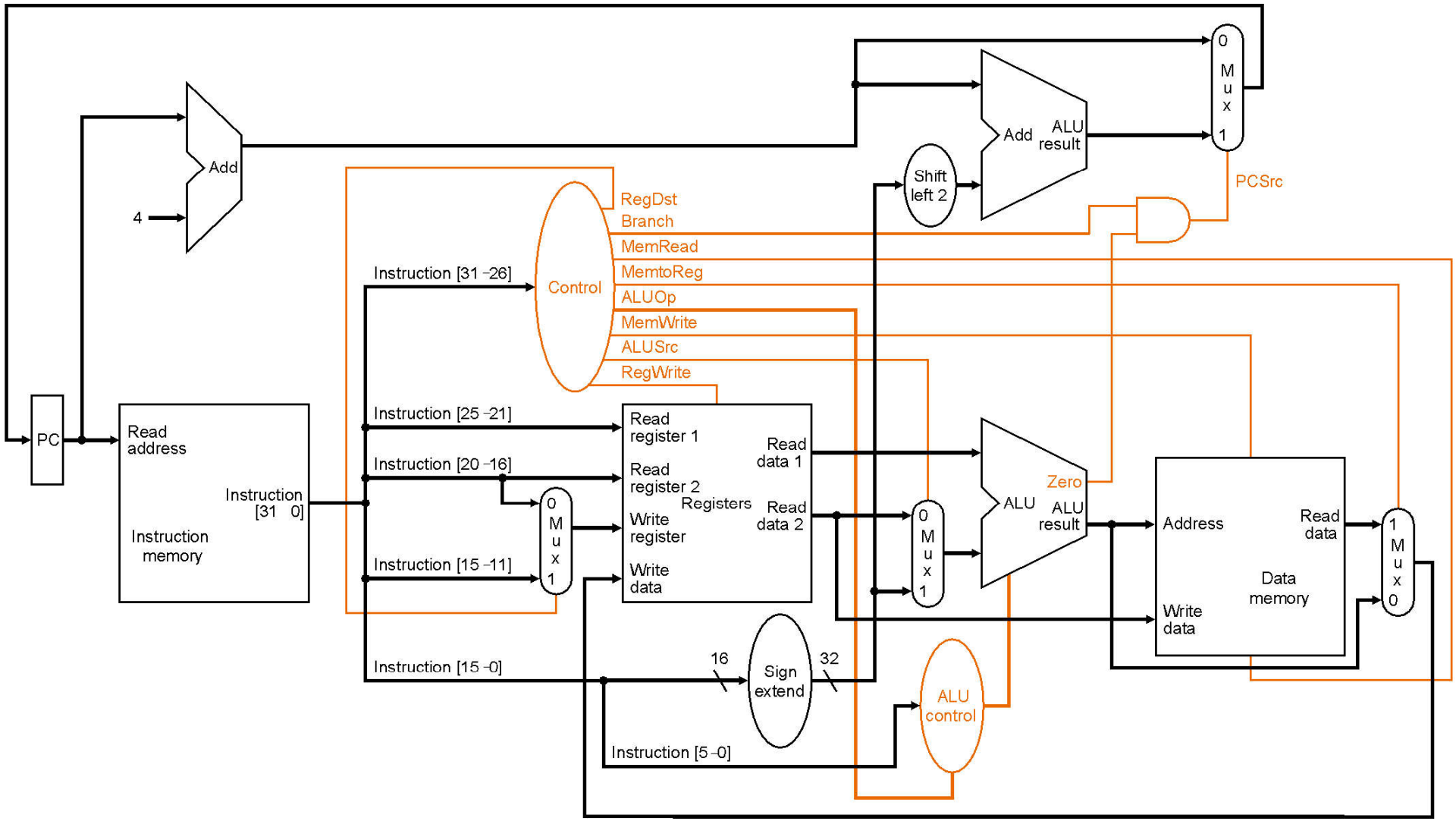
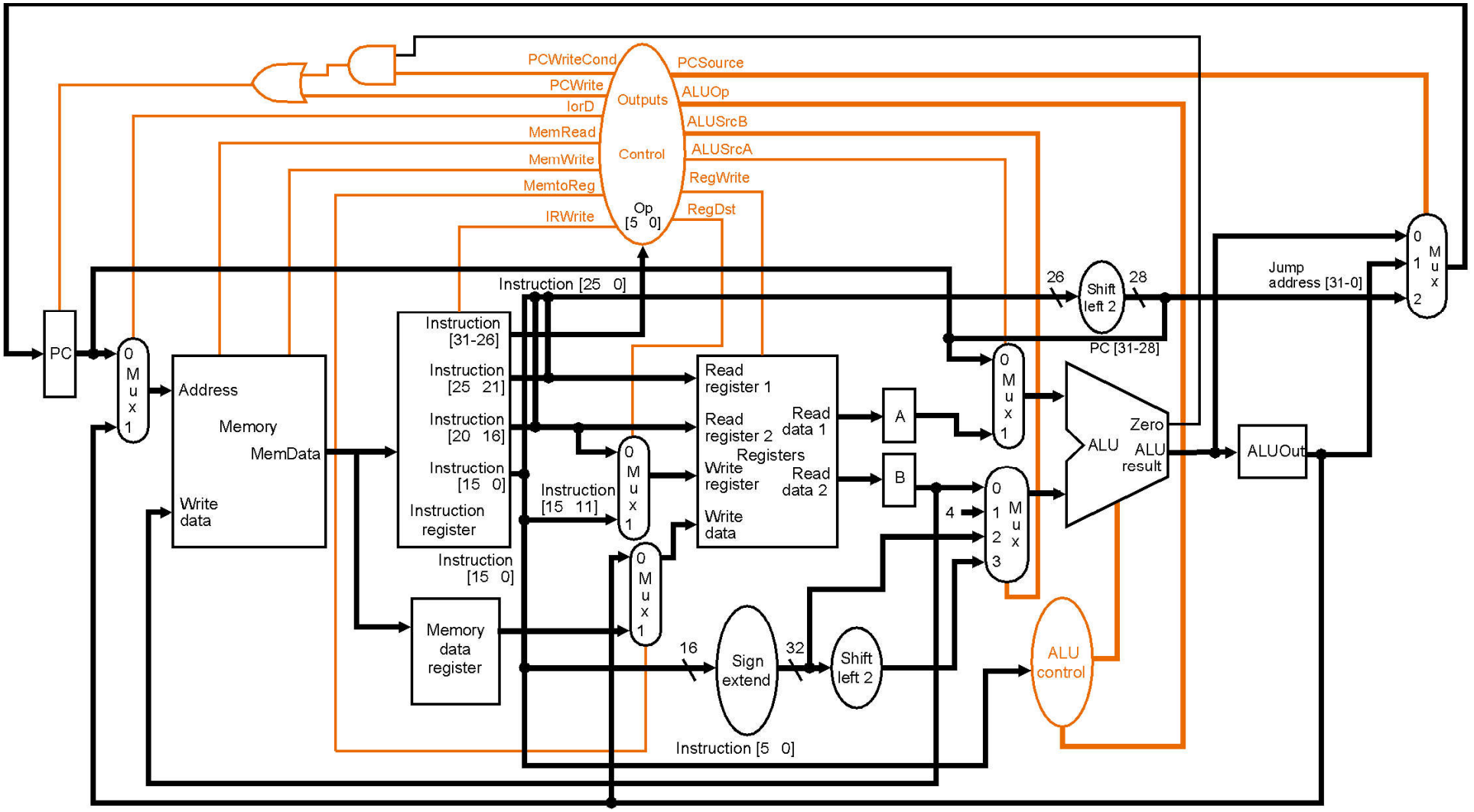
Designing a Pipelined CPU

Review -- Single Cycle CPU

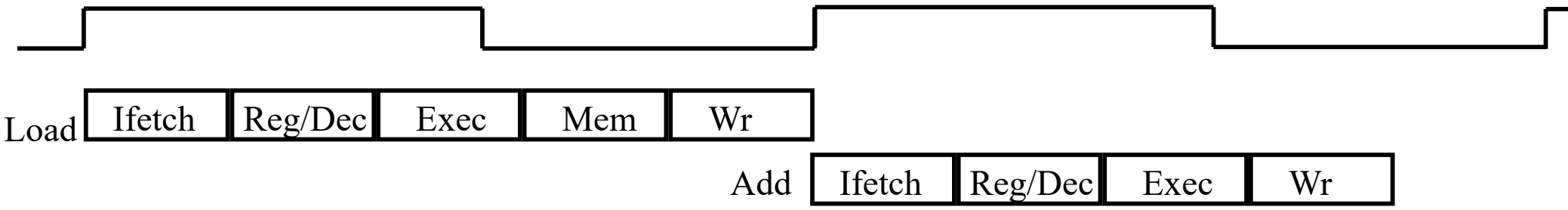


(not quite) Review -- Multiple Cycle CPU

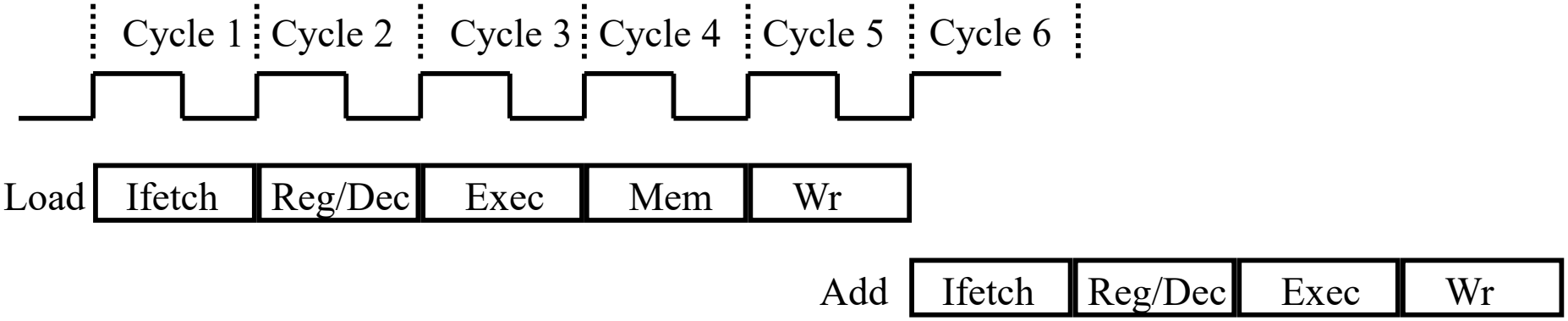


Review -- Instruction Latencies

•Single-Cycle CPU

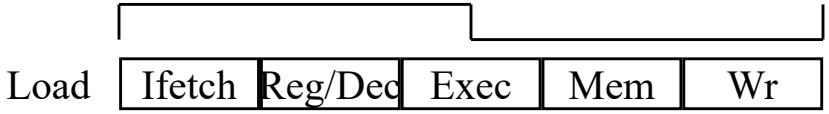


•Multiple Cycle CPU

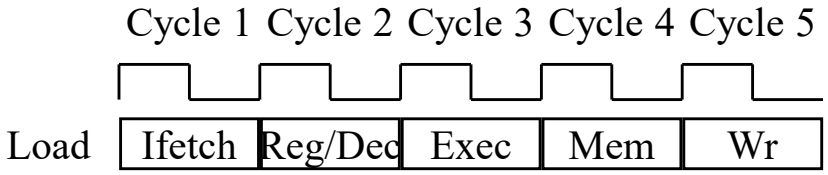


Instruction Latencies and Throughput

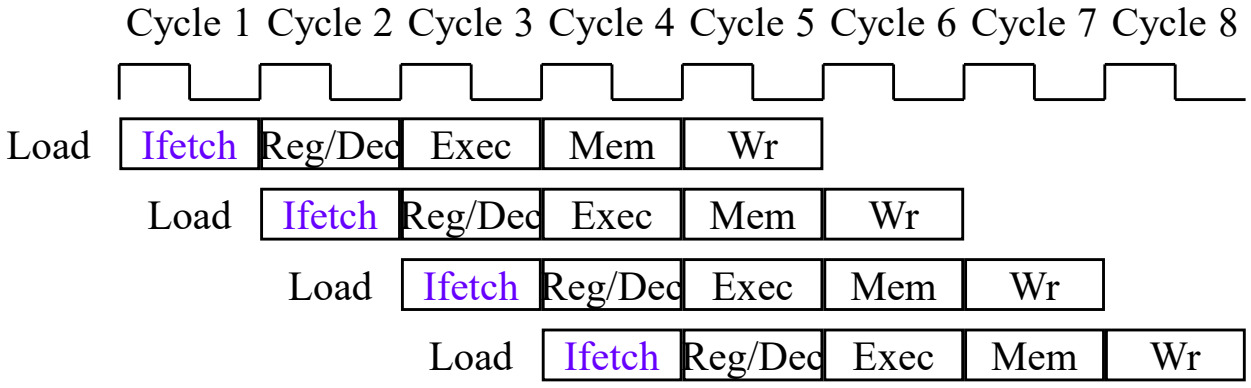
•Single-Cycle CPU



•Multiple Cycle CPU



•Pipelined CPU



Pipelining Advantages

- Higher *maximum* throughput
- Higher *utilization* of CPU resources

- But, more complicated *datapath*, more complex control(?)

Pipelining Advantages

CPU Design Technology	***Control Logic***	***Peak Throughput***
Single-Cycle CPU	Combinational Logic	$\frac{1}{\quad}$
Multiple-Cycle CPU	Messy (state machine)	$\frac{1}{\quad}$
Pipelined CPU		$\frac{1}{\quad}$

Pipelining in Modern CPUs

- CPU Datapath
- Arithmetic Units
- System Buses
- Software (at multiple levels)
- etc...

A Pipelined Datapath

IF: Instruction fetch

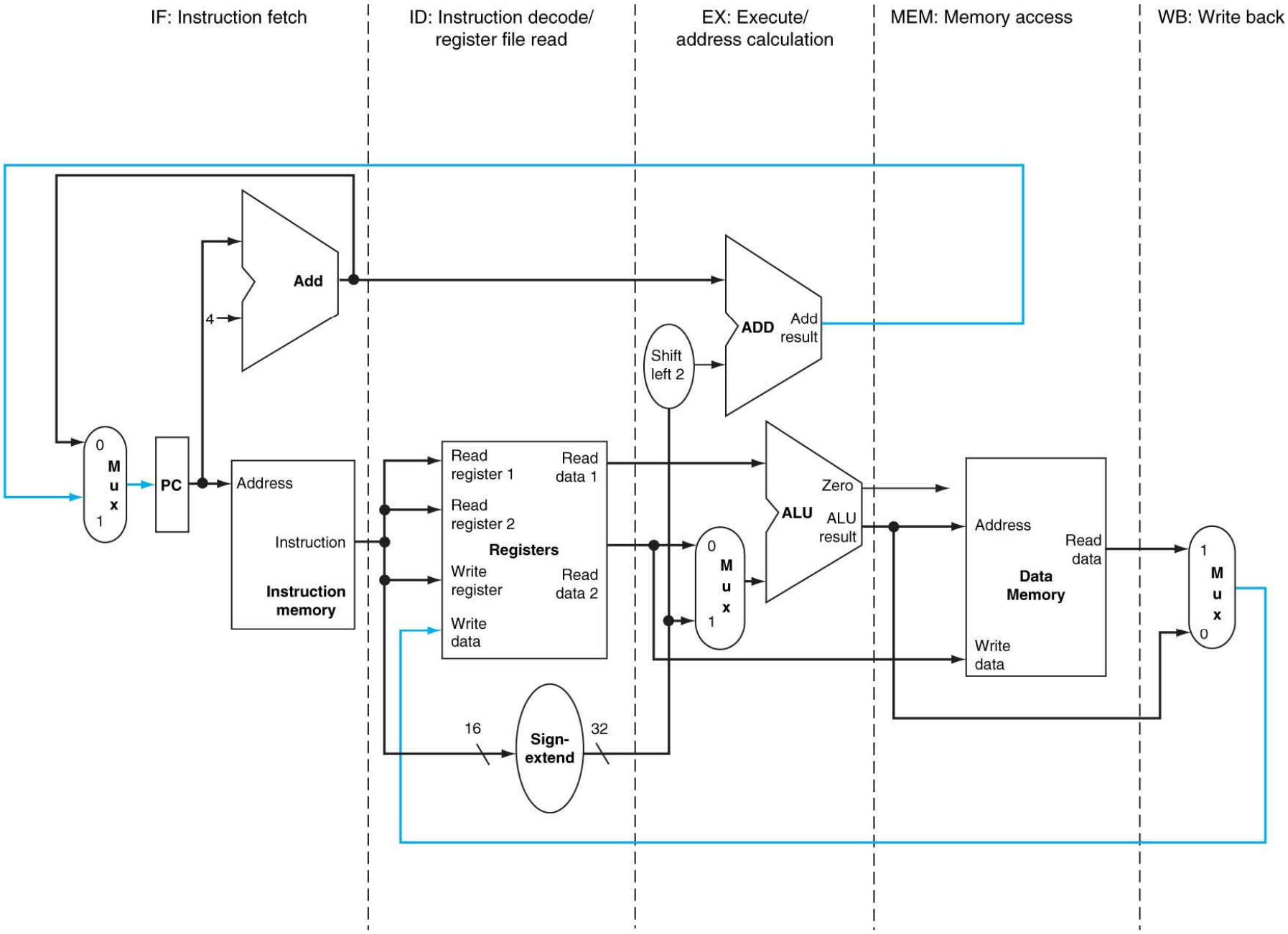
ID: Instruction decode and register fetch

EX: Execution and effective address calculation

MEM: Memory access

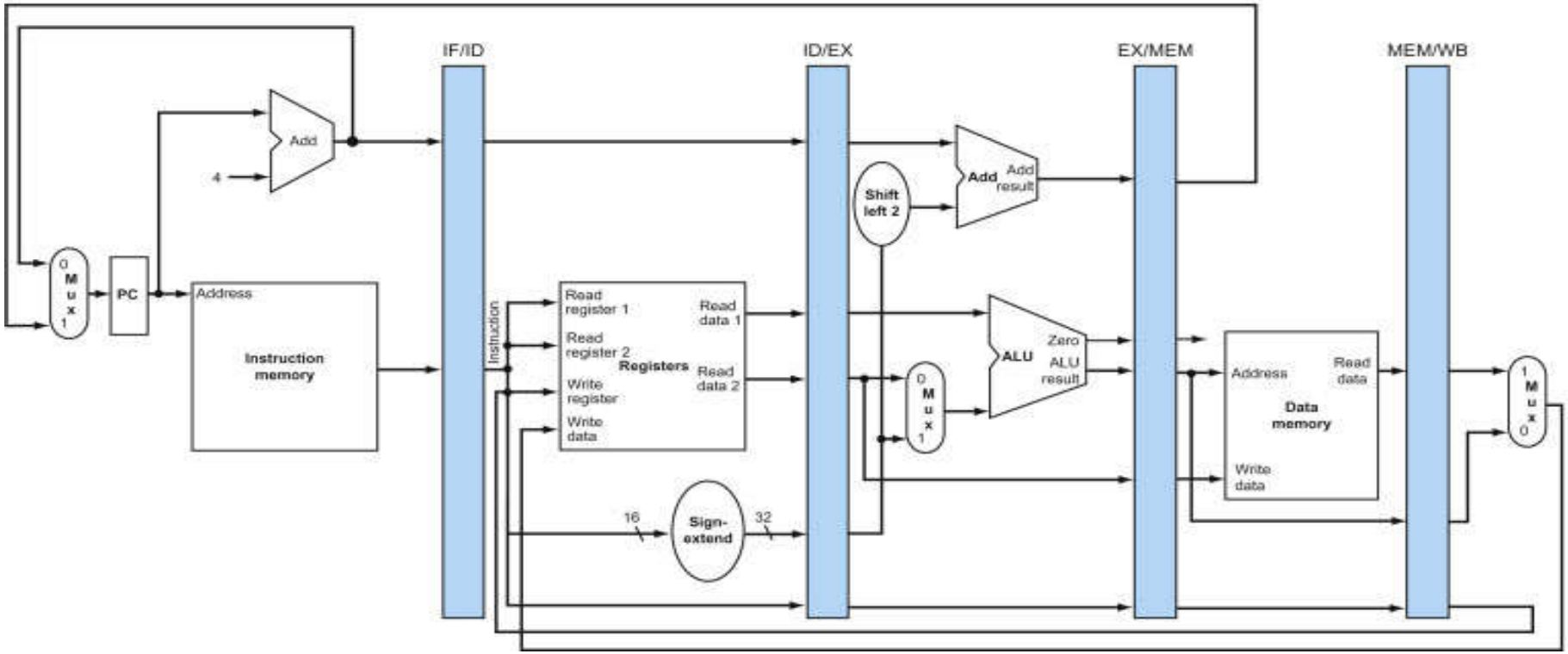
WB: Write back

Pipelined Datapath (roughly)

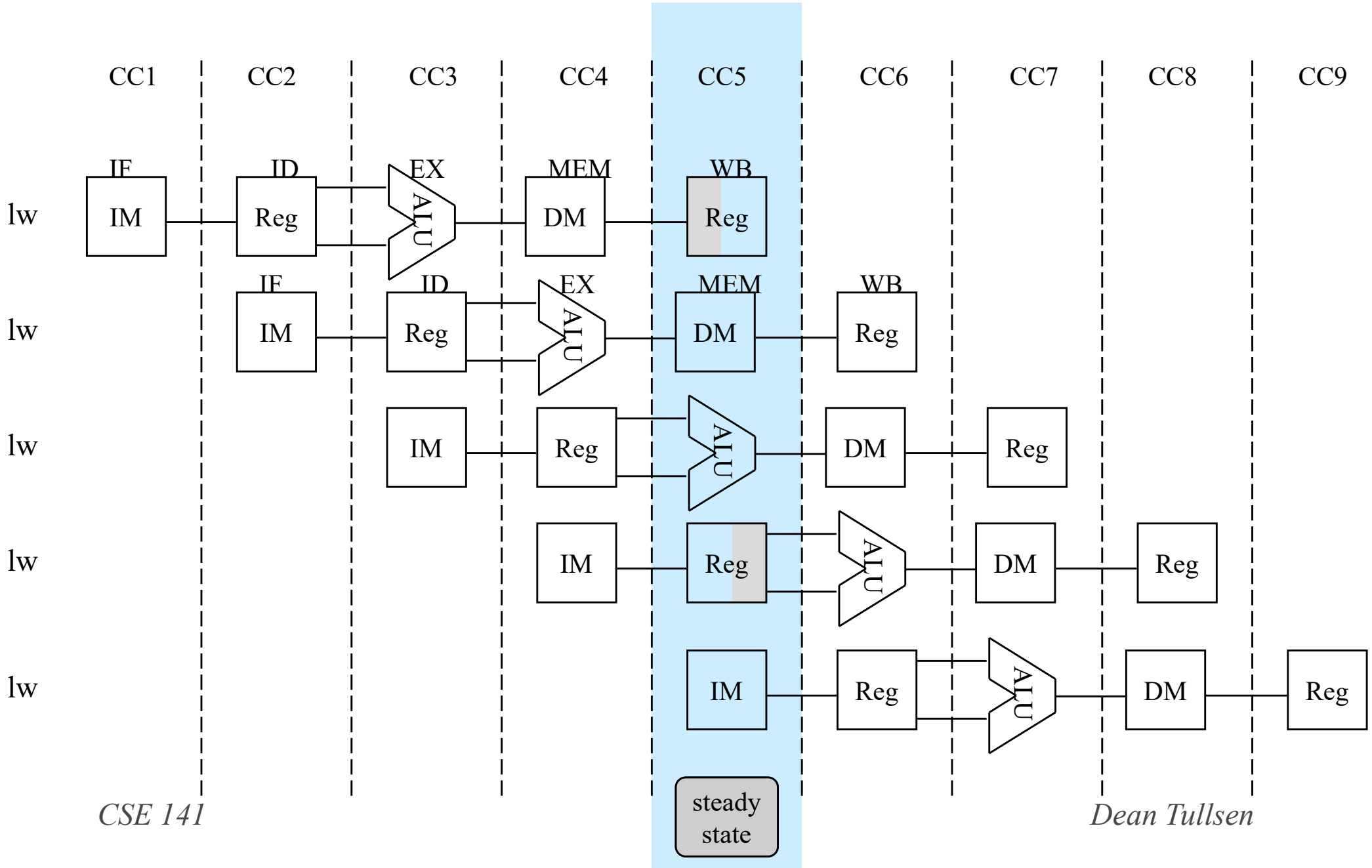


Pipelined Datapath

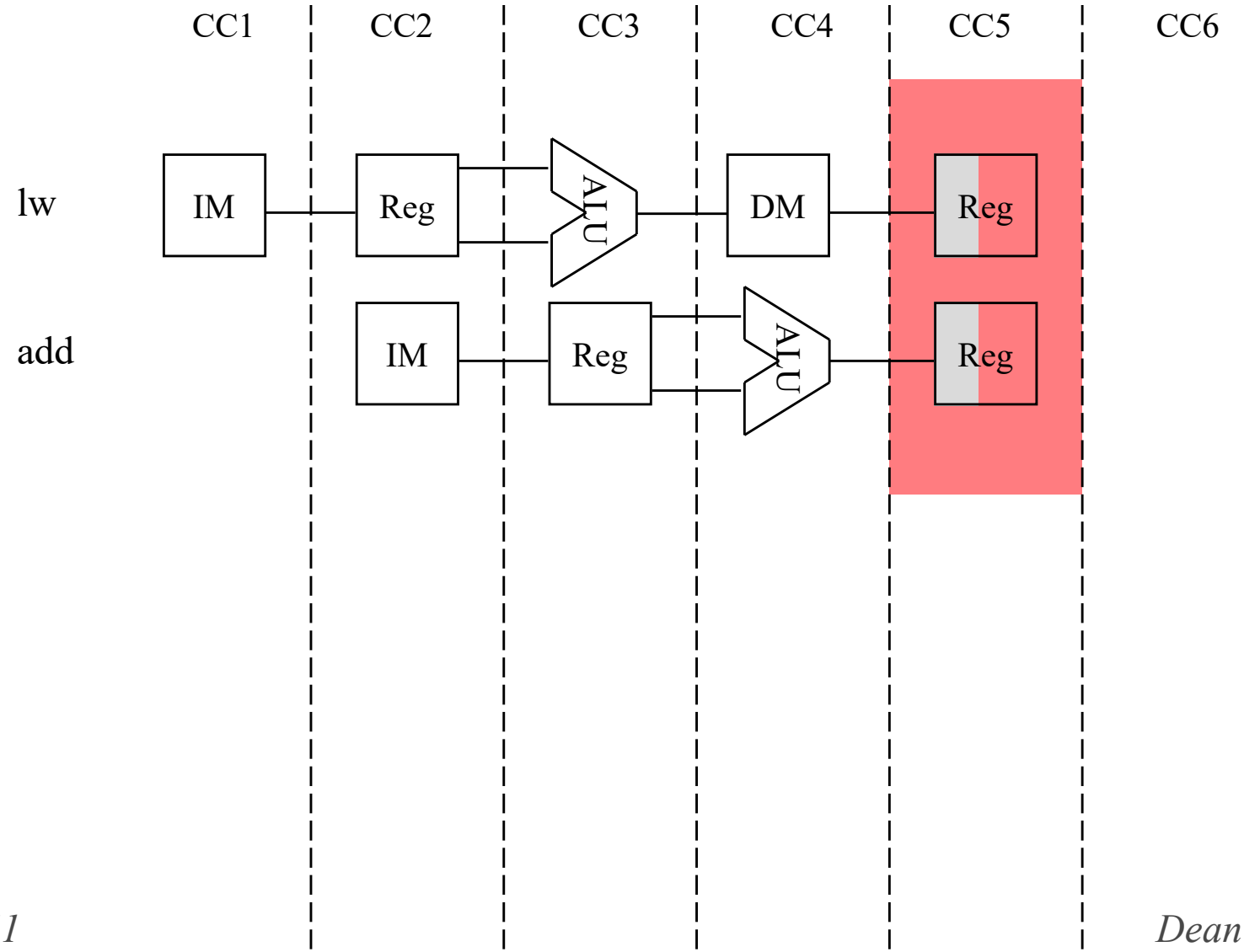
Instruction Fetch | Instruction Decode/ Register Fetch | Execute/ Address Calculation | Memory Access | Write Back



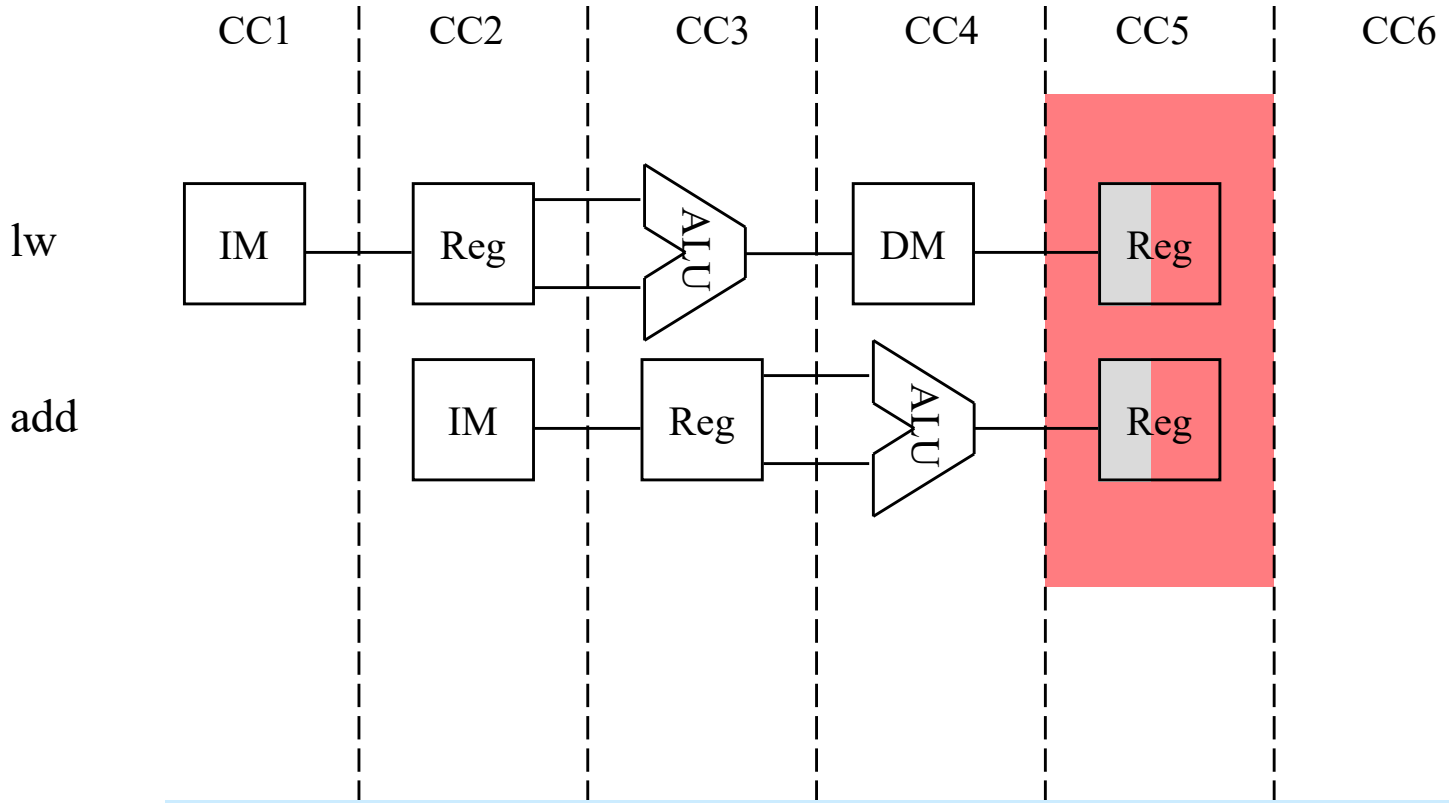
Execution in a Pipelined Datapath



Mixed Instructions in the Pipeline



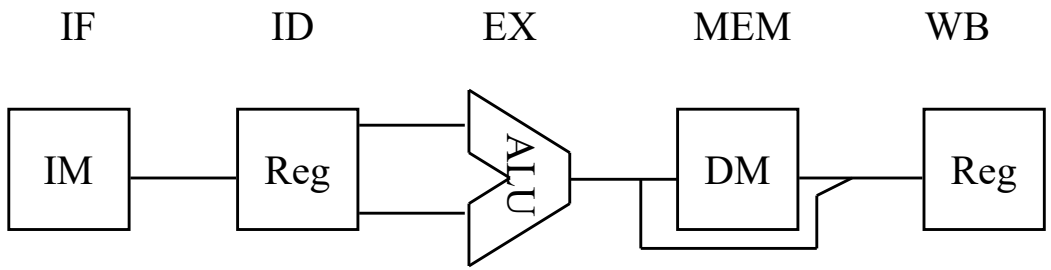
Mixed Instructions in the Pipeline



This is called a structural hazard – too many instructions want to use the same resource.
In our pipeline, we can make this hazard disappear (next slide). In more complex pipelines, structural hazards are again possible.

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.



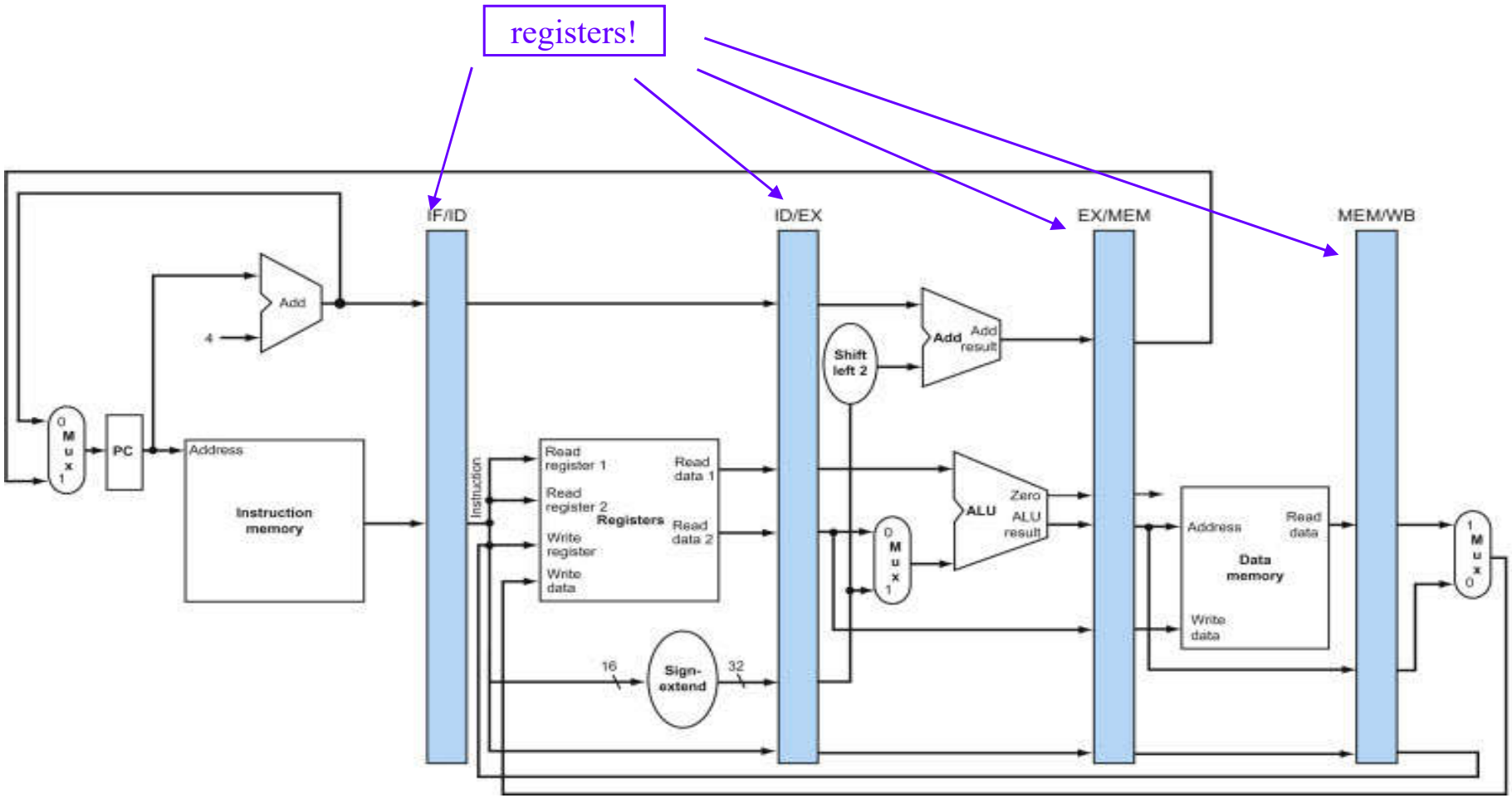
Pipeline Stages

What is the performance implication of making every instruction go through all 5 stages? (e.g., instead of 4 for add, 3 for beq, etc.)

Selection	(Choose BEST answer)
A	Decreases peak throughput by 20%
B	Increases program latency by 20%
C	No significant impact on peak throughput or program latency
D	Depends on how many R-type instructions, beq, etc.
E	None of the above

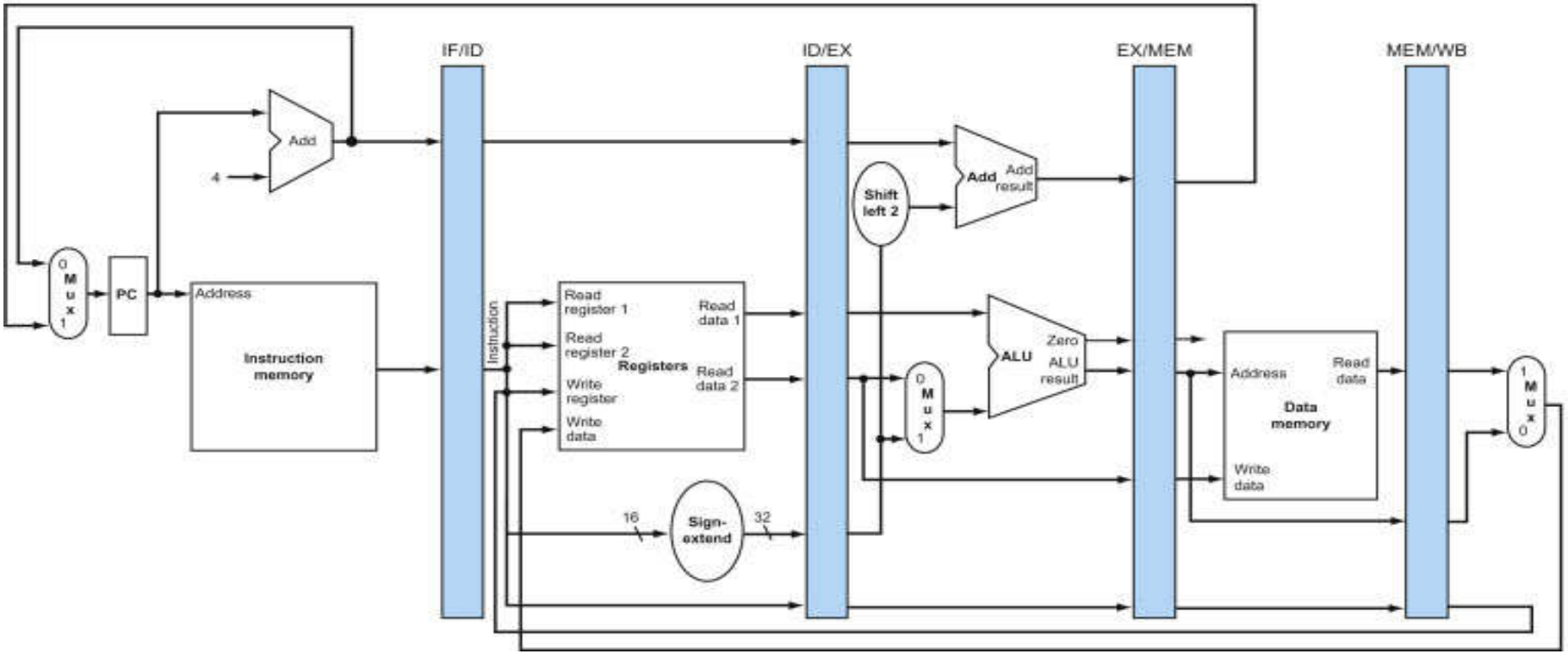
Pipelined Datapath

Instruction Fetch | Instruction Decode/ Register Fetch | Execute/ Address Calculation | Memory Access | Write Back



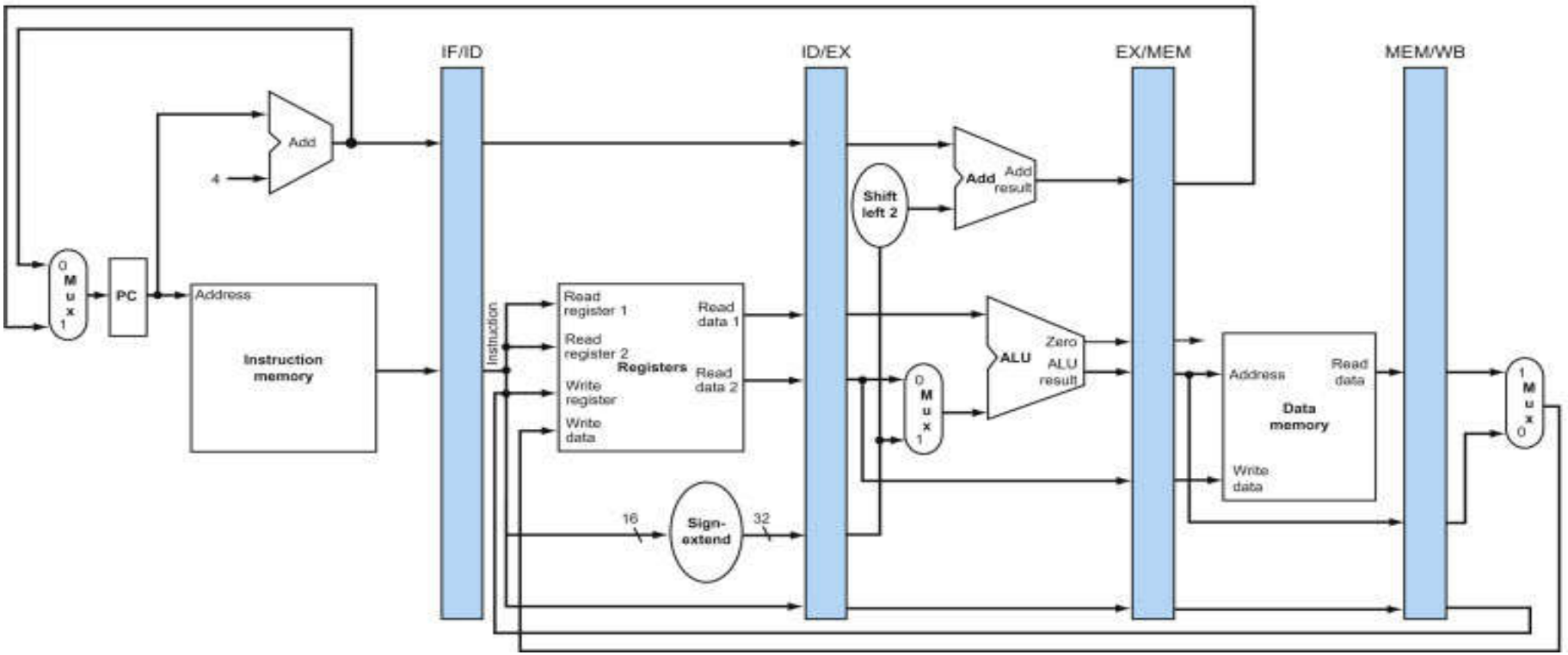
The Pipeline in Execution

add $10, $1, $2 | Instruction Decode/ Register Fetch | Execute/ Address Calculation | Memory Access | Write Back



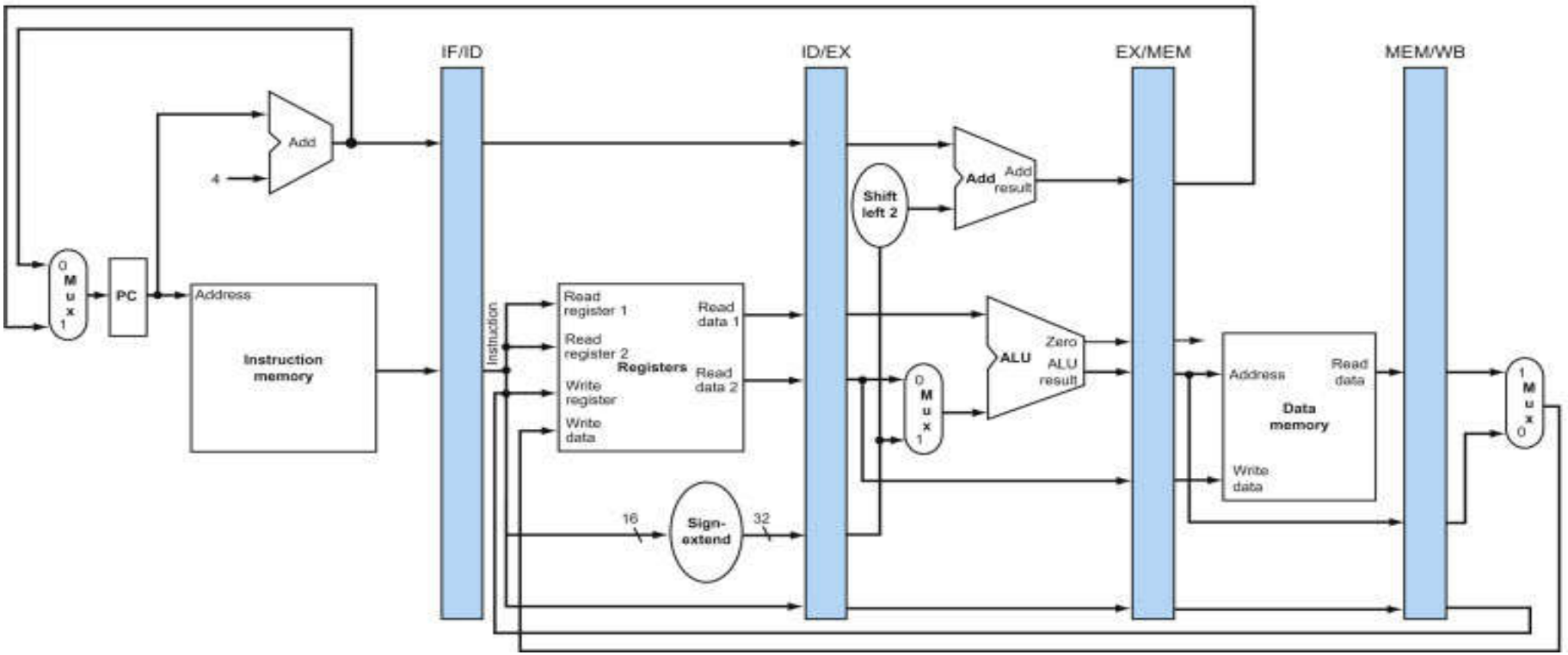
The Pipeline in Execution

lw $12, 1000($4) **add $10, $1, $2** Execute/ Address Calculation Memory Access Write Back



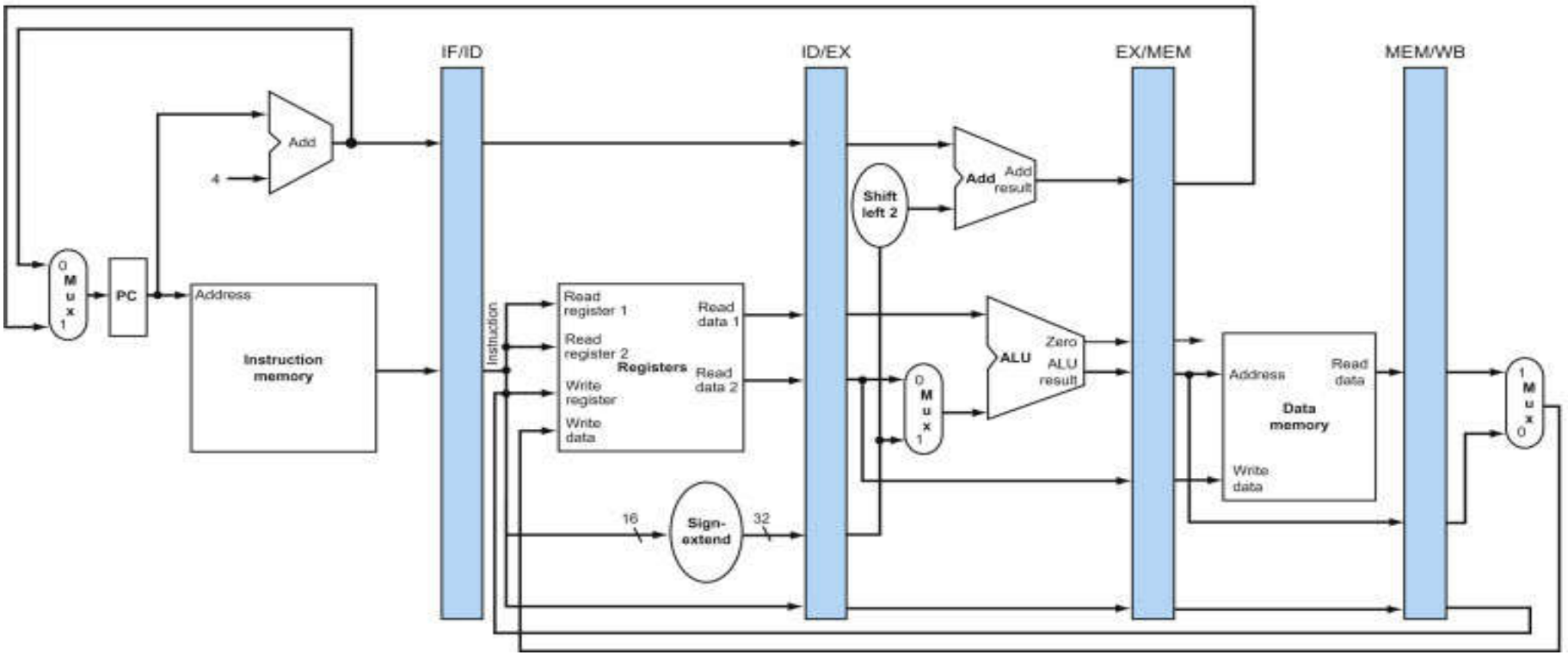
The Pipeline in Execution

sub $15, $4, $1 **lw $12, 1000($4)** **add $10, $1, $2** Memory Access Write Back



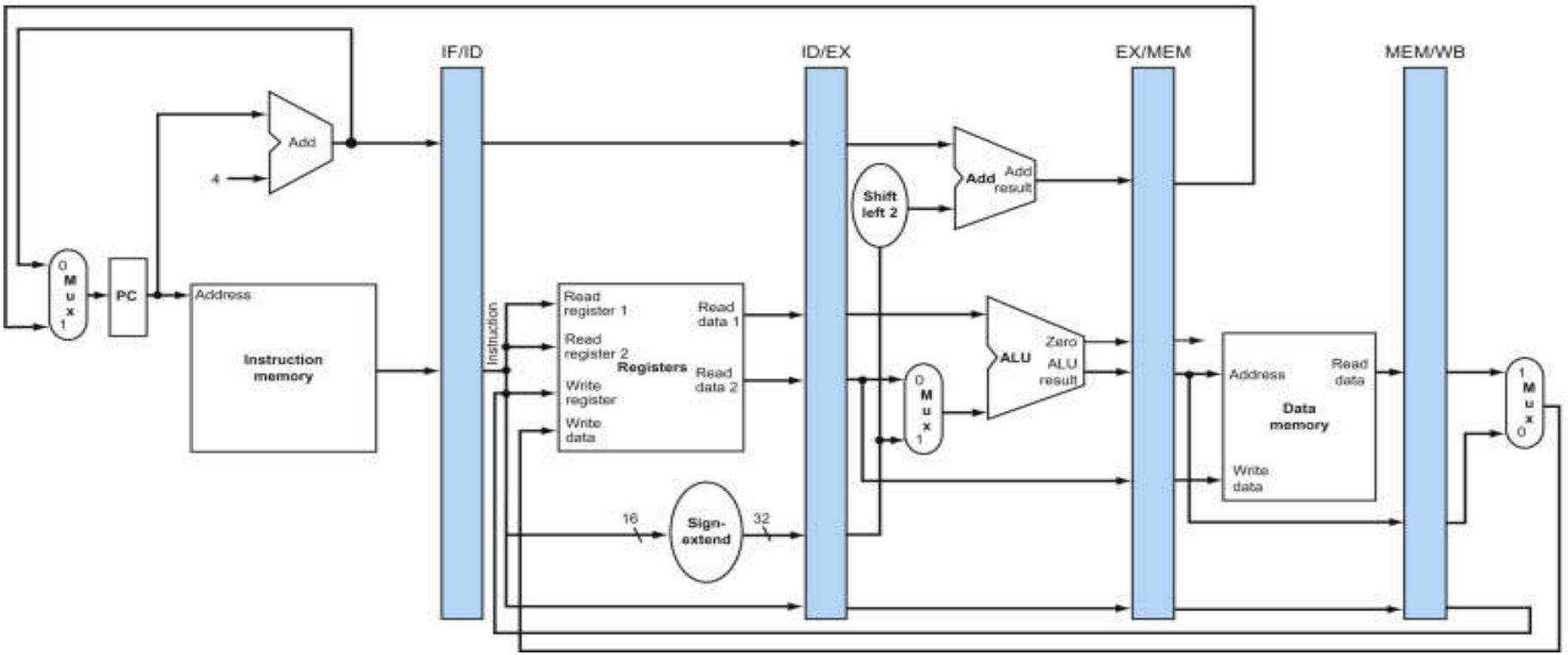
The Pipeline in Execution

Instruction Fetch | **sub $15, $4, $1** | **lw $12, 1000($4)** | **add $10, $1, $2** | Write Back



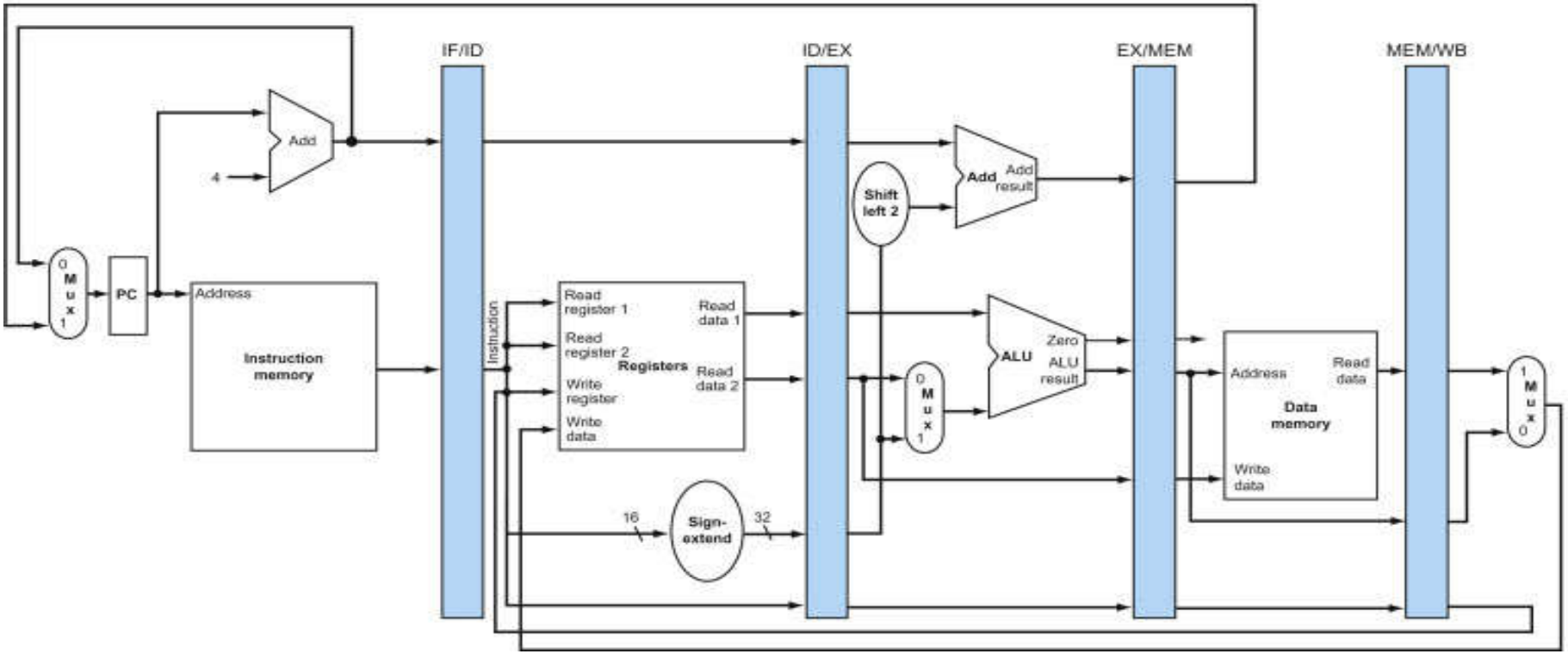
The Pipeline in Execution

Instruction Fetch | Instruction Decode/ Register Fetch | **sub $15, $4, $1** | **lw $12, 1000($4)** | **add $10, $1, $2**

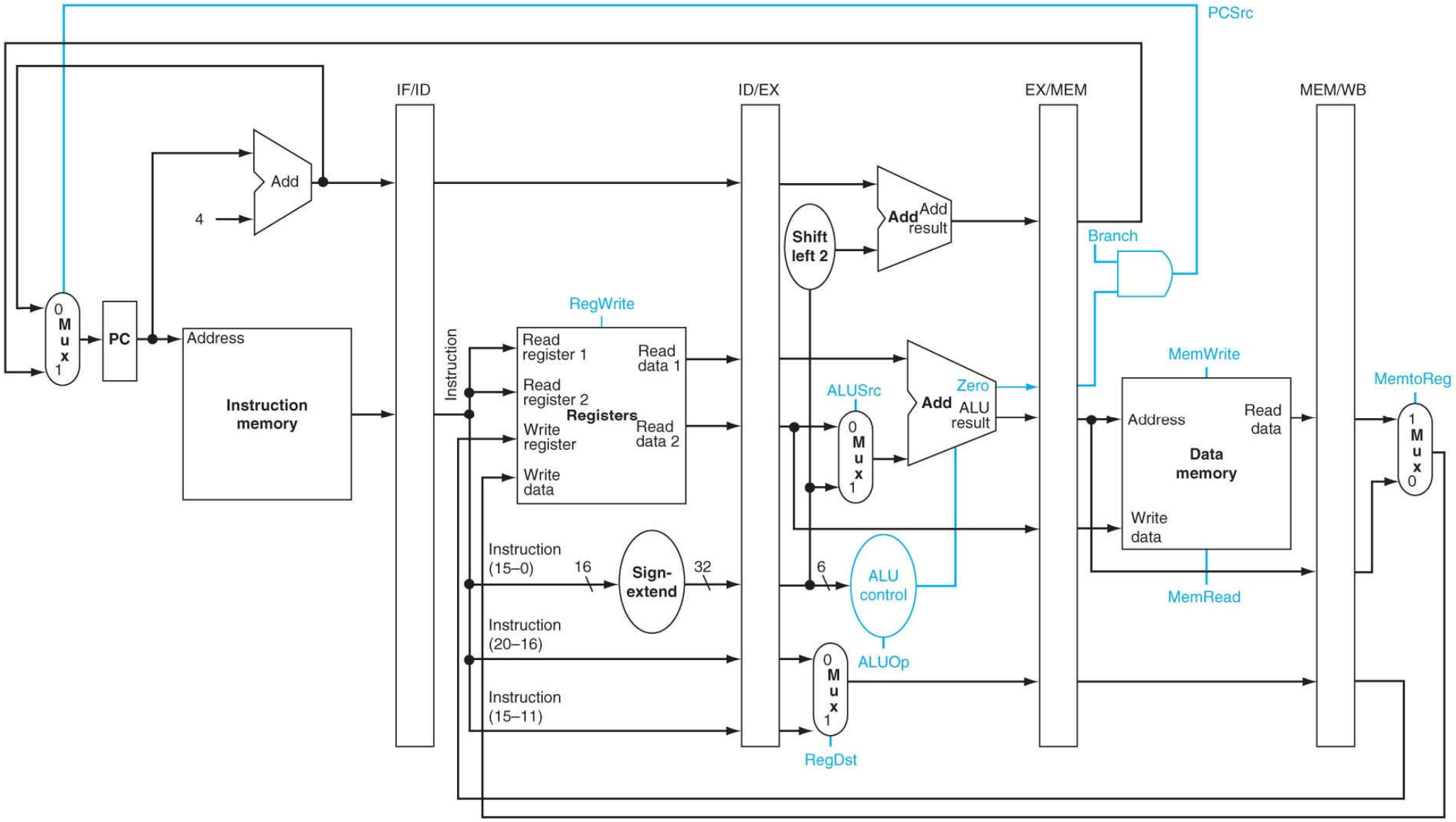


The Pipeline in Execution

Instruction Fetch	Instruction Decode/ Register Fetch	Execute/ Address Calculation	**sub $15, $4, $1**	**lw $12, 1000($4)**



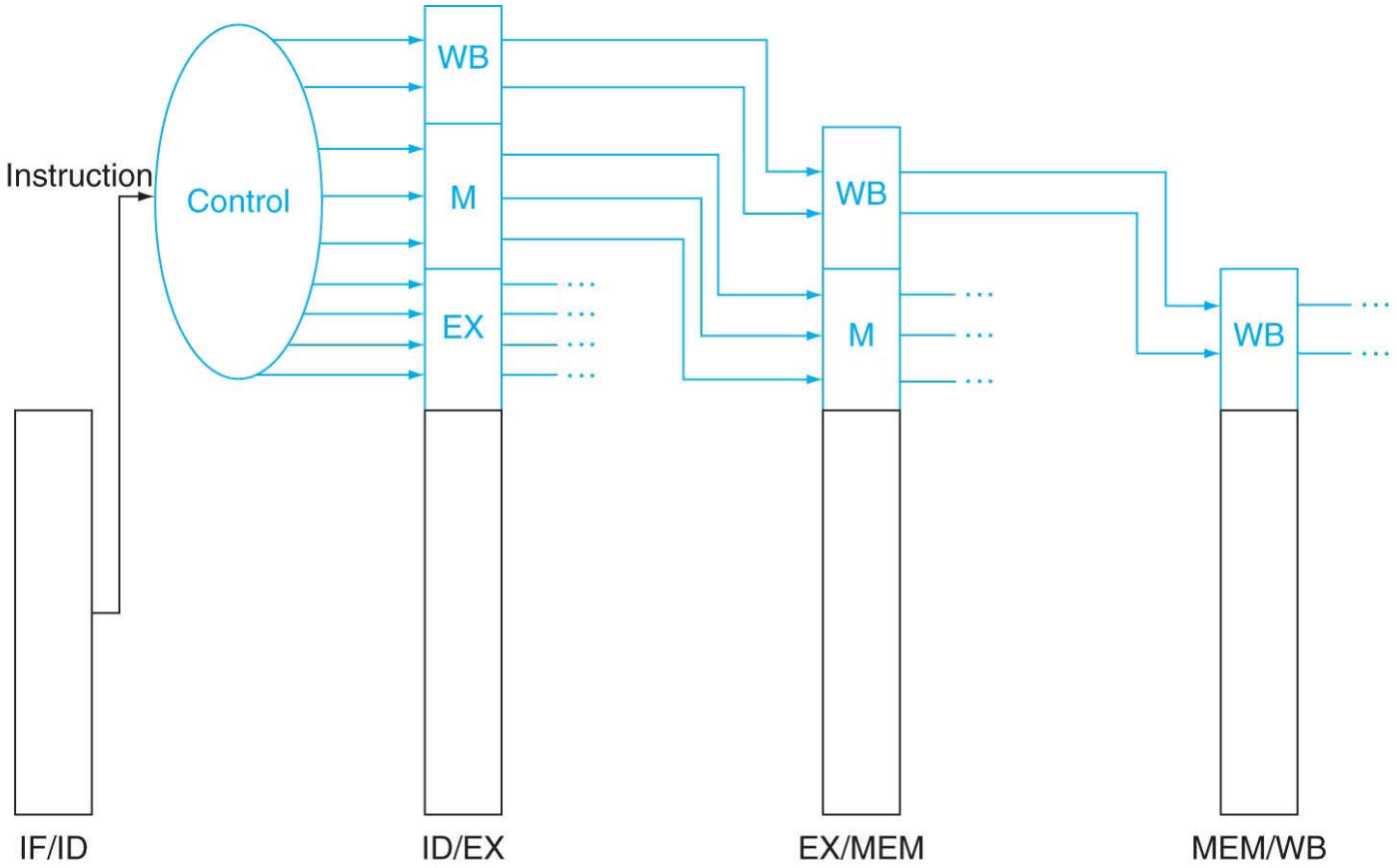
The Pipeline, with controls But….



Pipelined Control

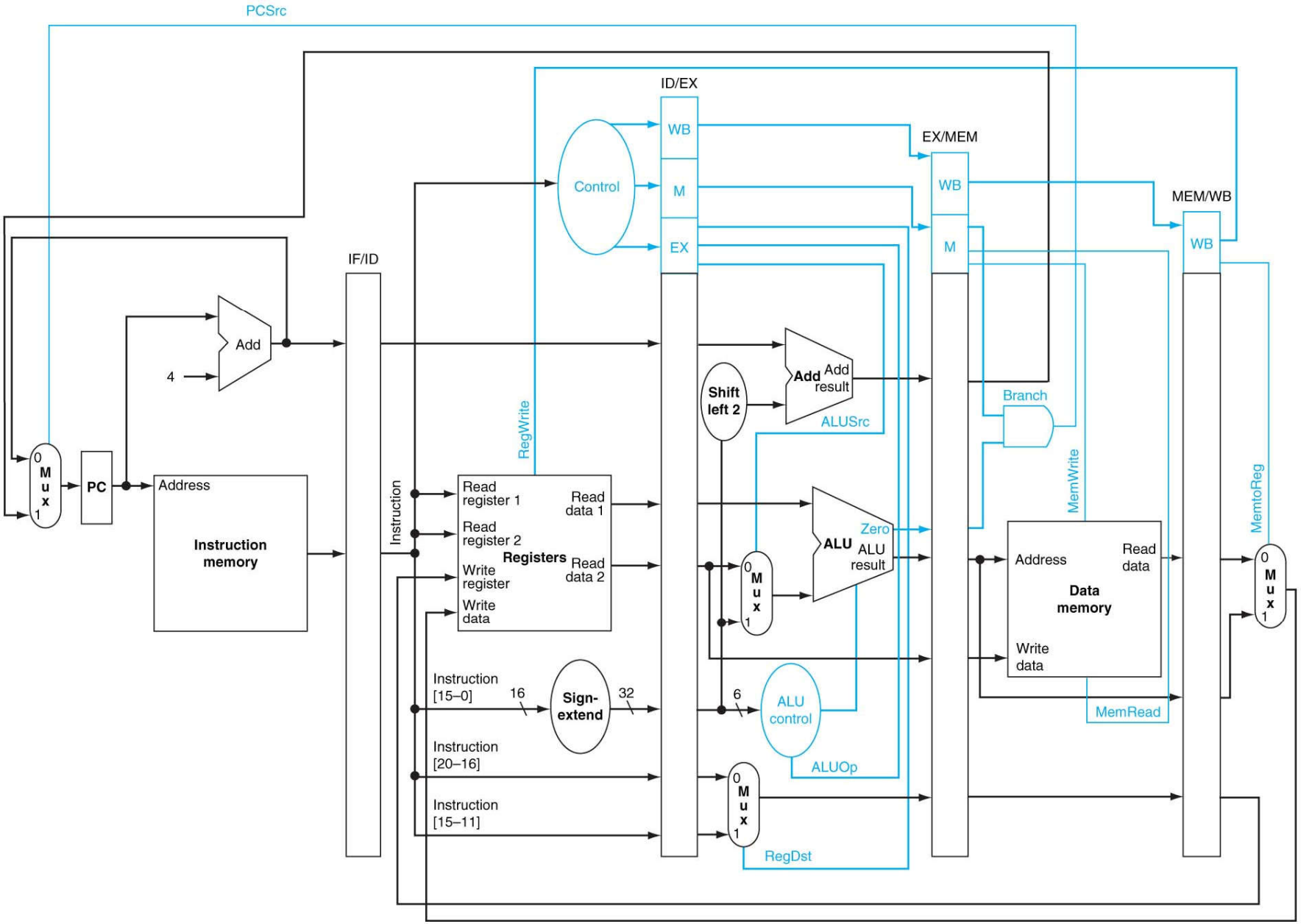
- I told you multicycle control was messy. We would expect pipelined control to be messier.
  - Why?
- But it turns out we can do it with just…
- Combinational logic!
  - signals generated once, but follow instruction through the pipeline

Pipelined Control



So, really it is combinational logic and some registers to propagate the signals to the right stage.

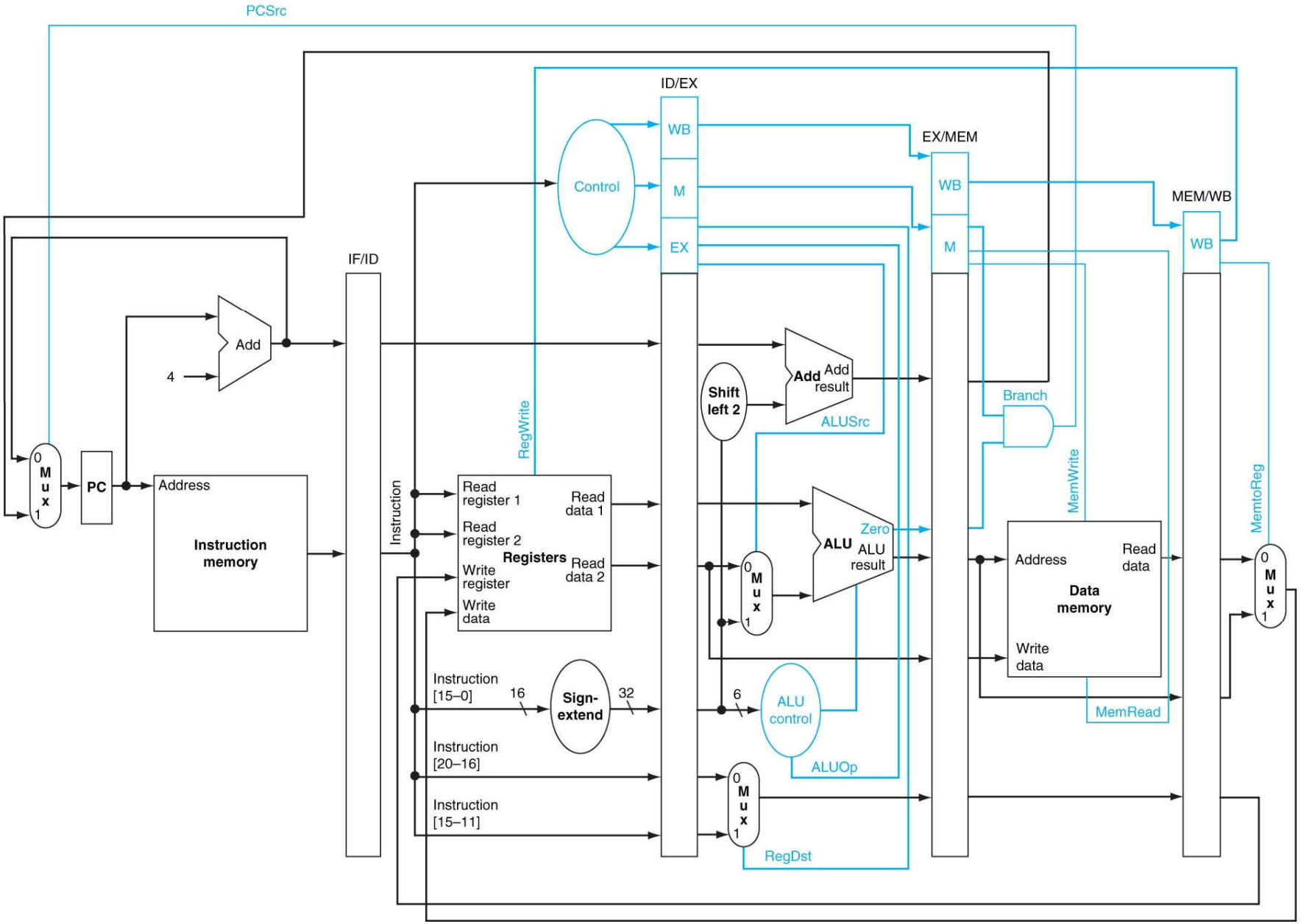
The Pipeline with Control Logic



Pipelined 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

The Pipeline with Control Logic



Is it really that easy?

- What happens when...

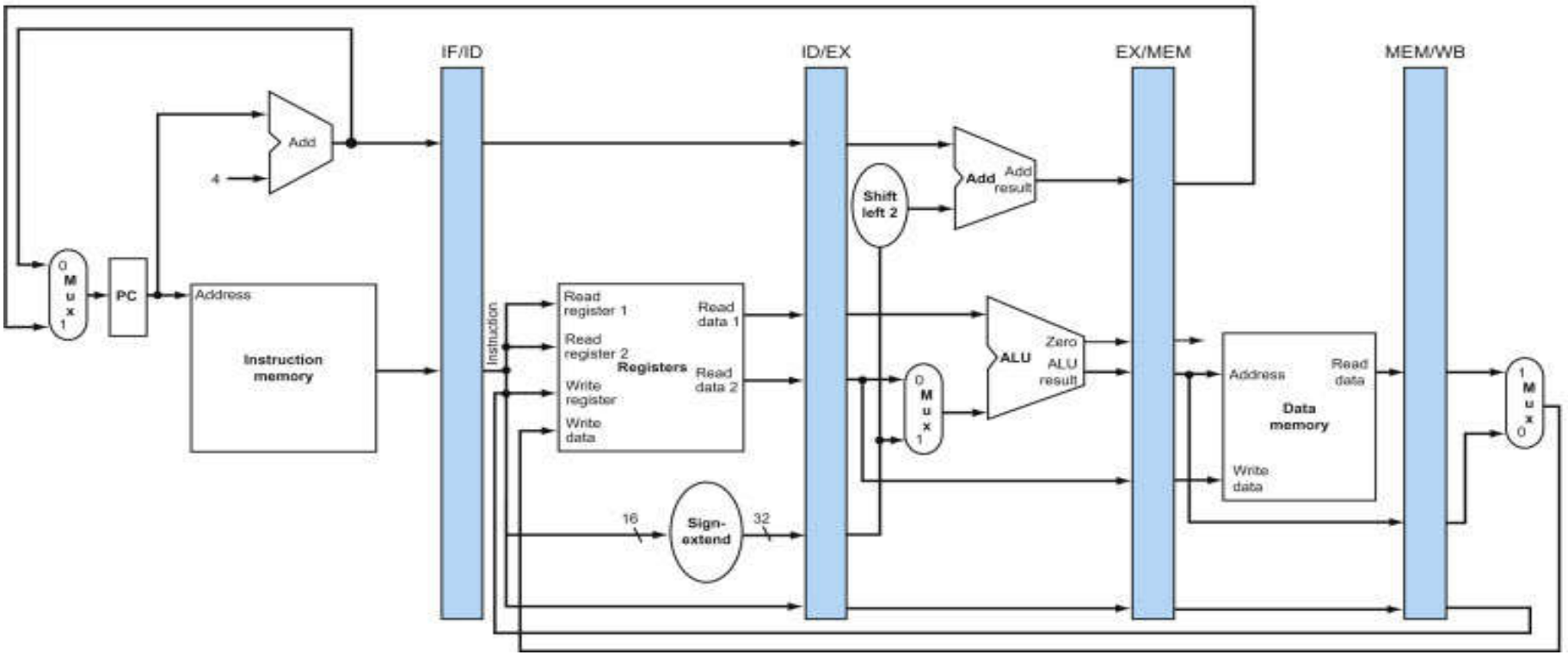
  add $3, $10, $11

  lw $8, 1000($3)

  sub $11, $8, $7

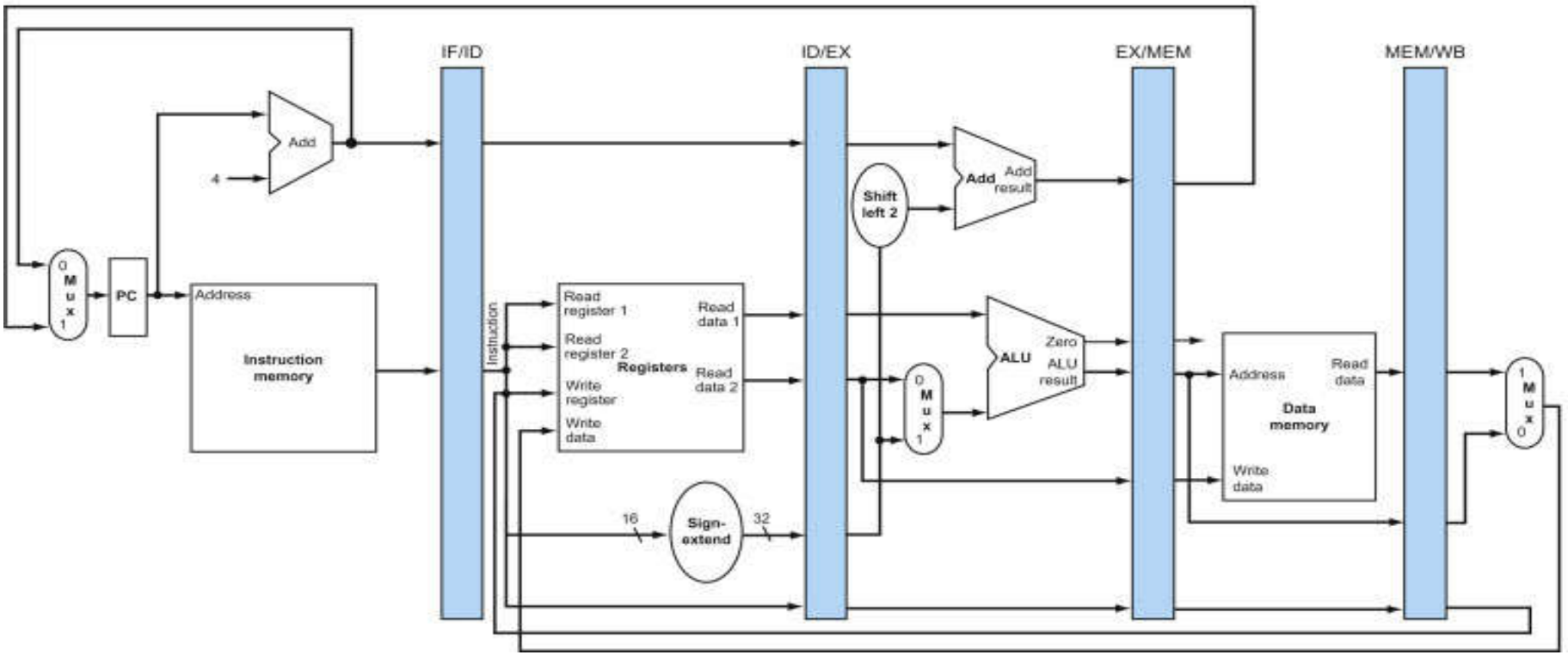
The Pipeline in Execution

lw $8, 1000($3) **add $3, $10, $11** Execute/ Address Calculation Memory Access Write Back



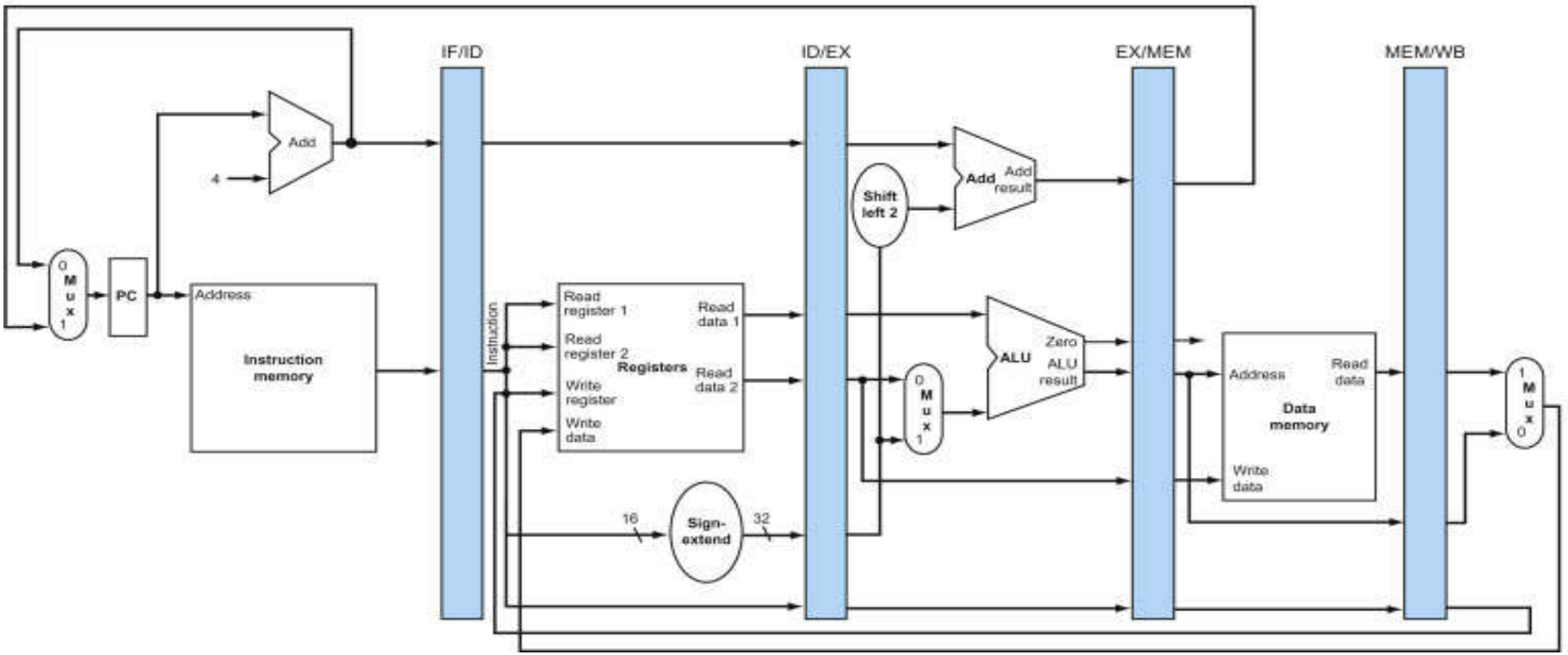
The Pipeline in Execution

sub $11, $8, $7 **lw $8, 1000($3)** **add $3, $10, $11** Memory Access Write Back



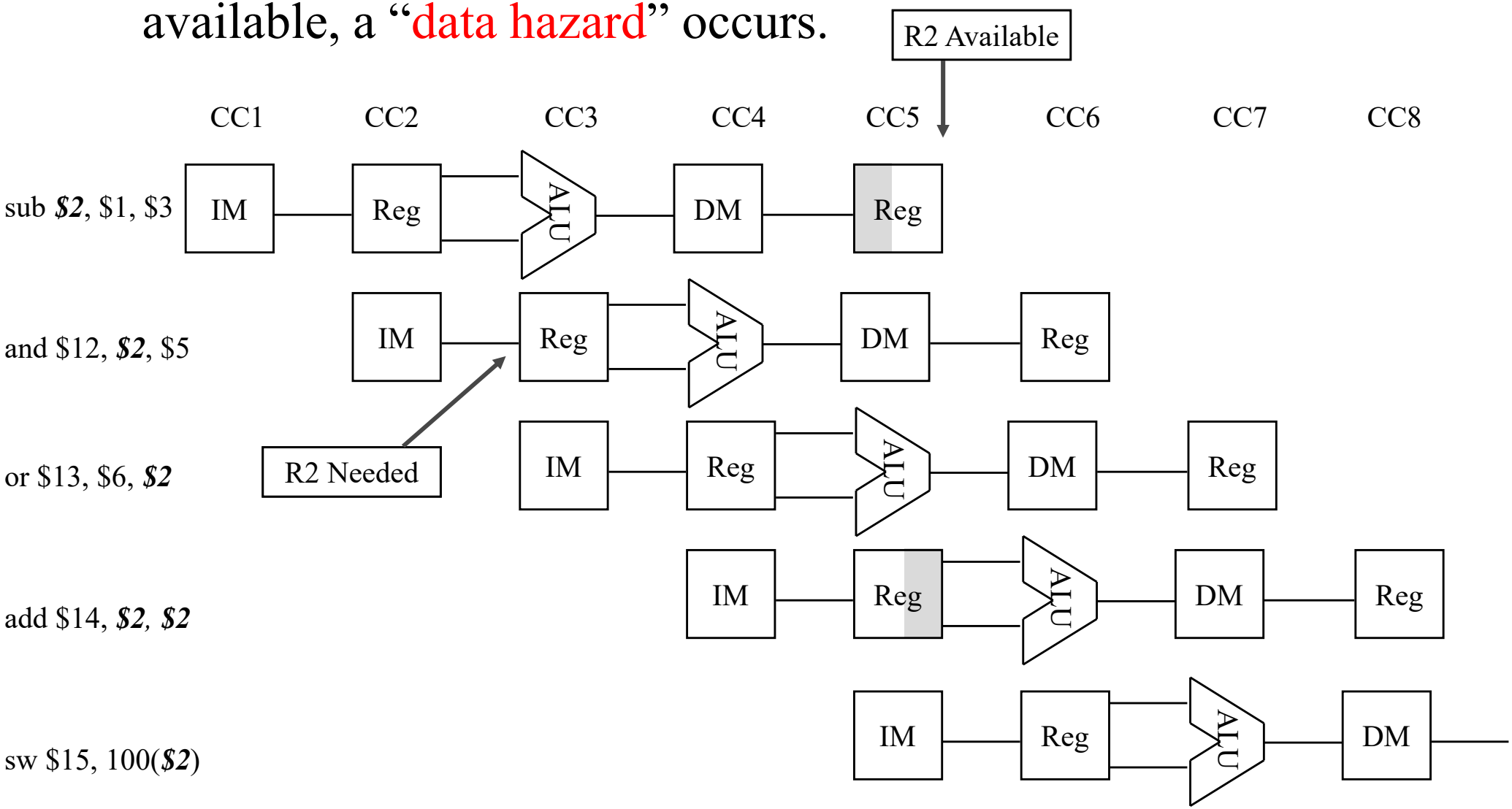
The Pipeline in Execution

add $10, $1, $2 **sub $11, $8, $7** **lw $8, 1000($3)** **add $3, $10, $11** Write Back



Data Hazards

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



So… what are we going to do about data hazards?

Pipelining Key Points

- ET = IC * CPI * CT
- We achieve high *throughput* without reducing instruction *latency*.
- Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles by different instructions).
- Pipelining uses combinational logic to generate (and registers to propogate) control signals.
- Pipelining creates potential hazards.