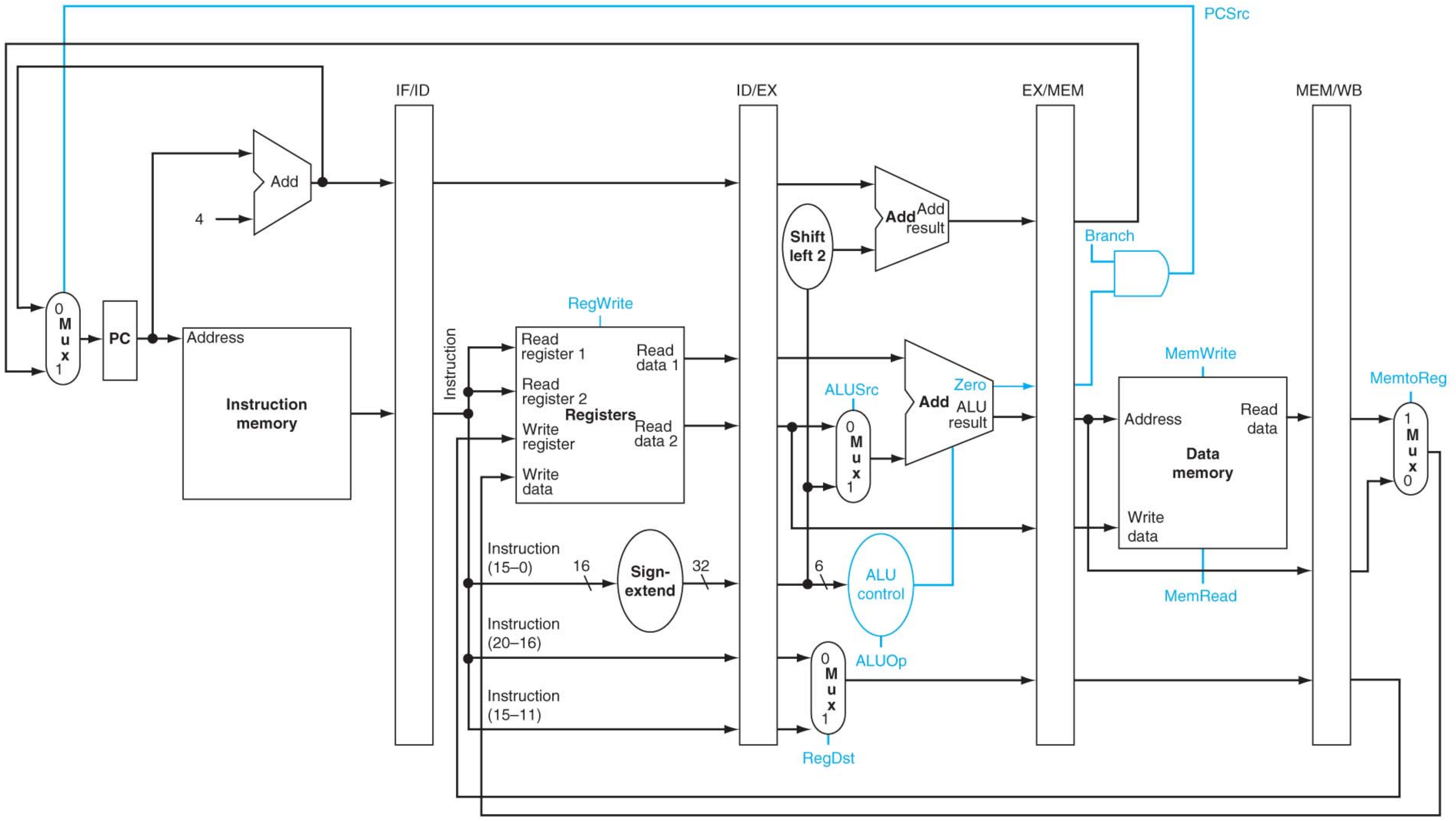
Write-back



Pipeline with Controls



Pipeline with Controls

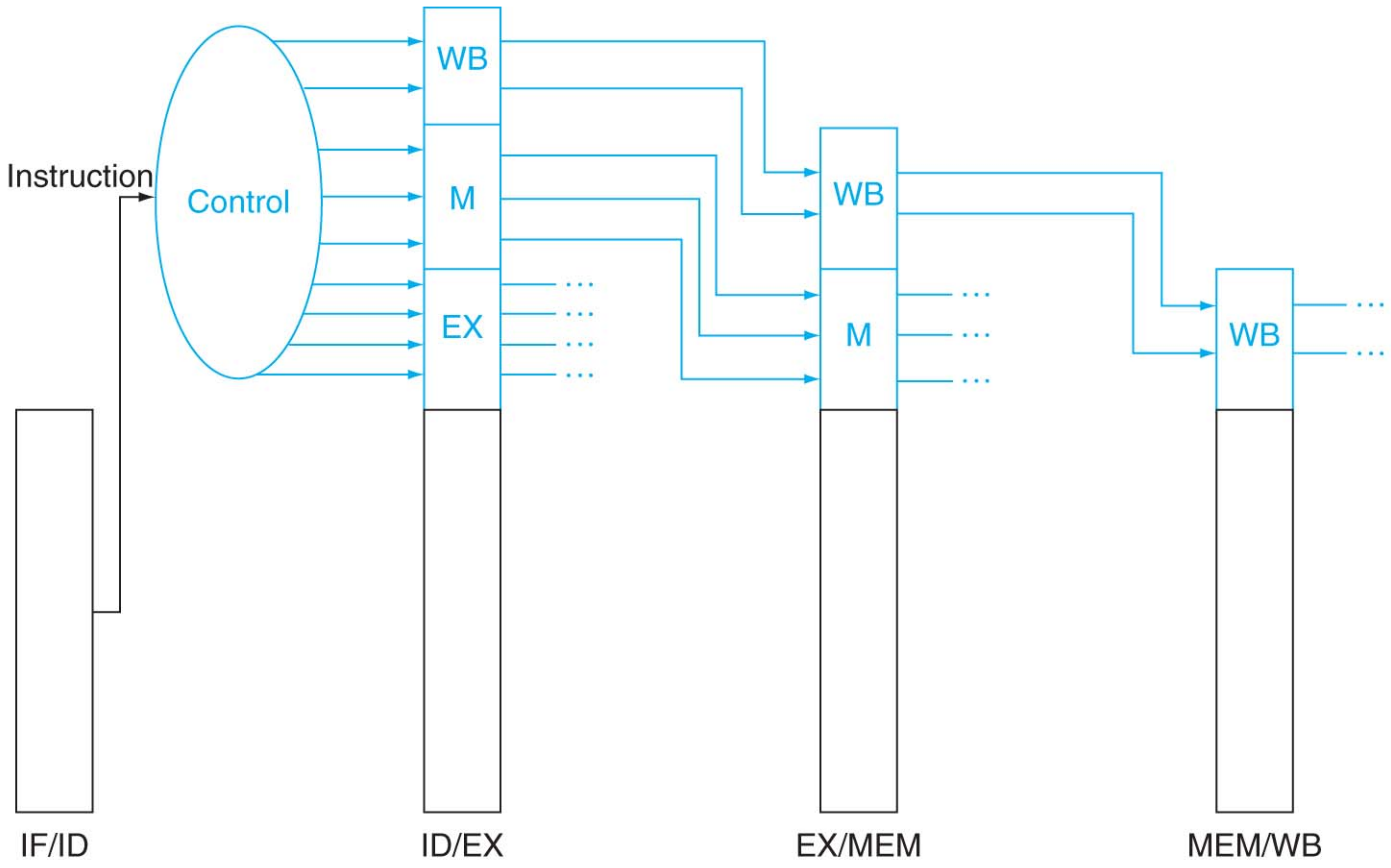


But...

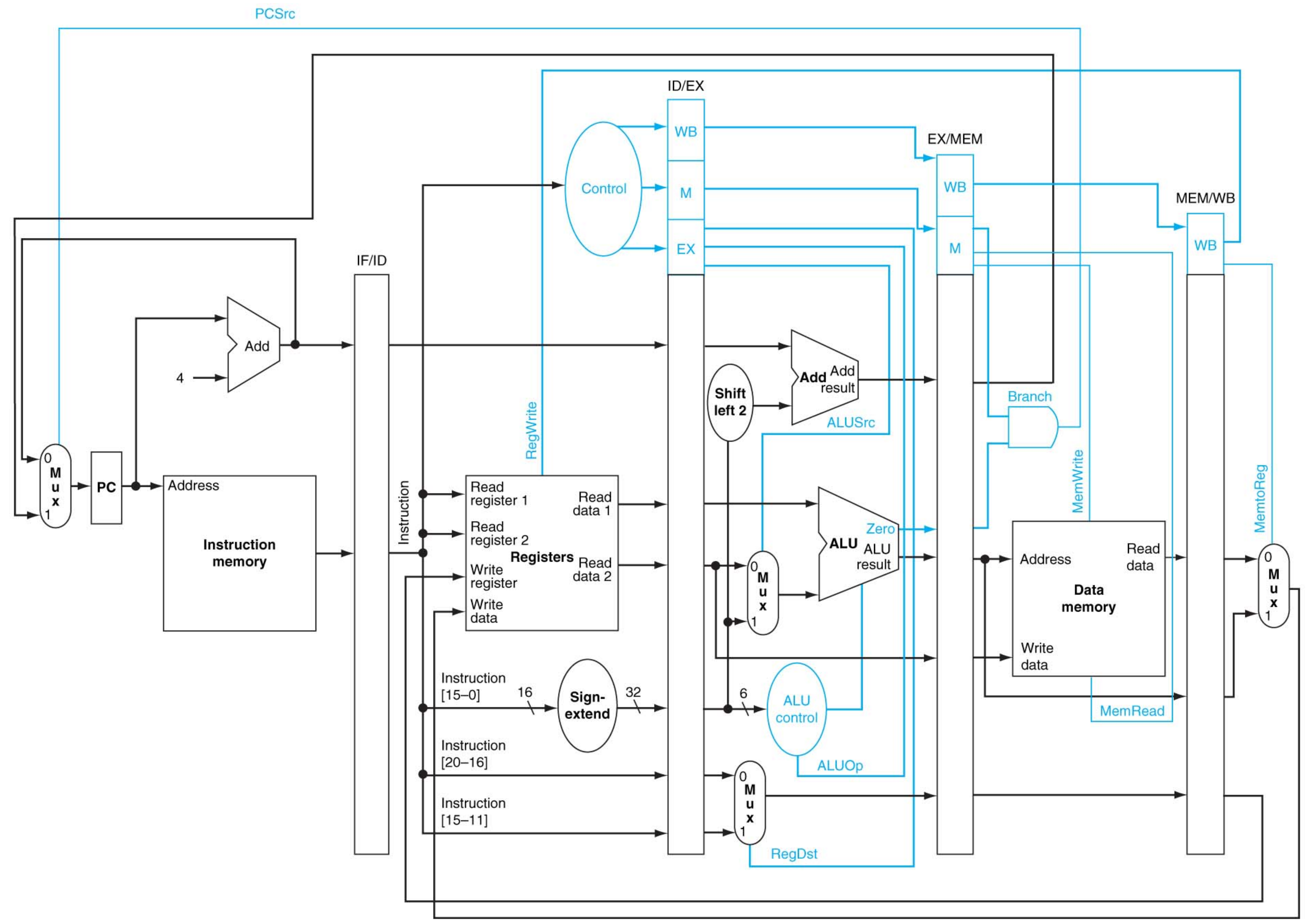
Pipeline Control

- FSM not really appropriate.
- Combinational logic!
    - signals generated once, but follow instruction through the pipeline

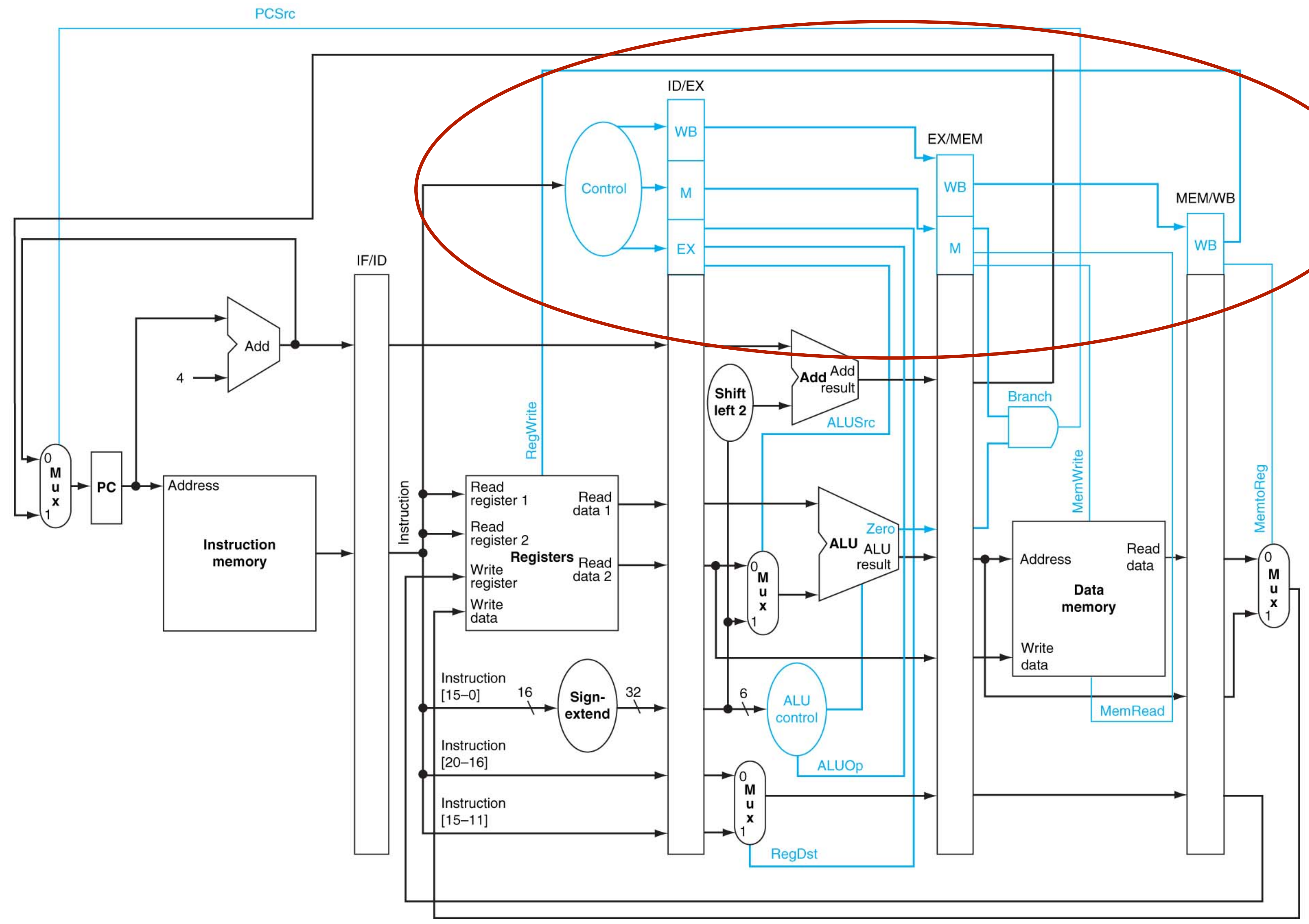
Pipeline Control



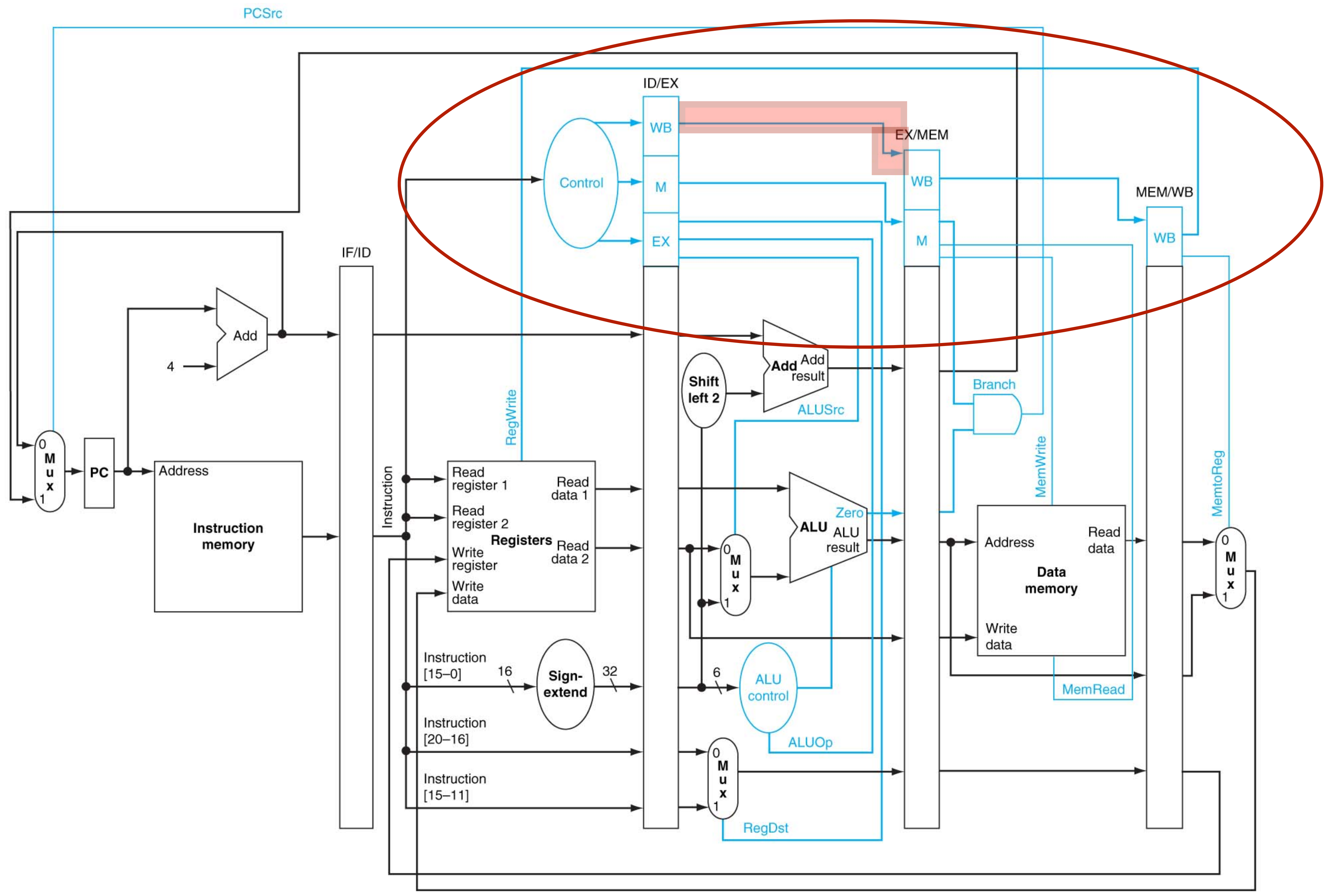
Pipeline with Control Logic



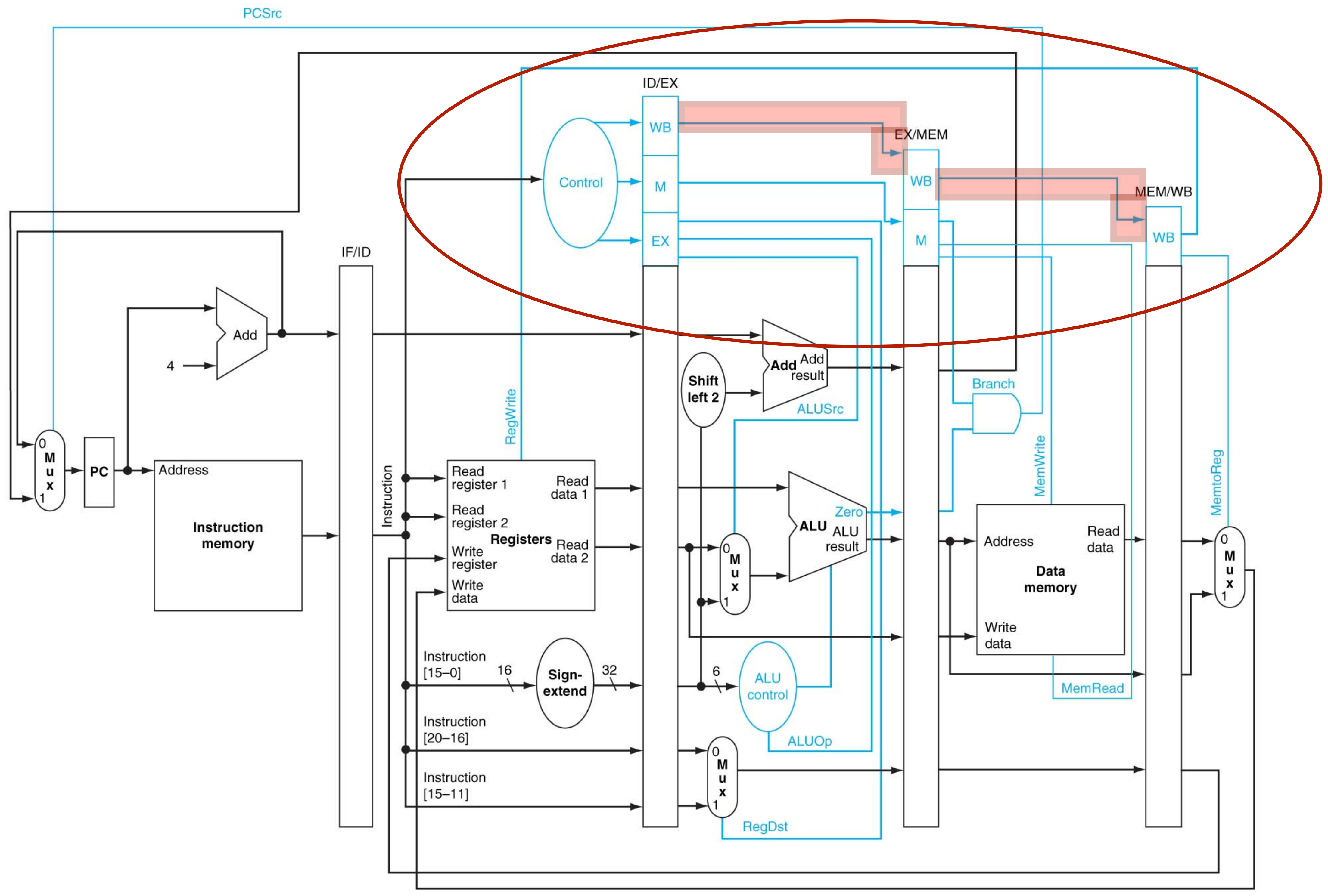
Pipeline with Control Logic



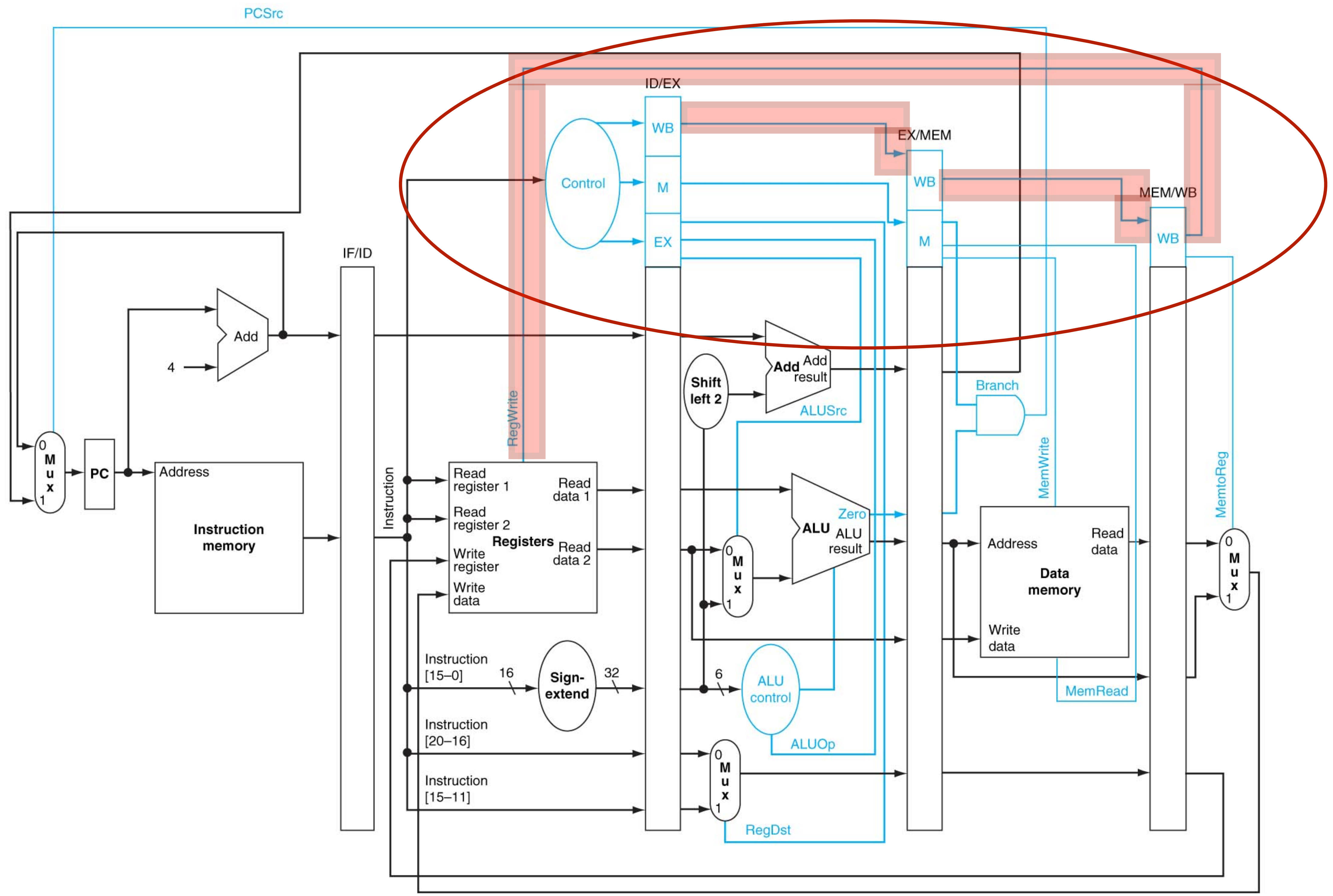
Pipeline with Control Logic



Pipeline with Control Logic



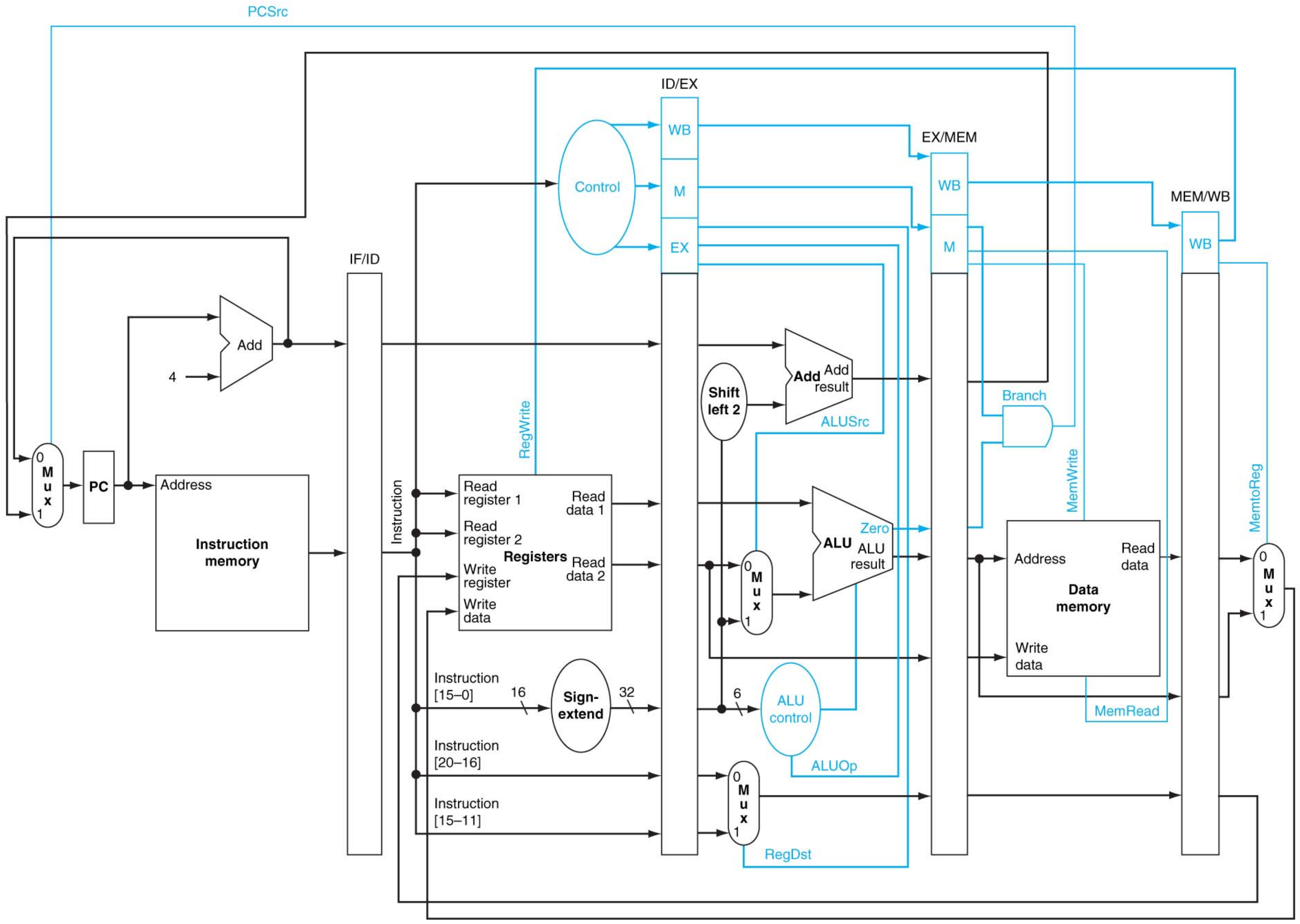
Pipeline with Control Logic



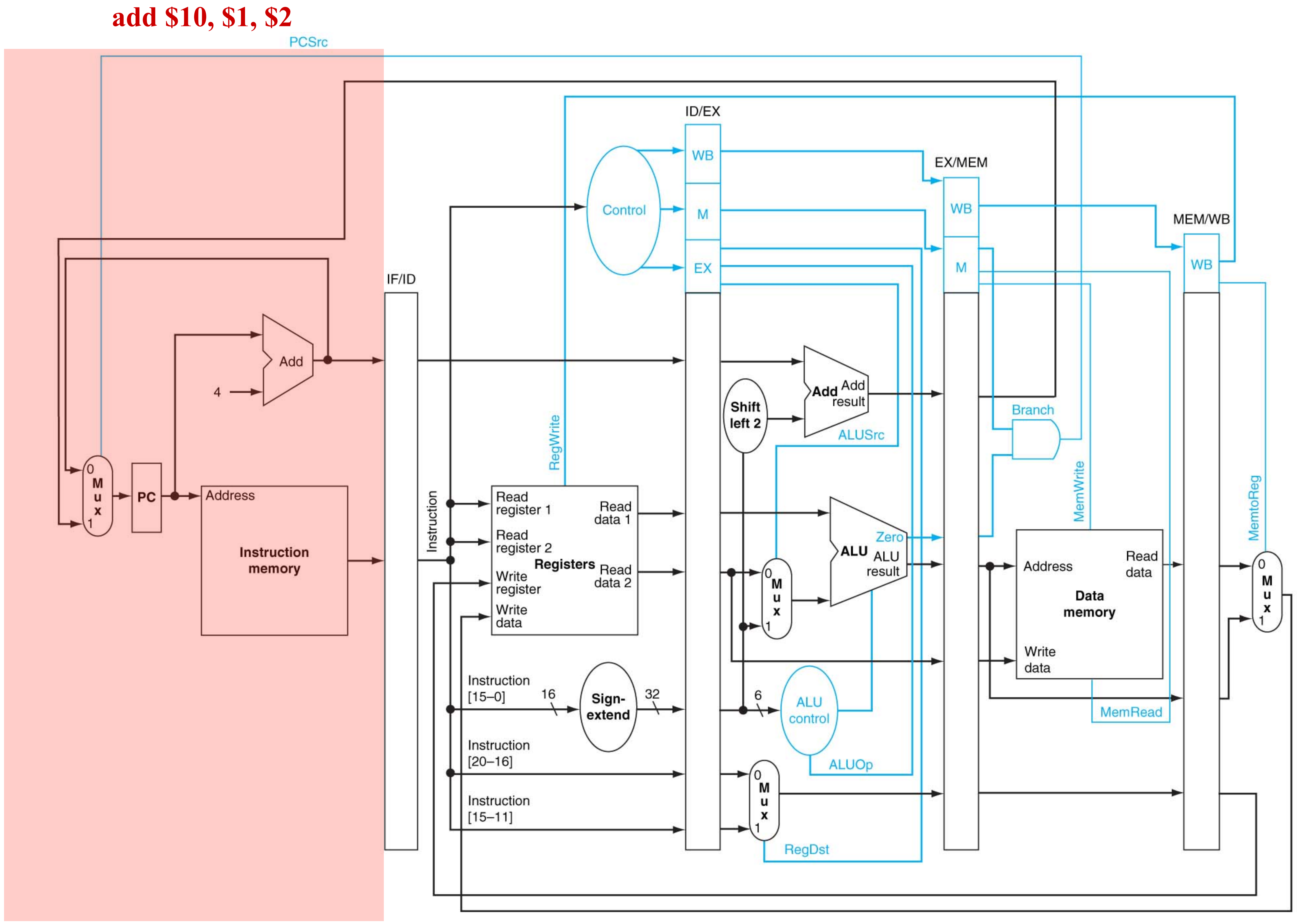
Pipeline Control Signals

	Execution Stage Control Lines				Memory Stage Control Lines			Write Back Stage Control Lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
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

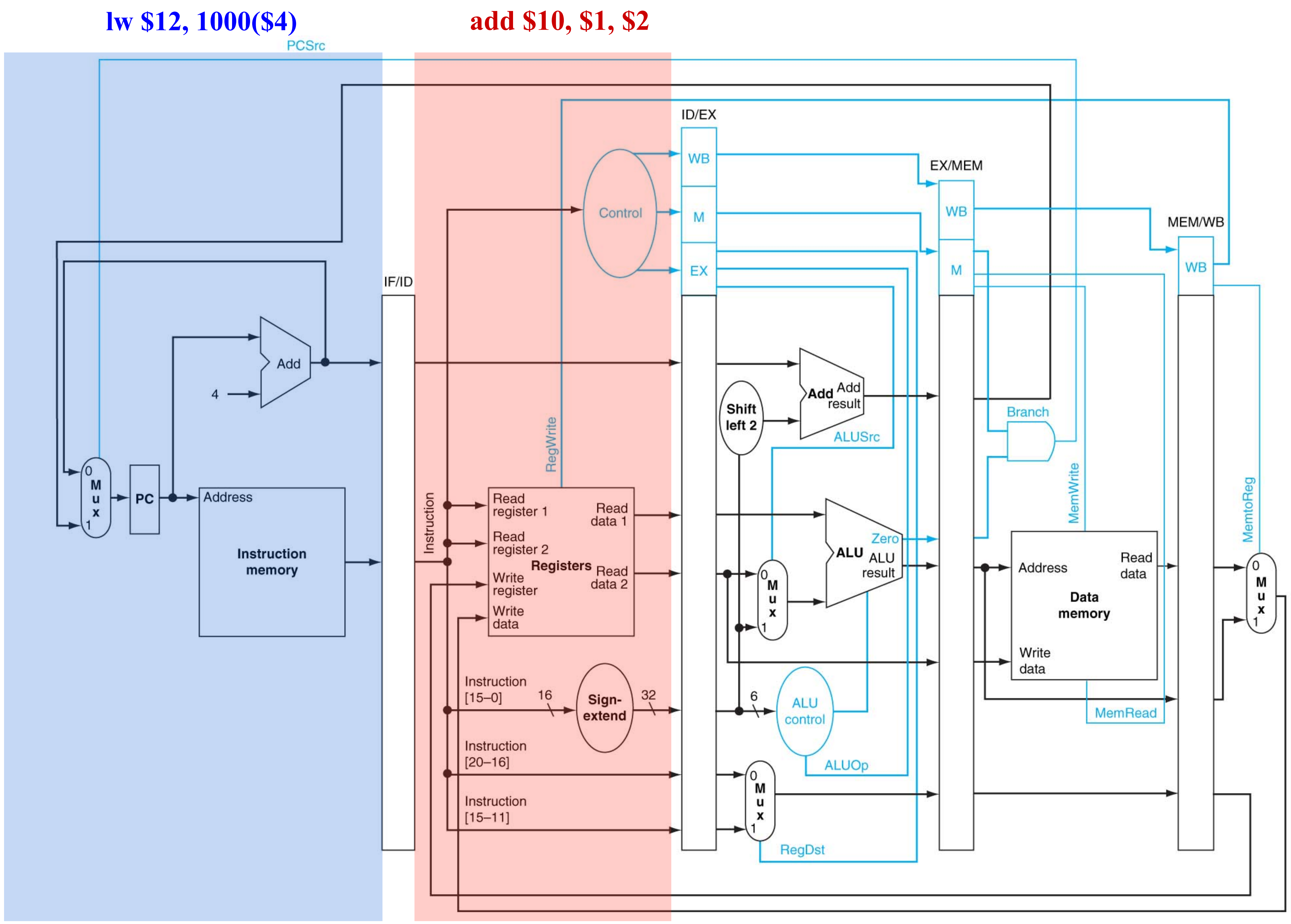
Guess the Signal



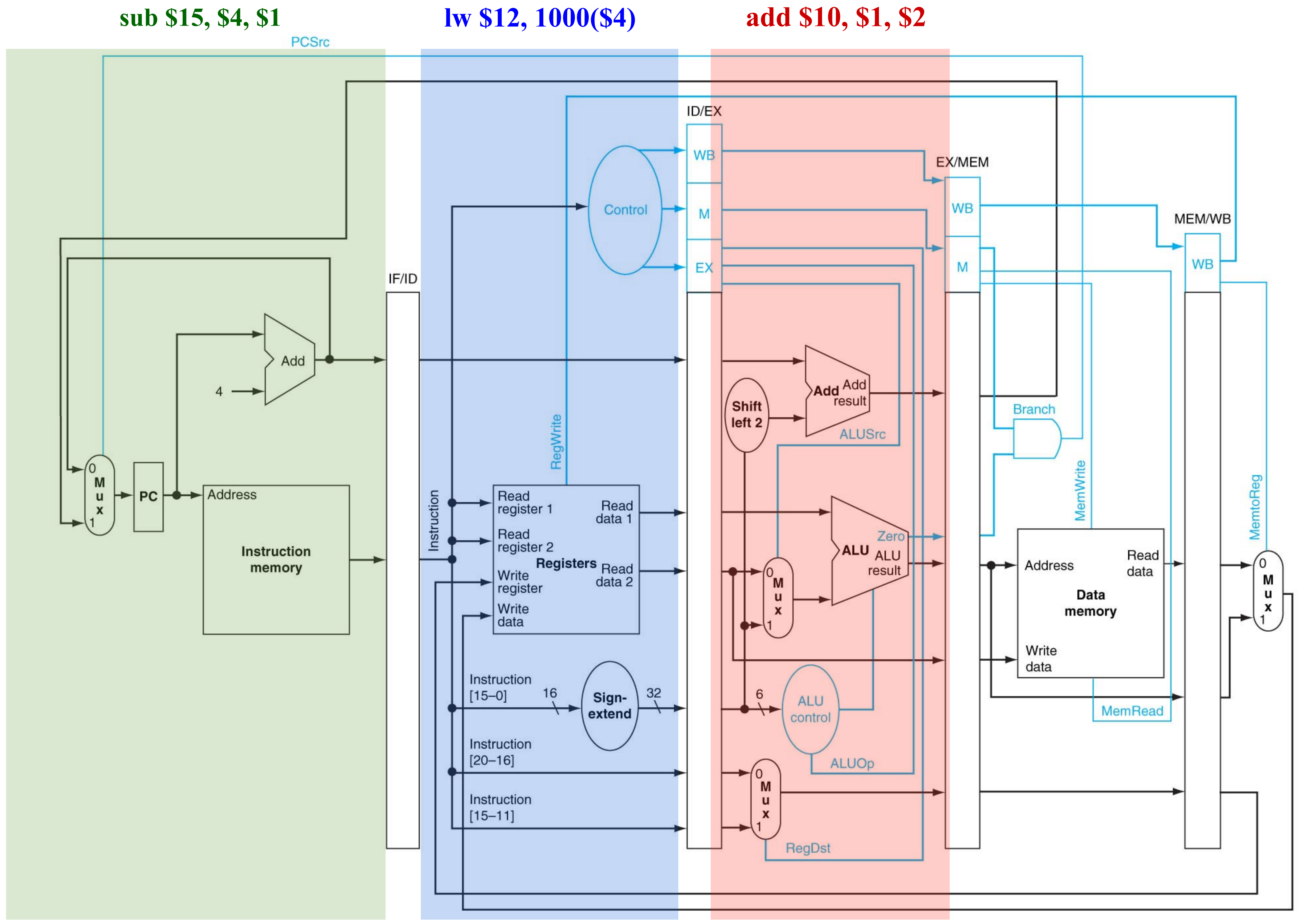
Guess the Signal



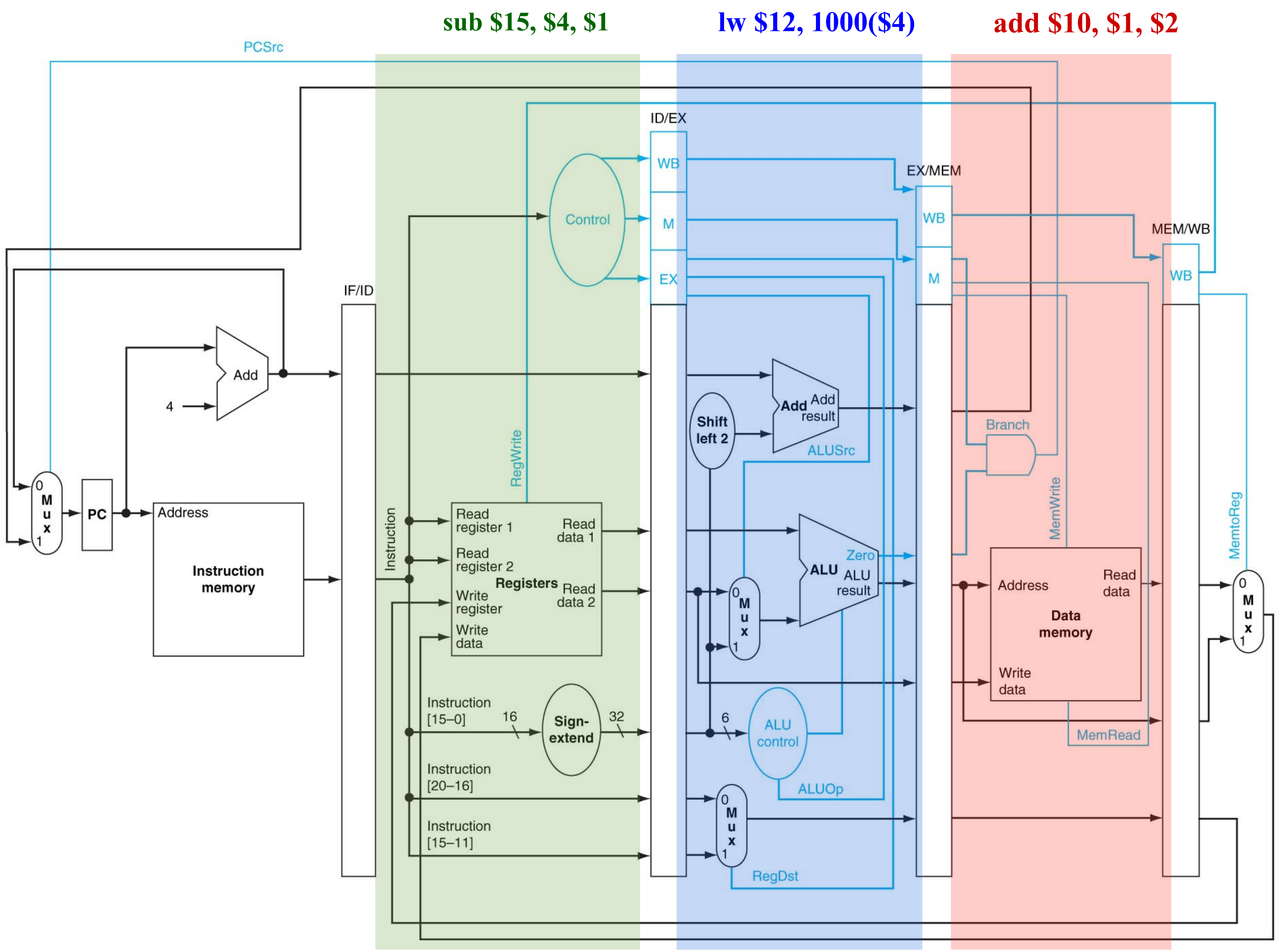
Guess the Signal



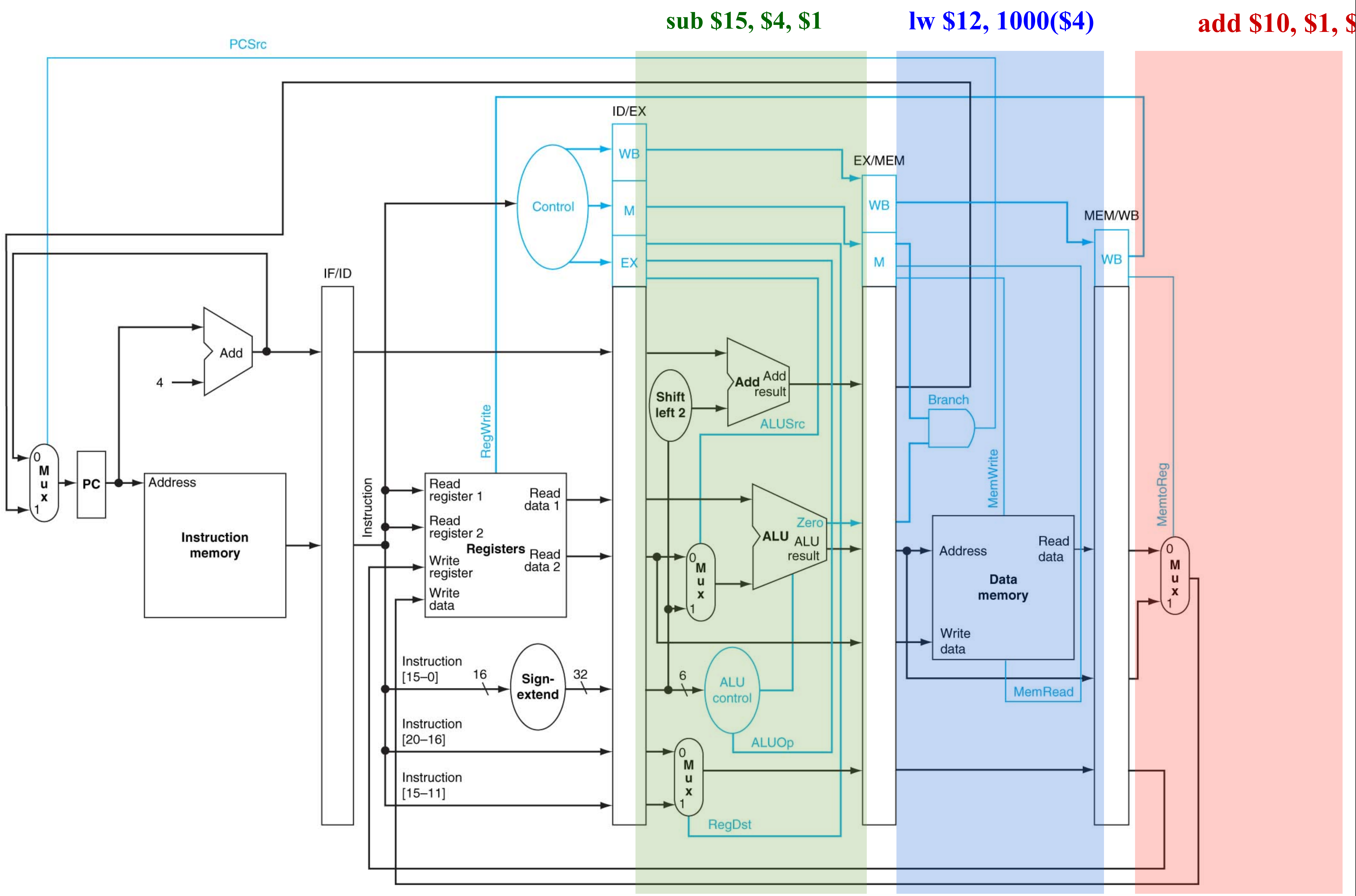
Guess the Signal



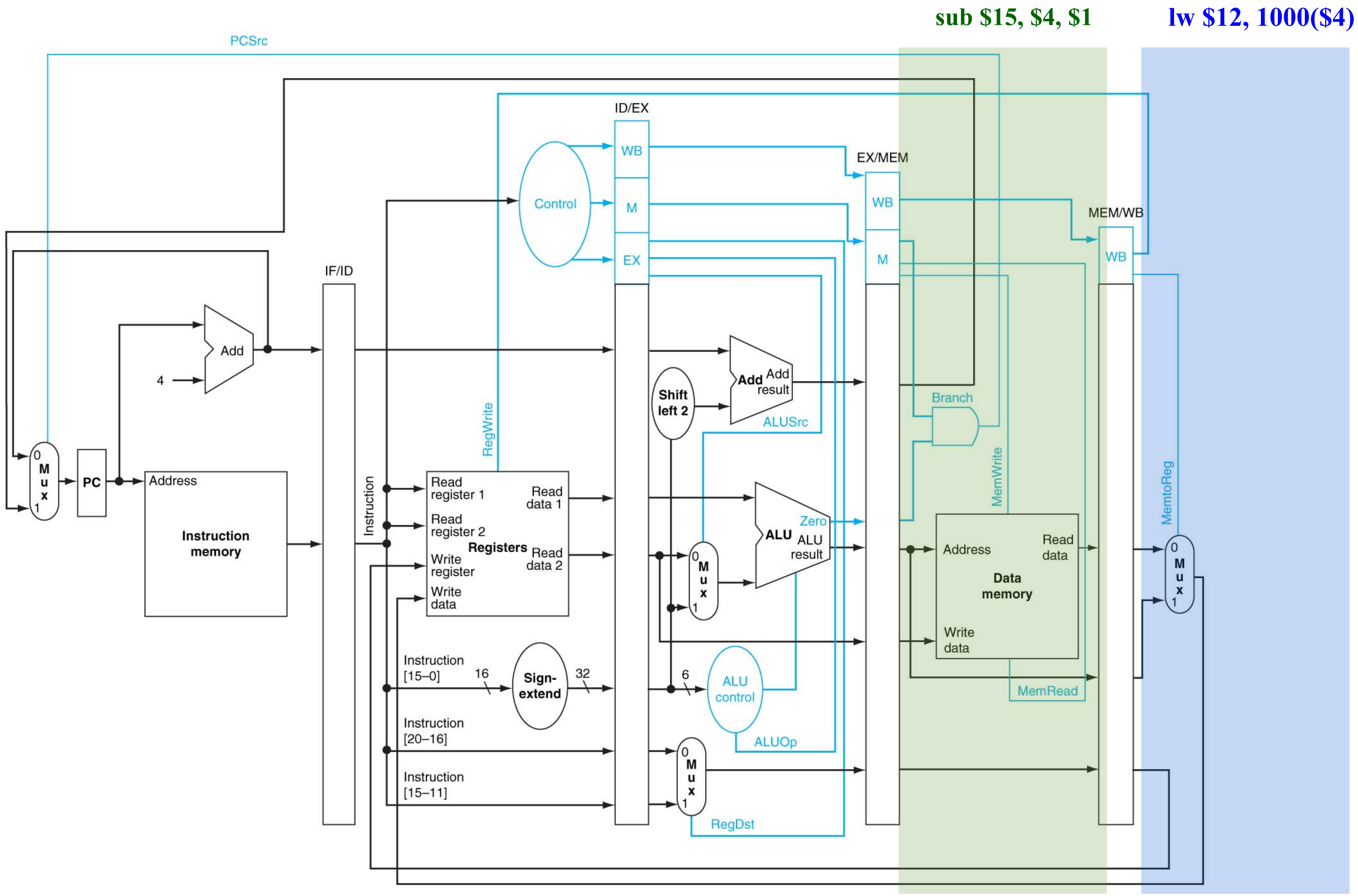
Guess the Signal



Guess the Signal

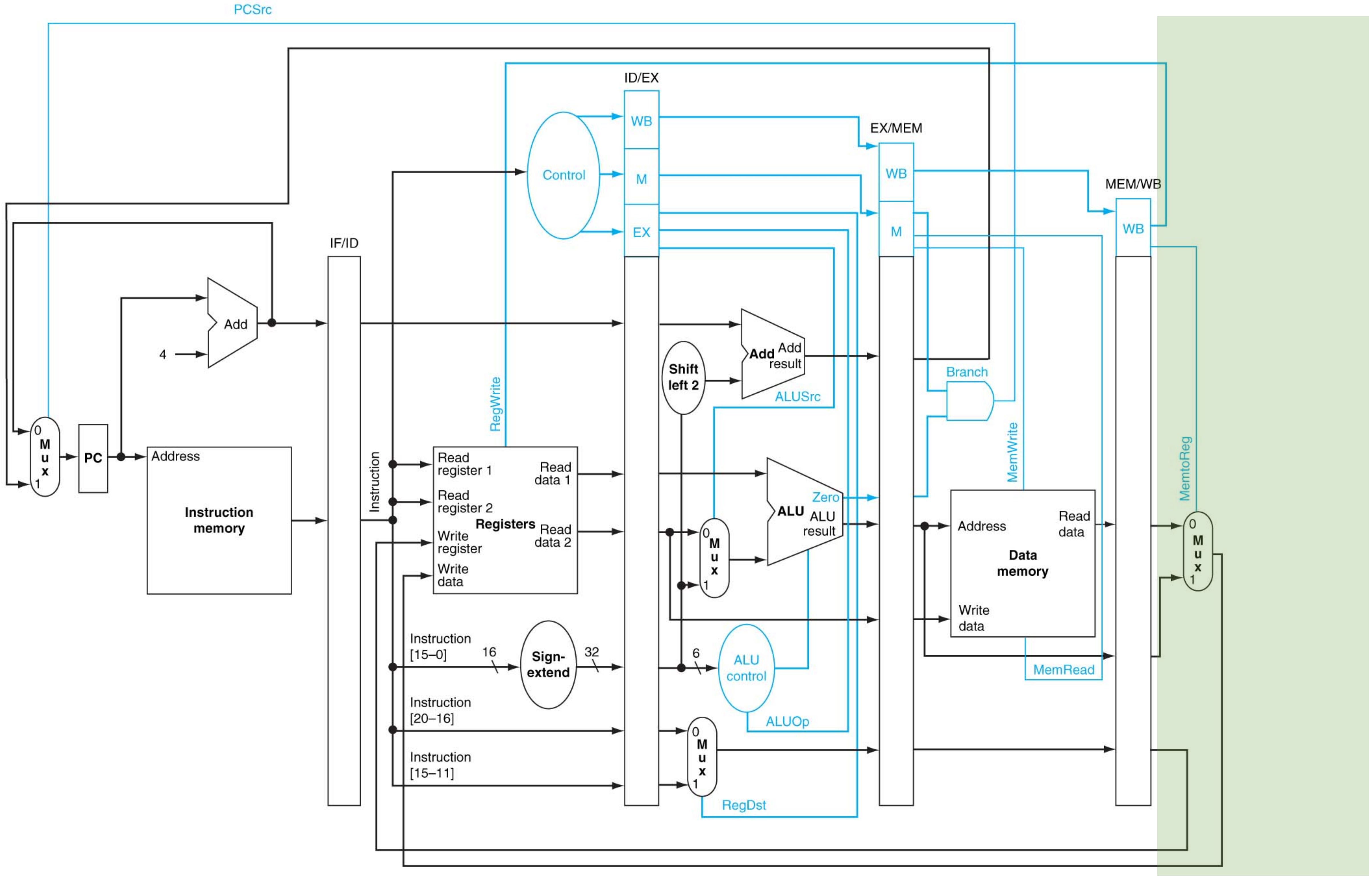


Guess the Signal

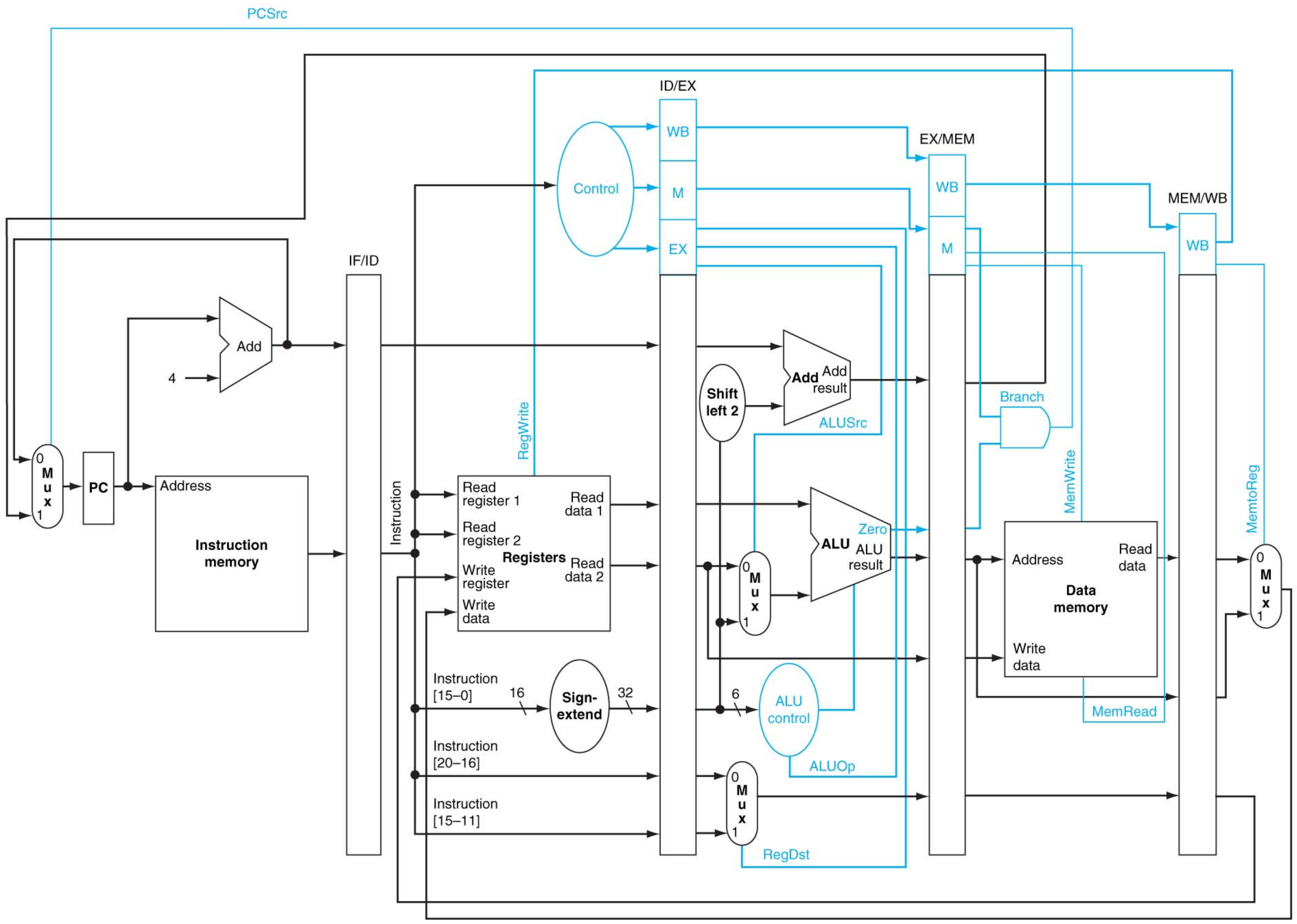


Guess the Signal

sub $15, $4, $1

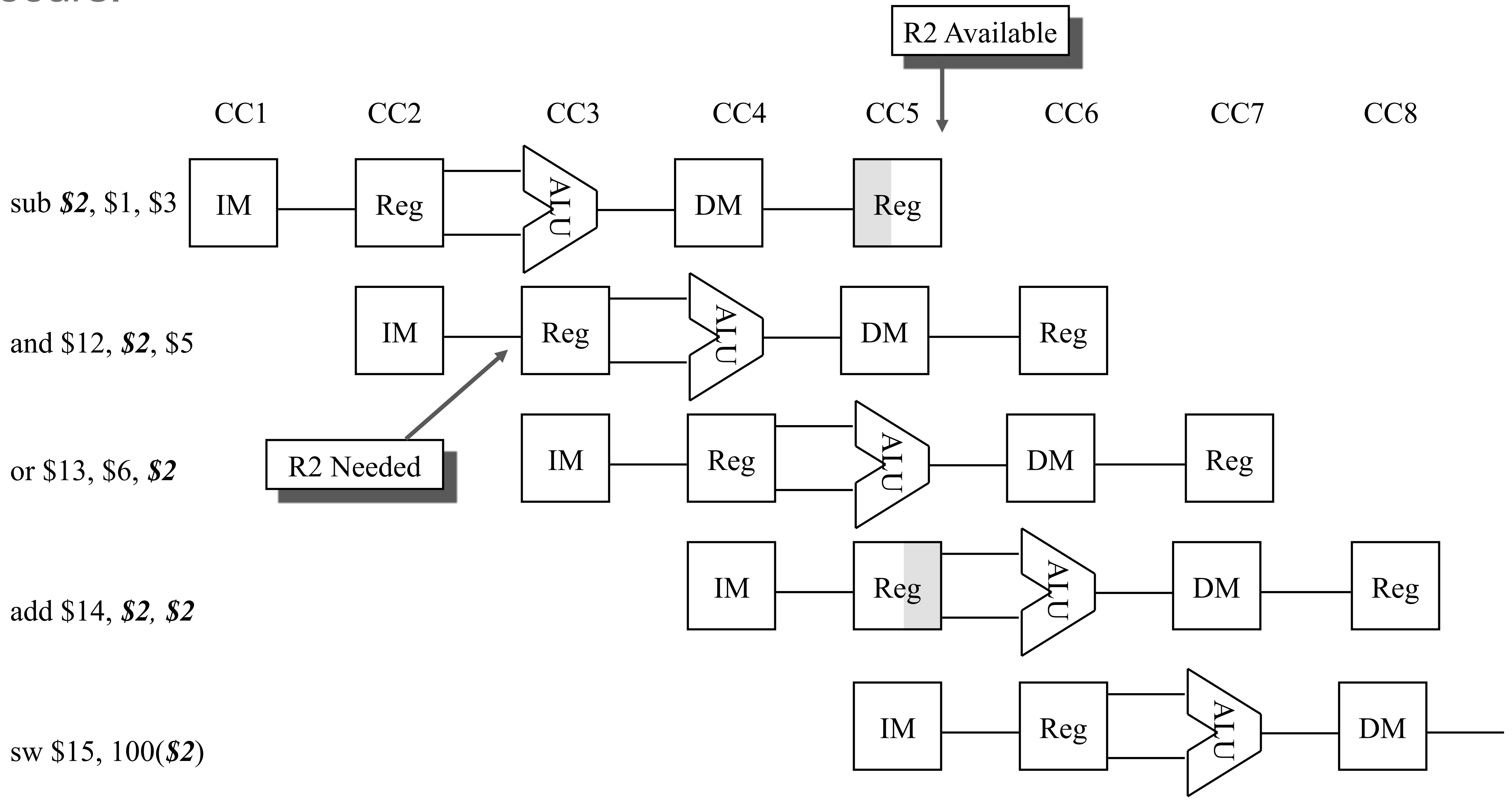


Guess the Signal



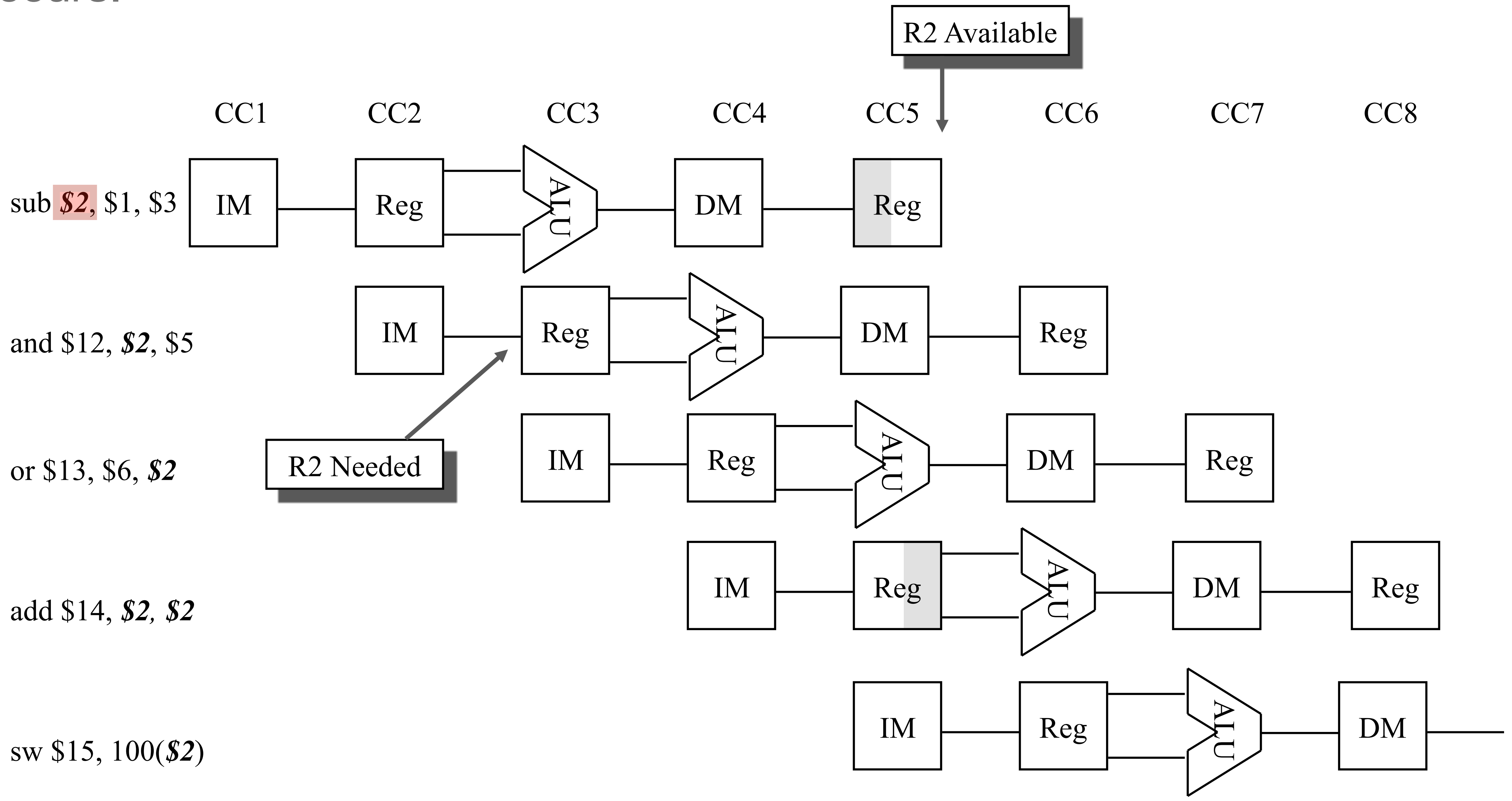
Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.



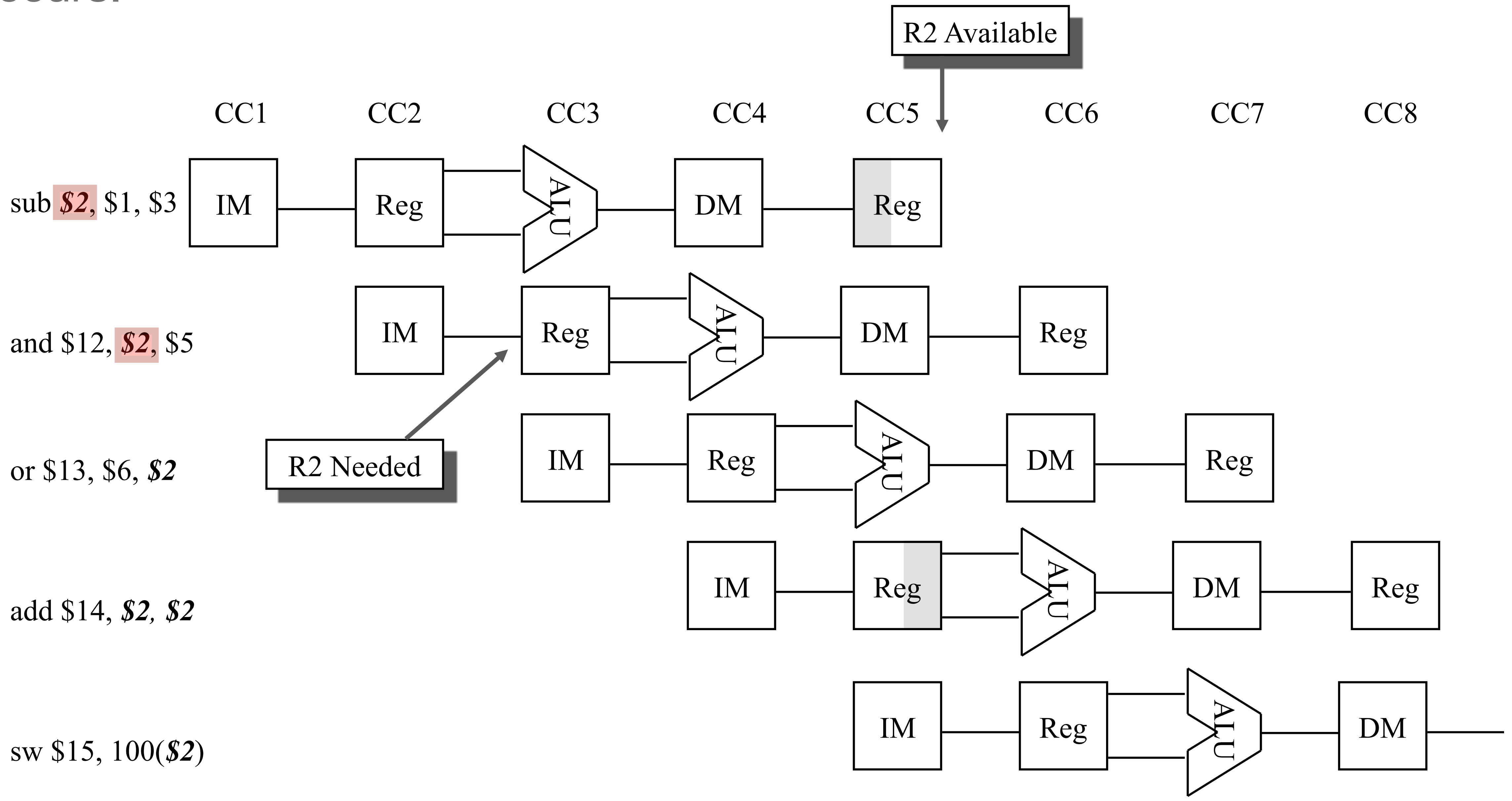
Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.



Data Hazards

- When a result is needed in the pipeline before it is available, a "data hazard" occurs.



Key Points

Key Points

- We achieve high throughput without reducing instruction latency.

Key Points

- We achieve high throughput without reducing instruction latency.
- Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).

Key Points

- We achieve high throughput without reducing instruction latency.
- Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).
- Pipelining uses combinational logic to generate (and registers to propagate) control signals.

Key Points

- We achieve high throughput without reducing instruction latency.
- Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).
- Pipelining uses combinational logic to generate (and registers to propagate) control signals.
- Pipelining creates potential hazards.