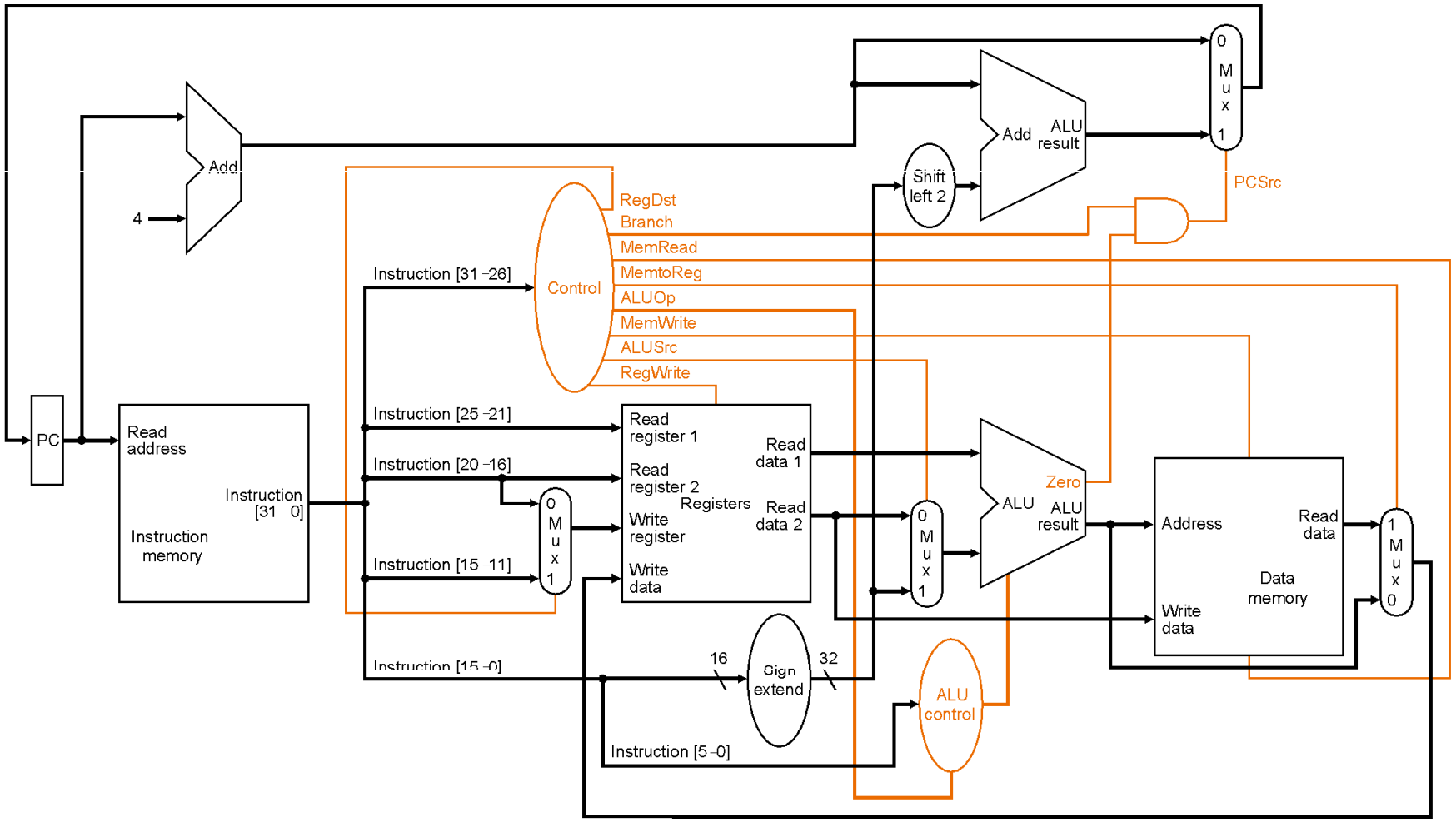
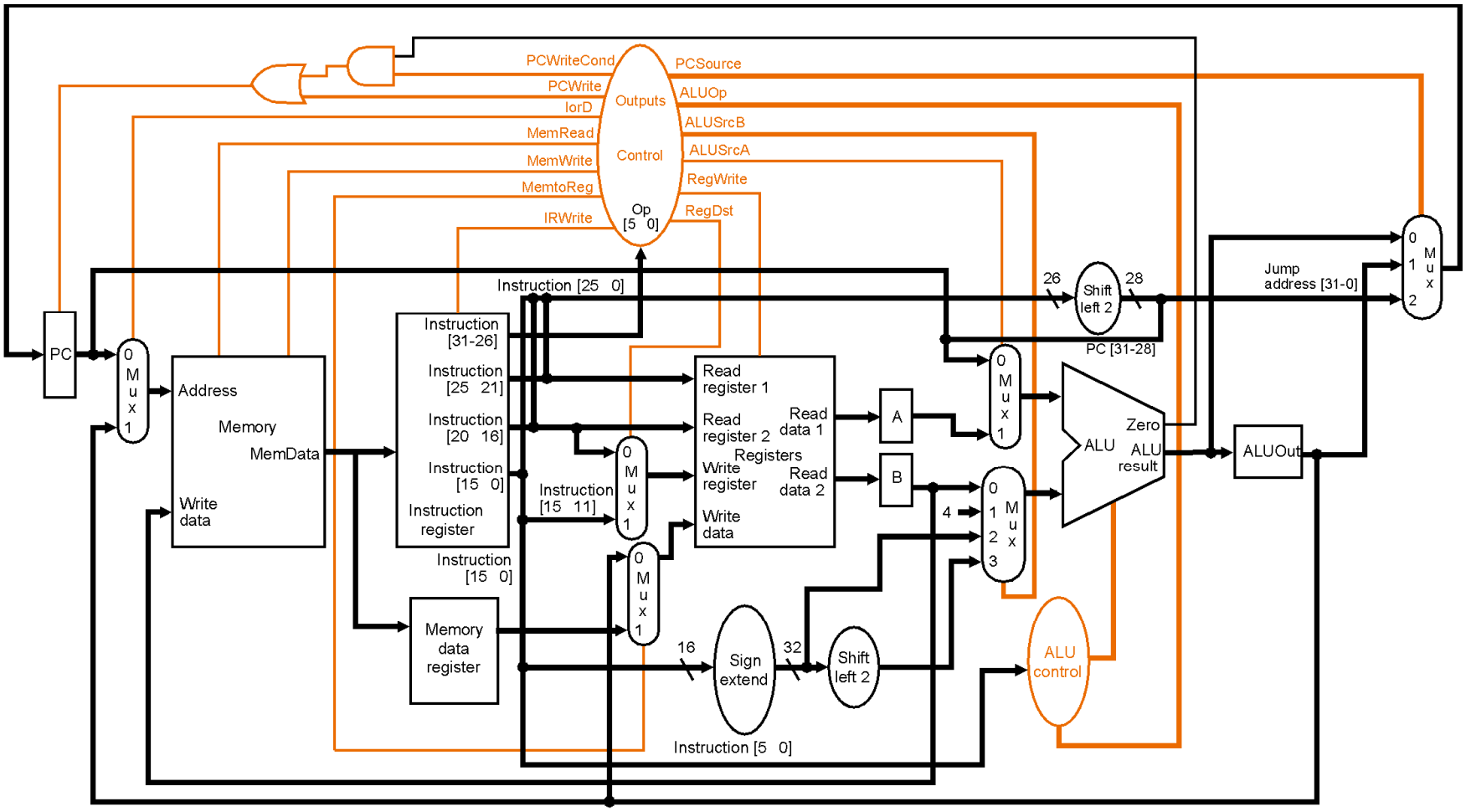
Designing a Pipelined CPU

Review -- Single Cycle CPU

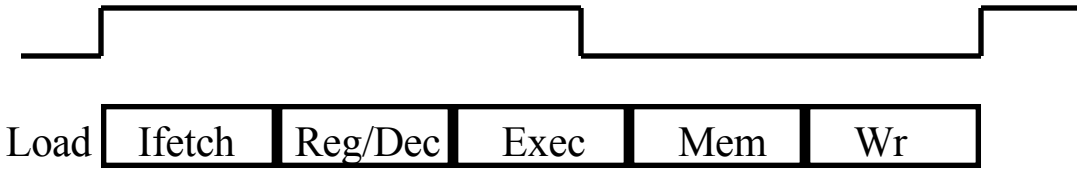


Review -- Multiple Cycle CPU

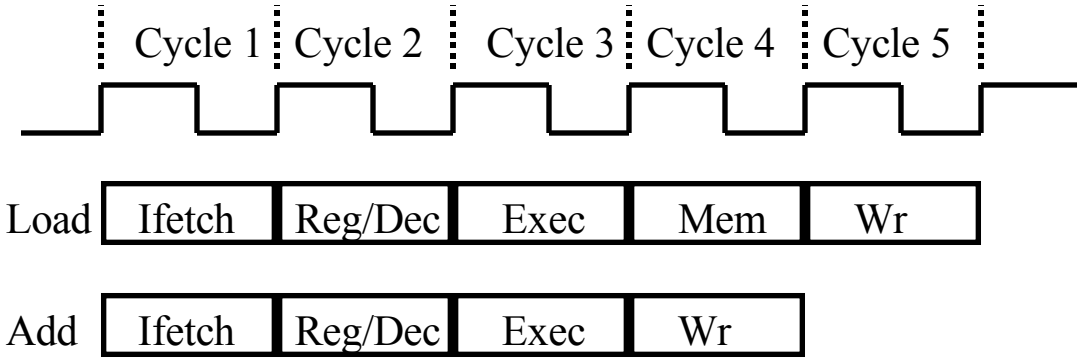


Review -- Instruction Latencies

- **Single-Cycle CPU**

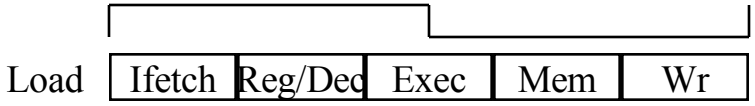


- **Multiple Cycle CPU**

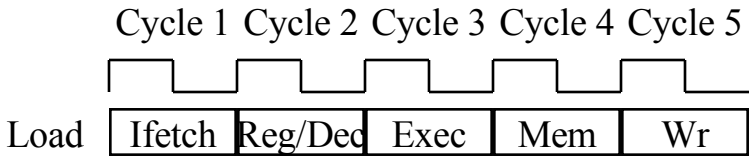


Instruction Latencies and Throughput

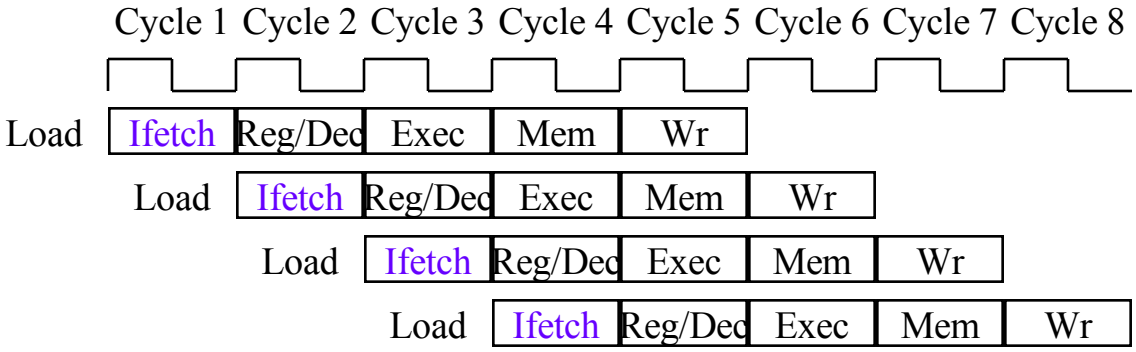
•Single-Cycle CPU



•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	FSM or Microprogram	$\frac{1}{\quad}$
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

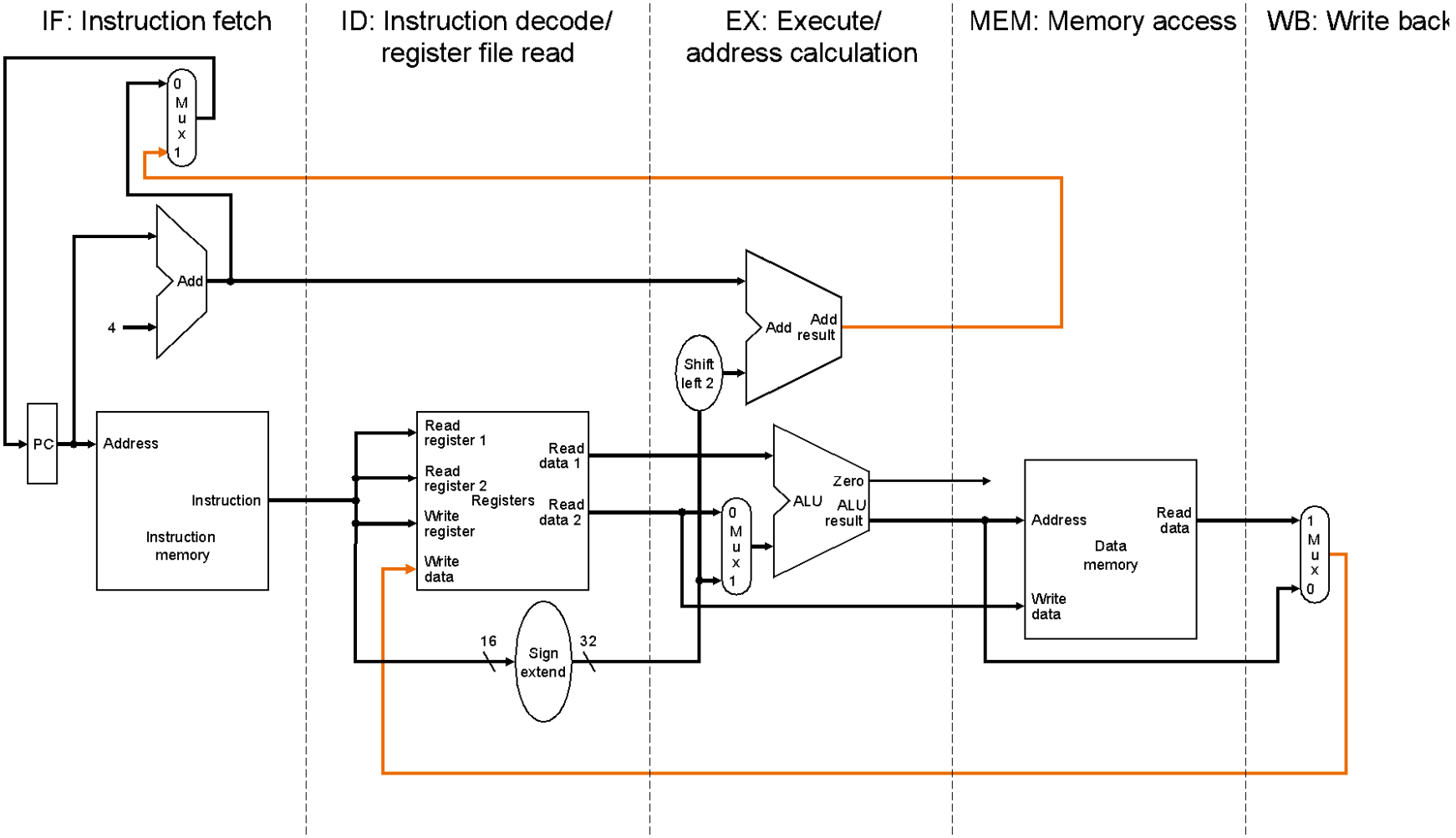
ID: Instruction decode and register fetch

EX: Execution and effective address calculation

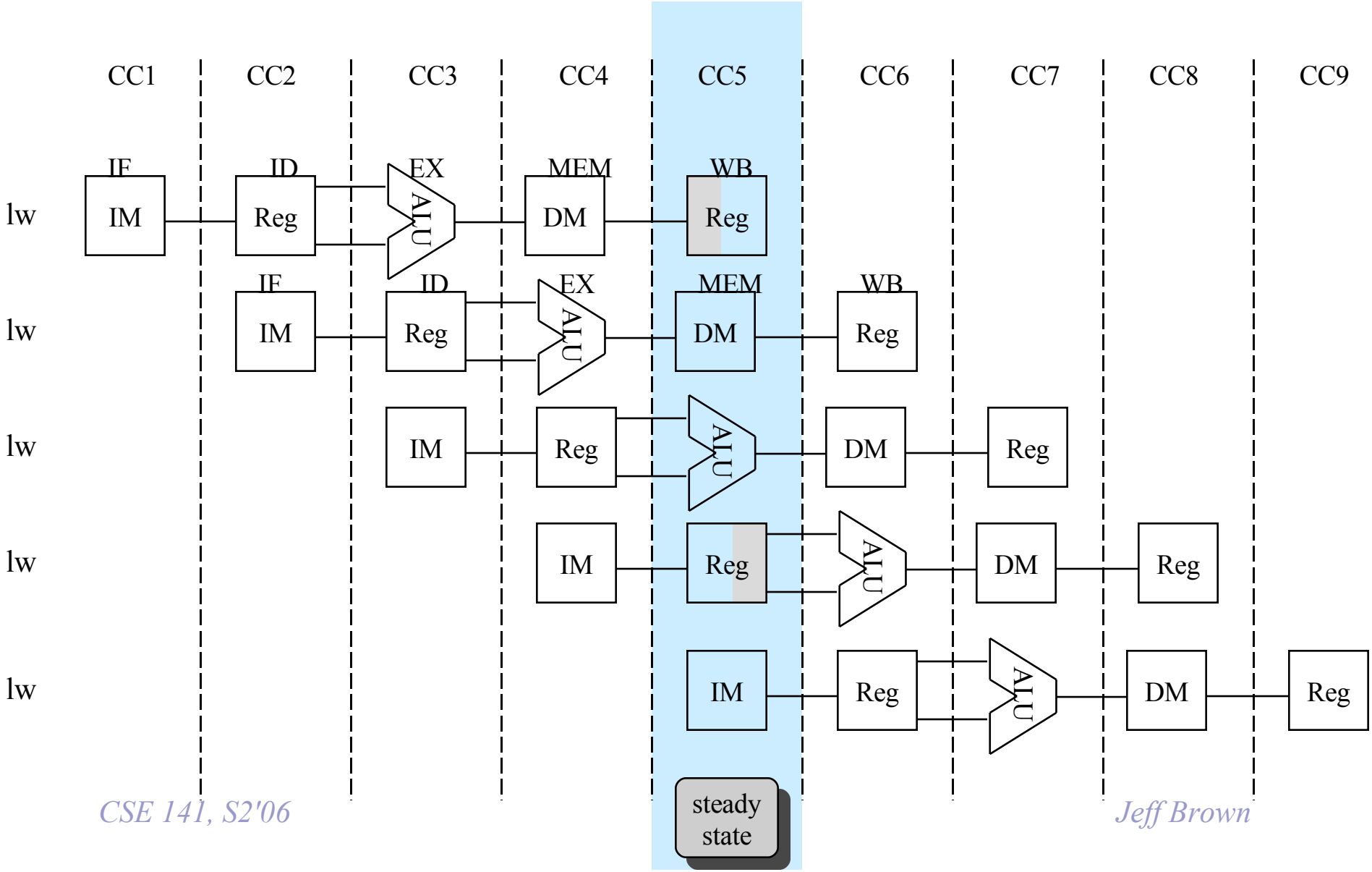
MEM: Memory access

WB: Write back

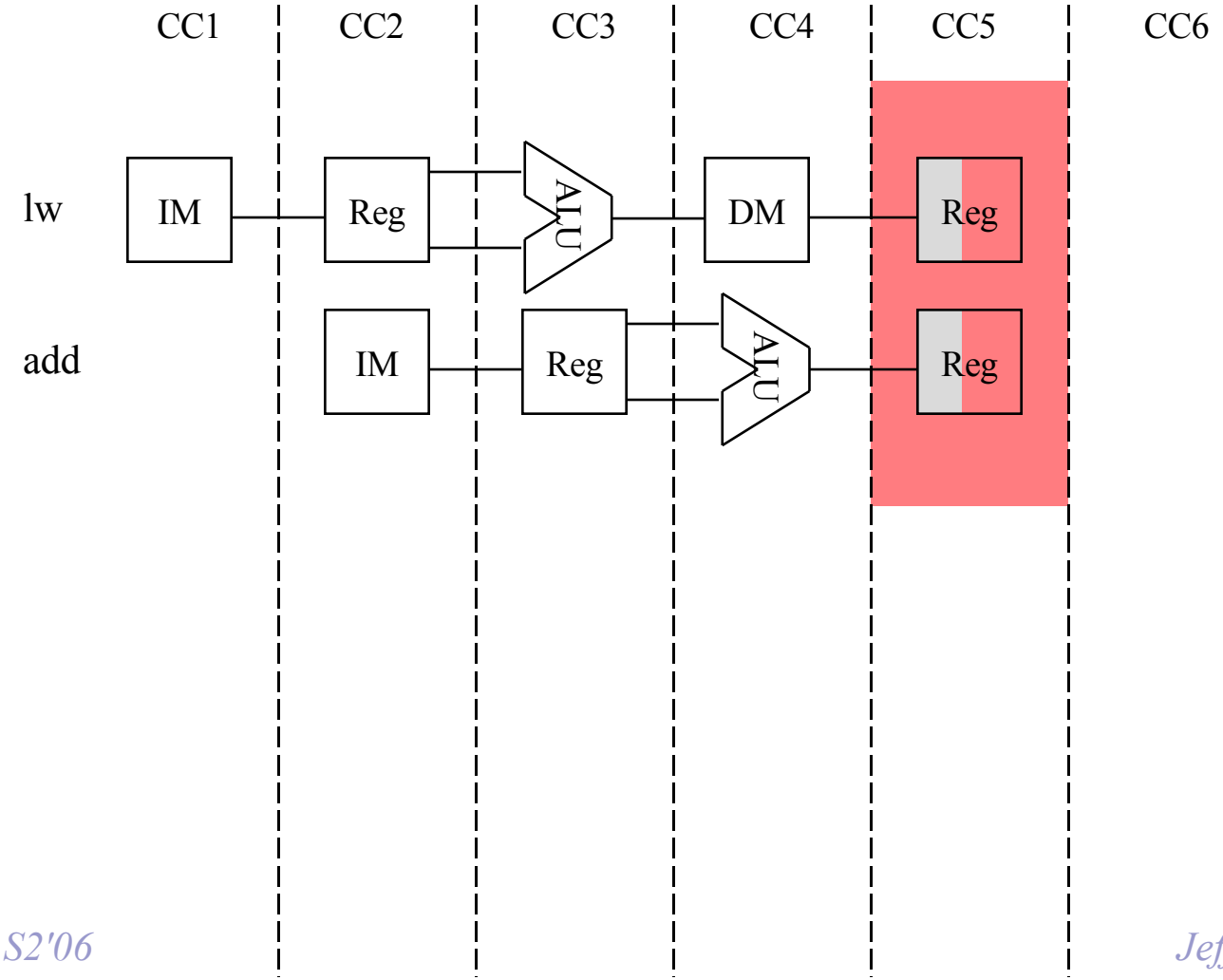
Pipelined Datapath



Execution in a Pipelined Datapath

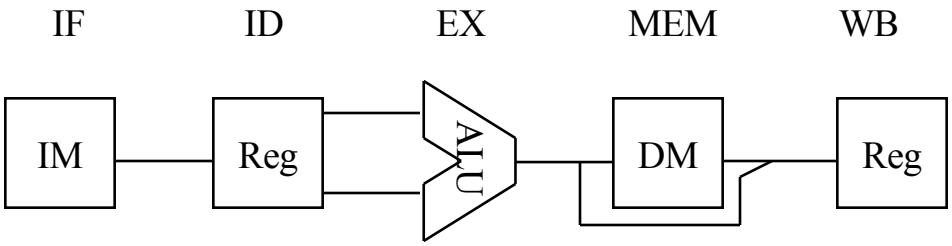


Mixed Instructions in the Pipeline

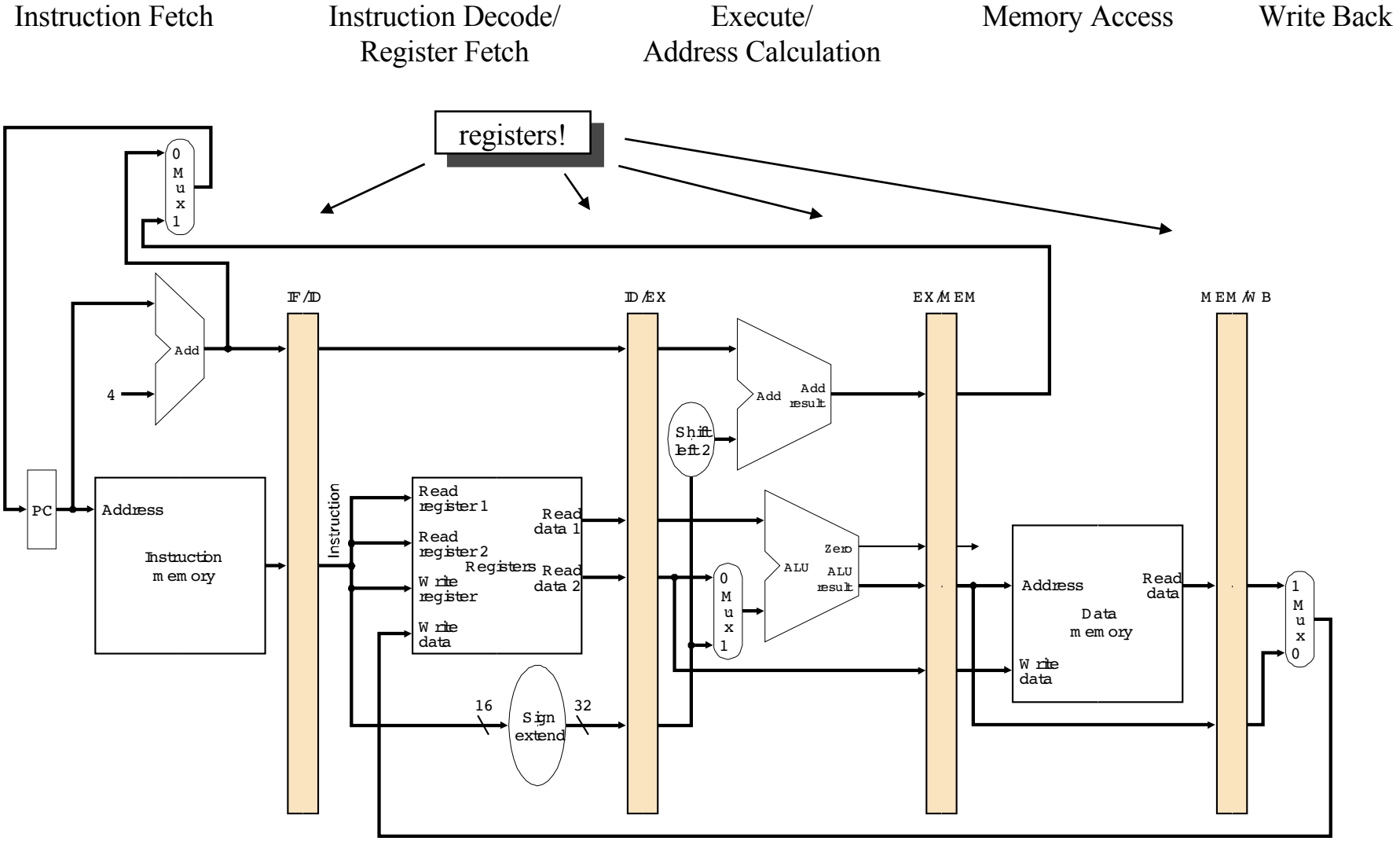


Pipeline Principles

- All instructions that share a pipeline must 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

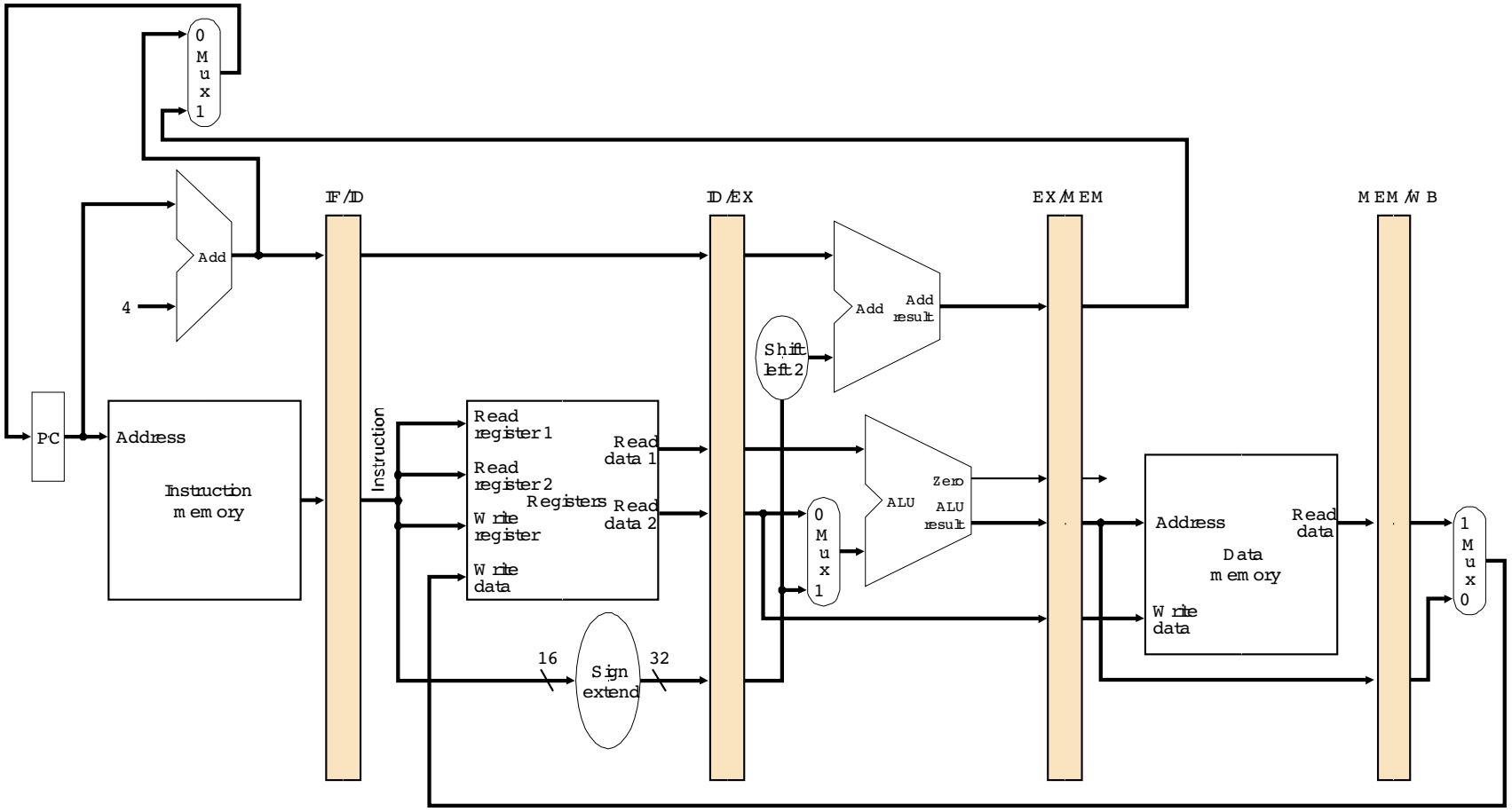


Pipelined Datapath



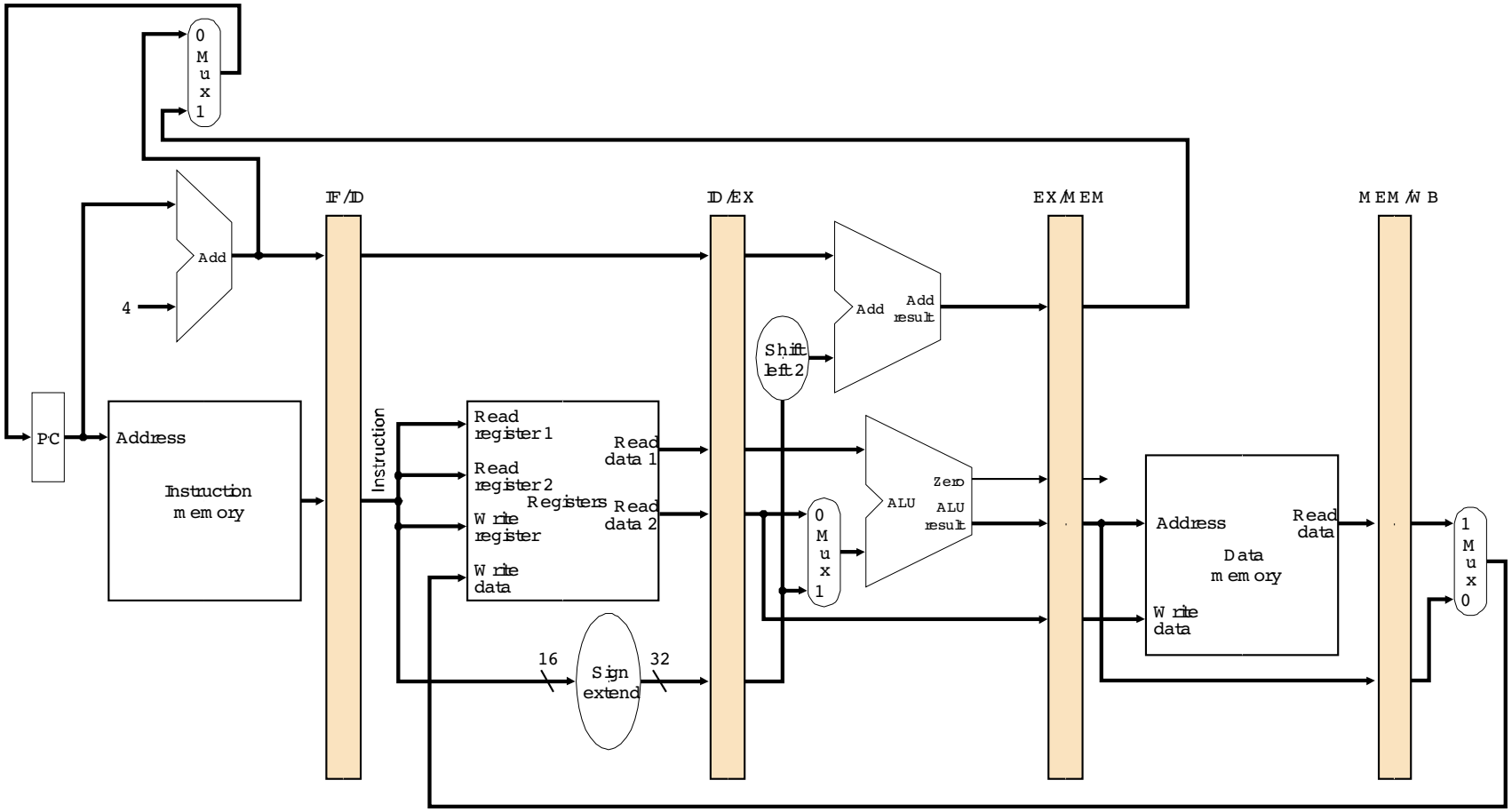
The Pipeline in Execution

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



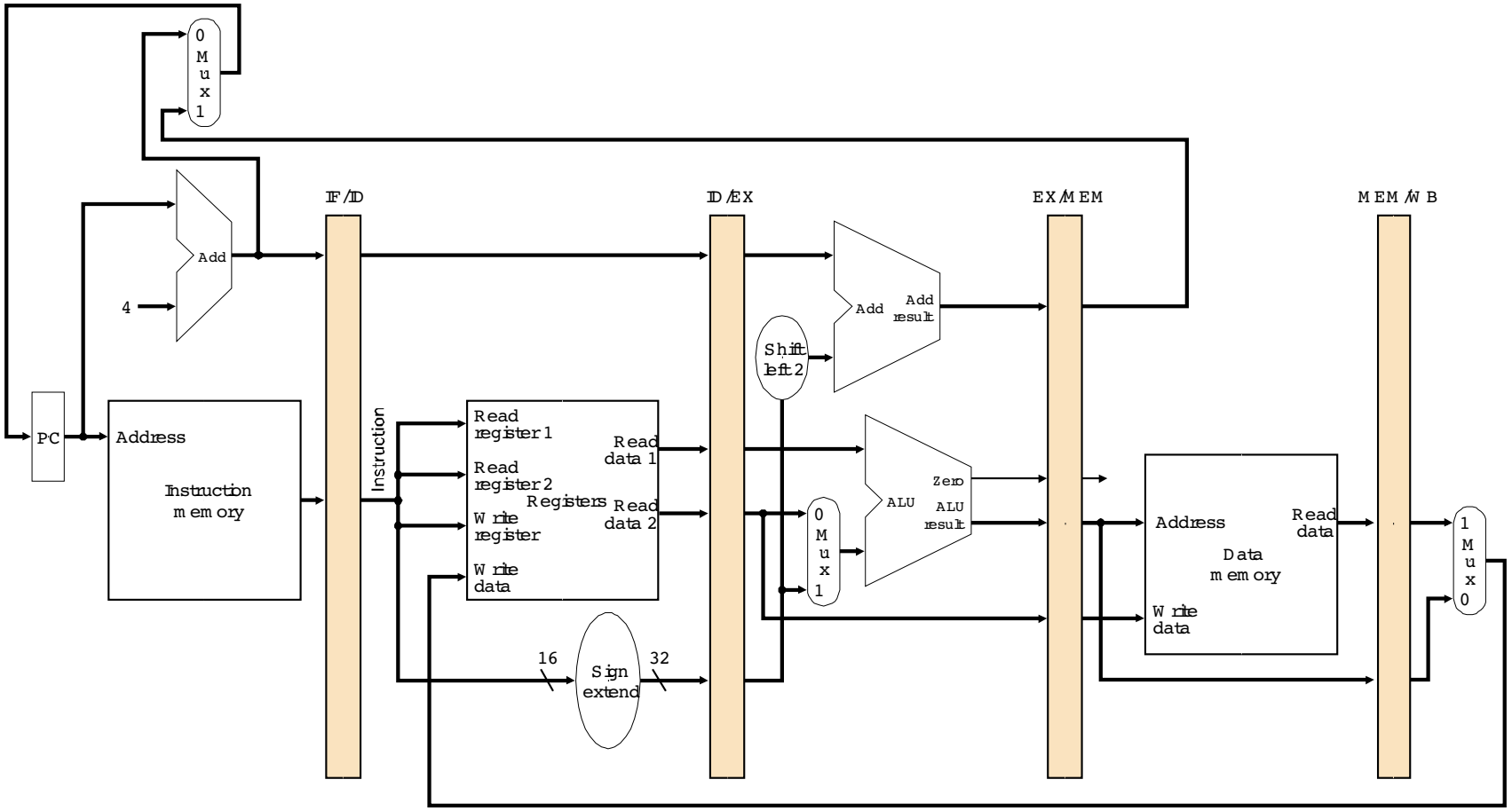
The Pipeline in Execution

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



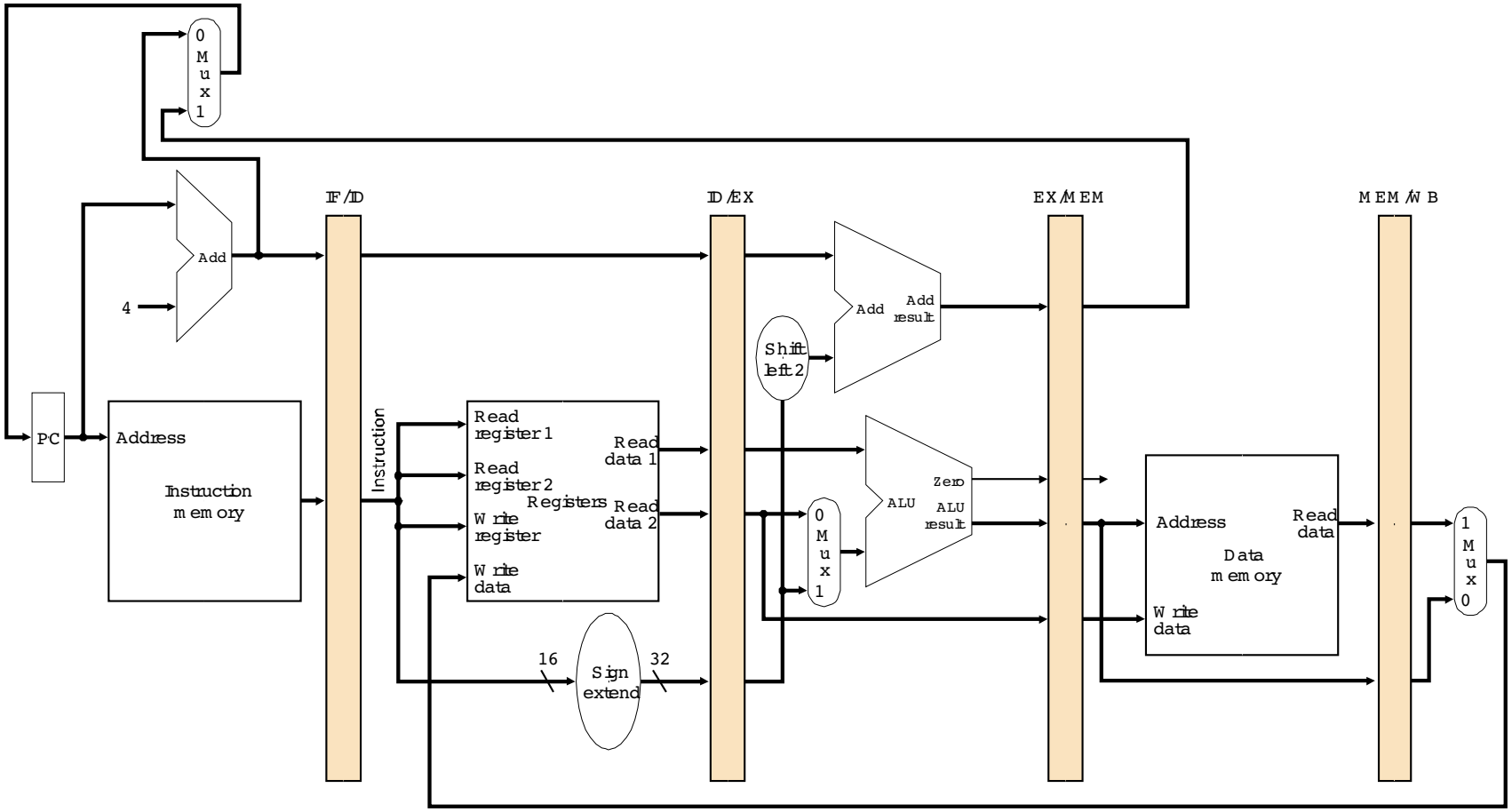
The Pipeline in Execution

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



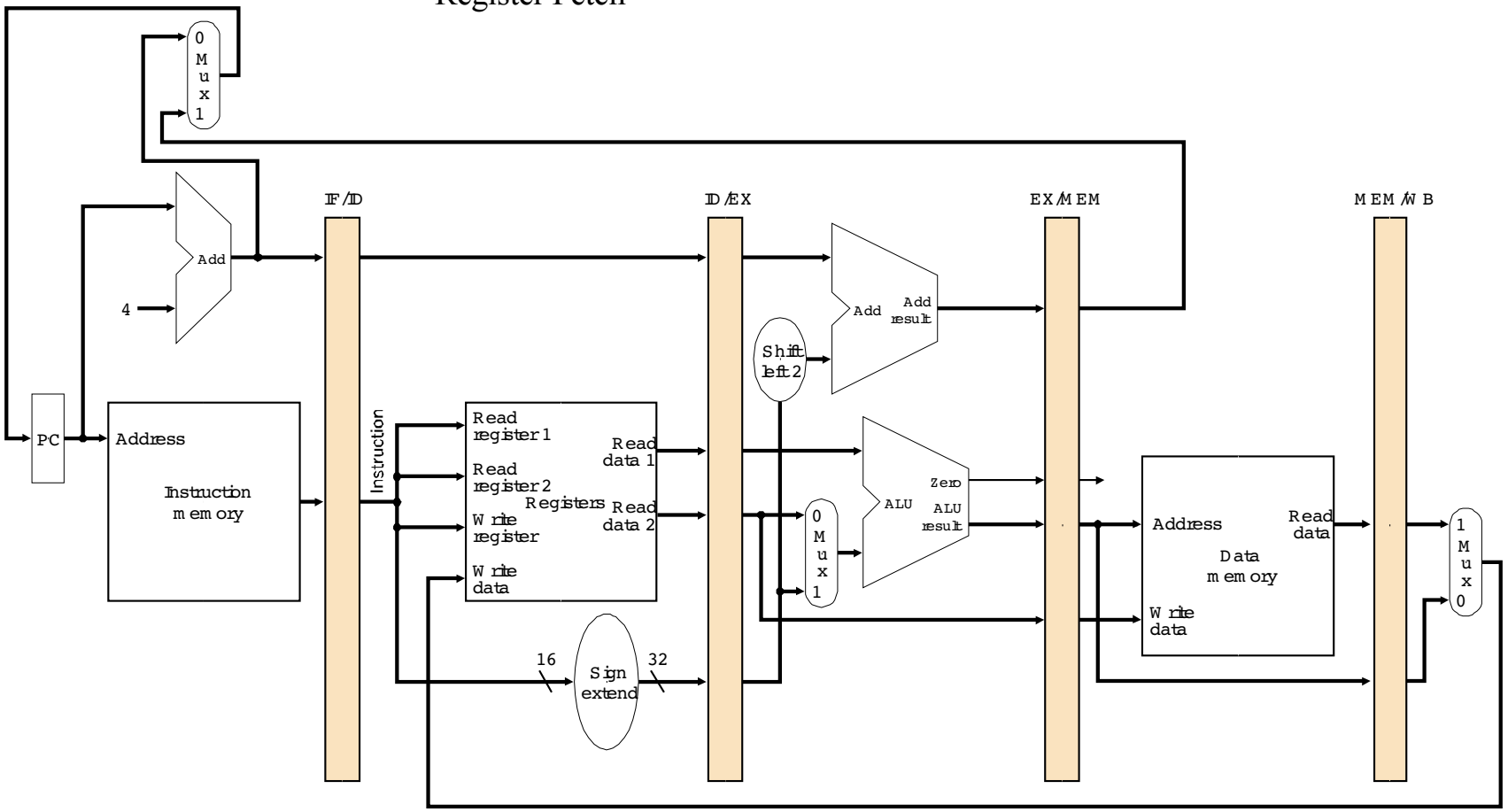
The Pipeline in Execution

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



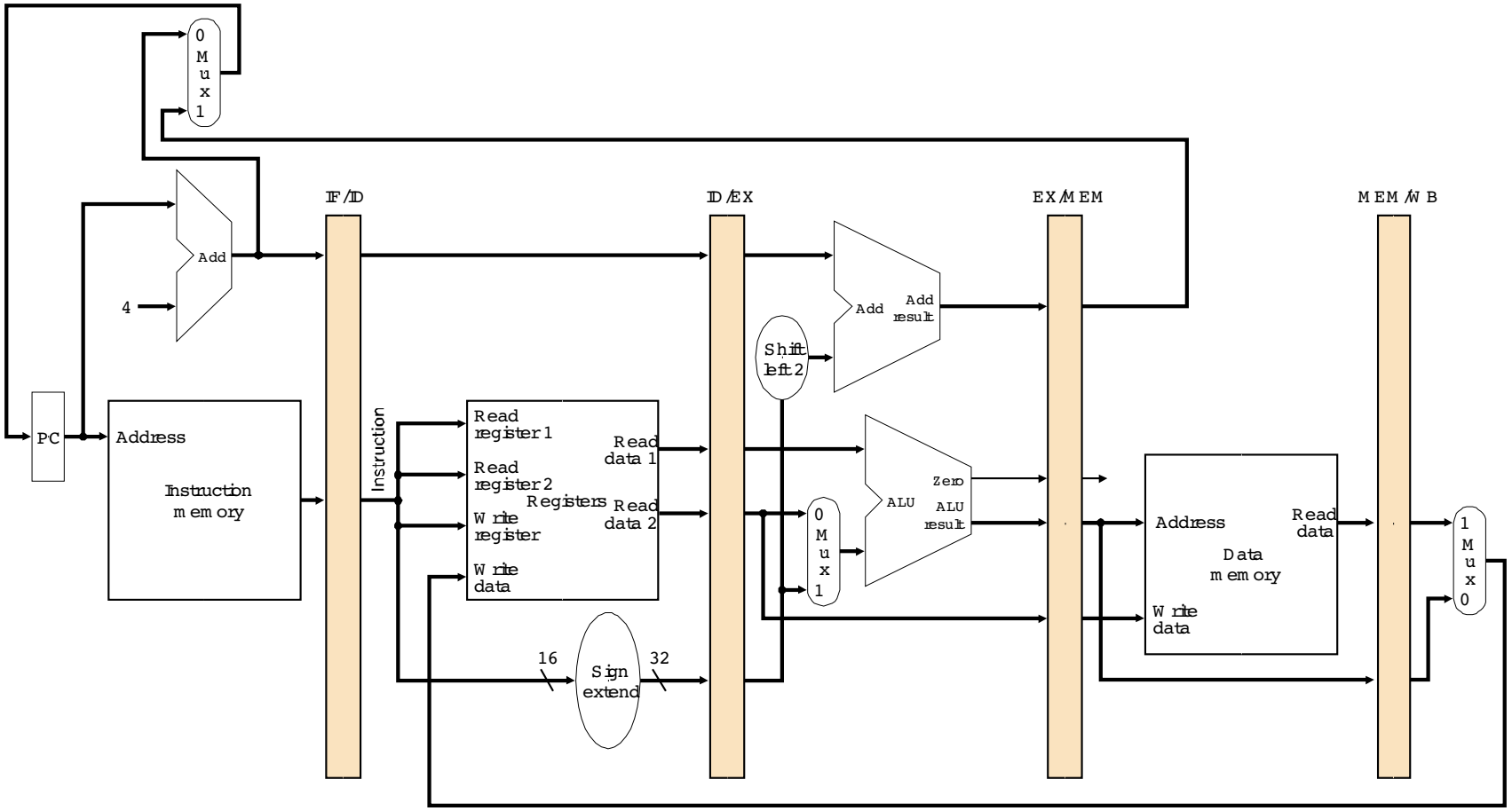
The Pipeline in Execution

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

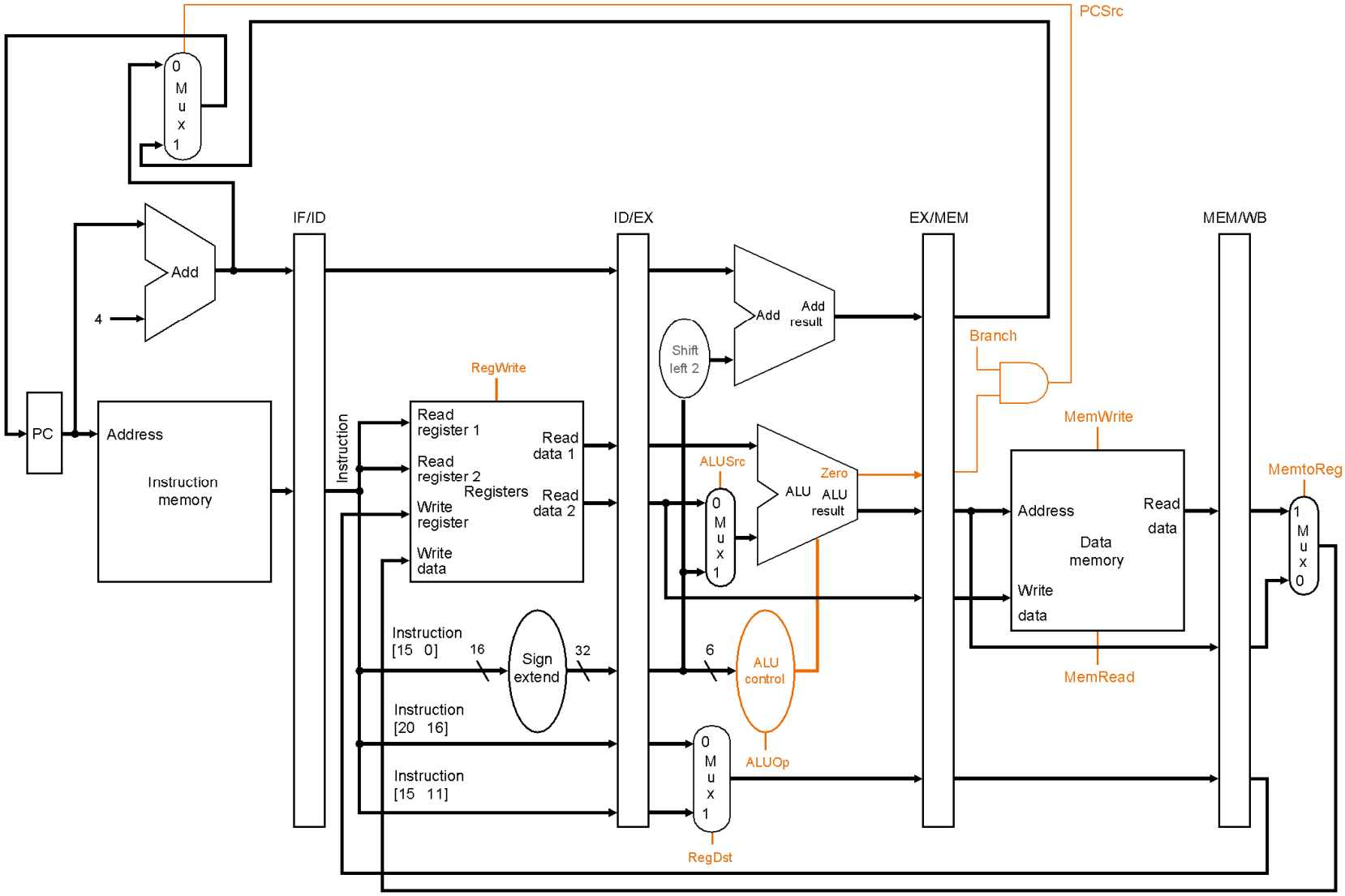


The Pipeline in Execution

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



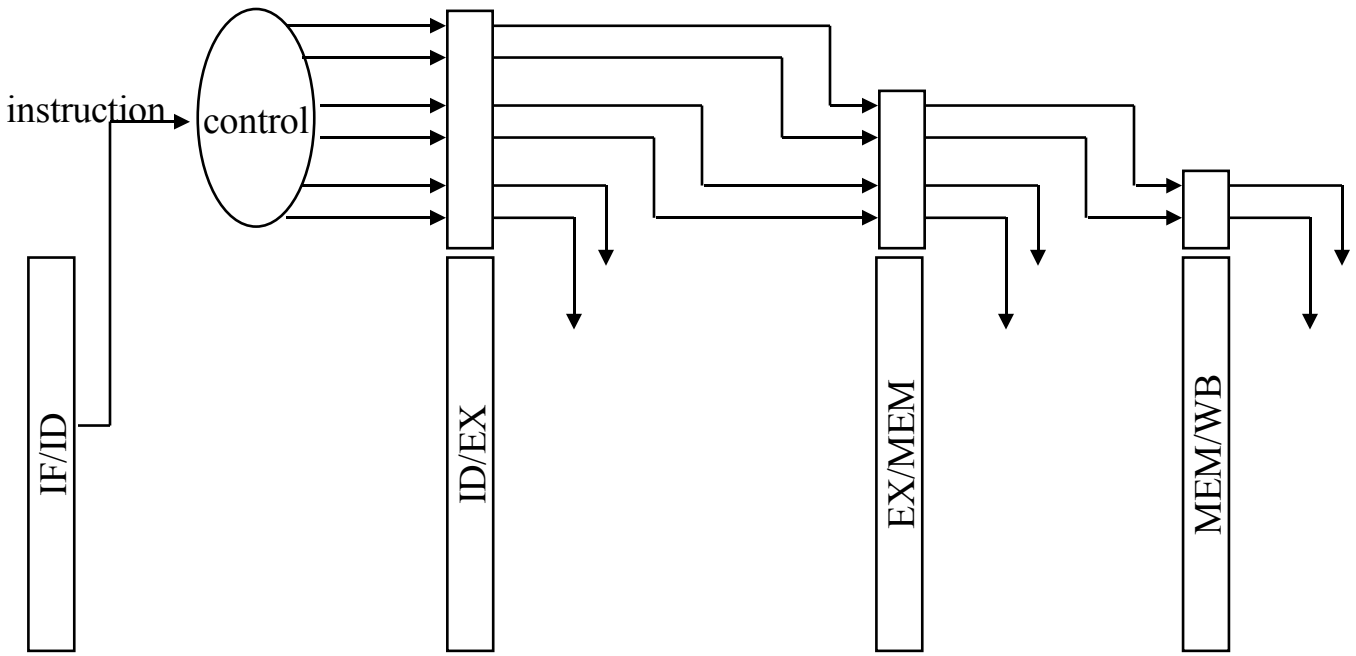
The Pipeline, with controls But….



Pipelined Control

- can't use microprogram.
- FSM not really appropriate.
- Combinational logic!
  - signals generated once, but follow instruction through the pipeline

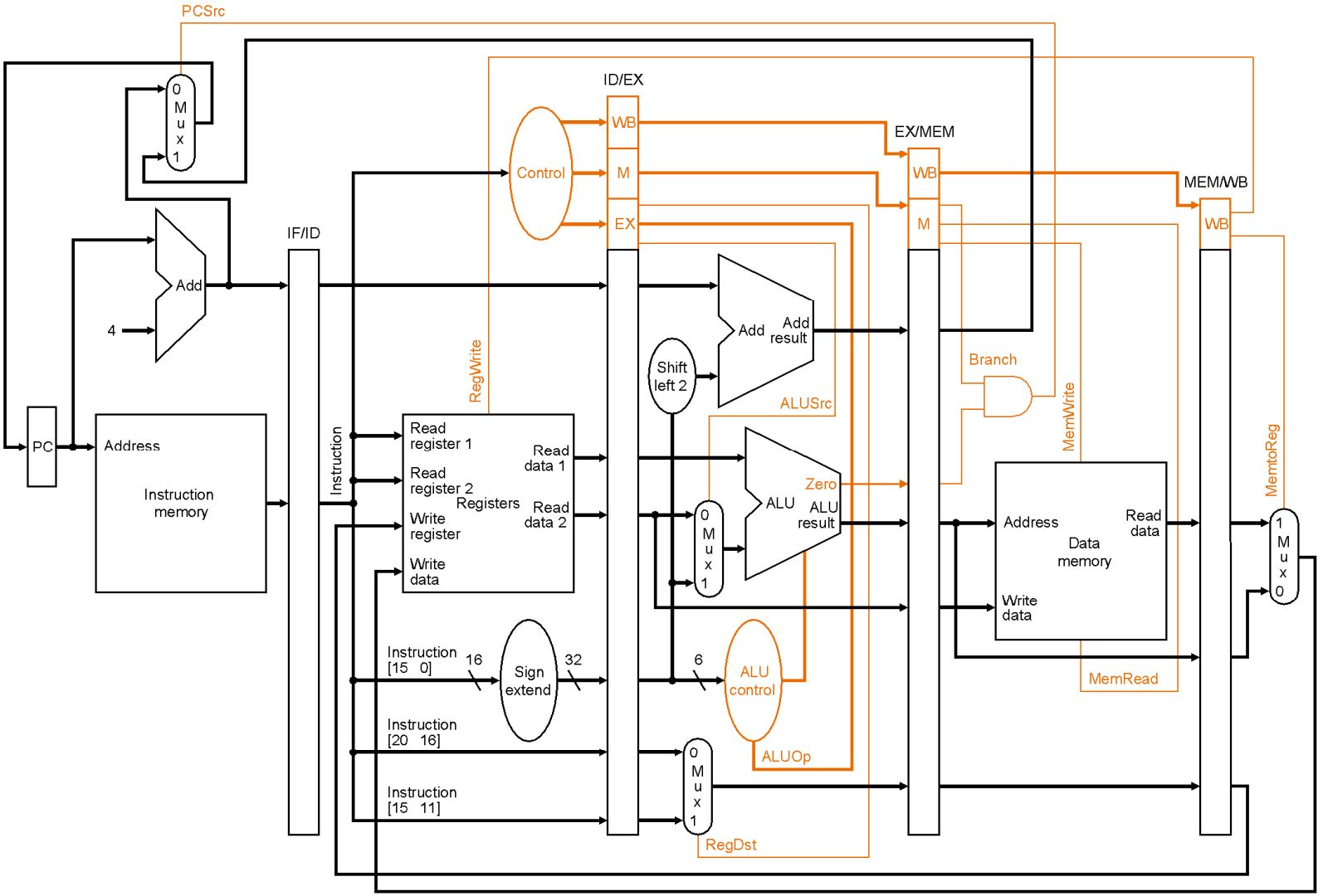
Pipelined Control



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...

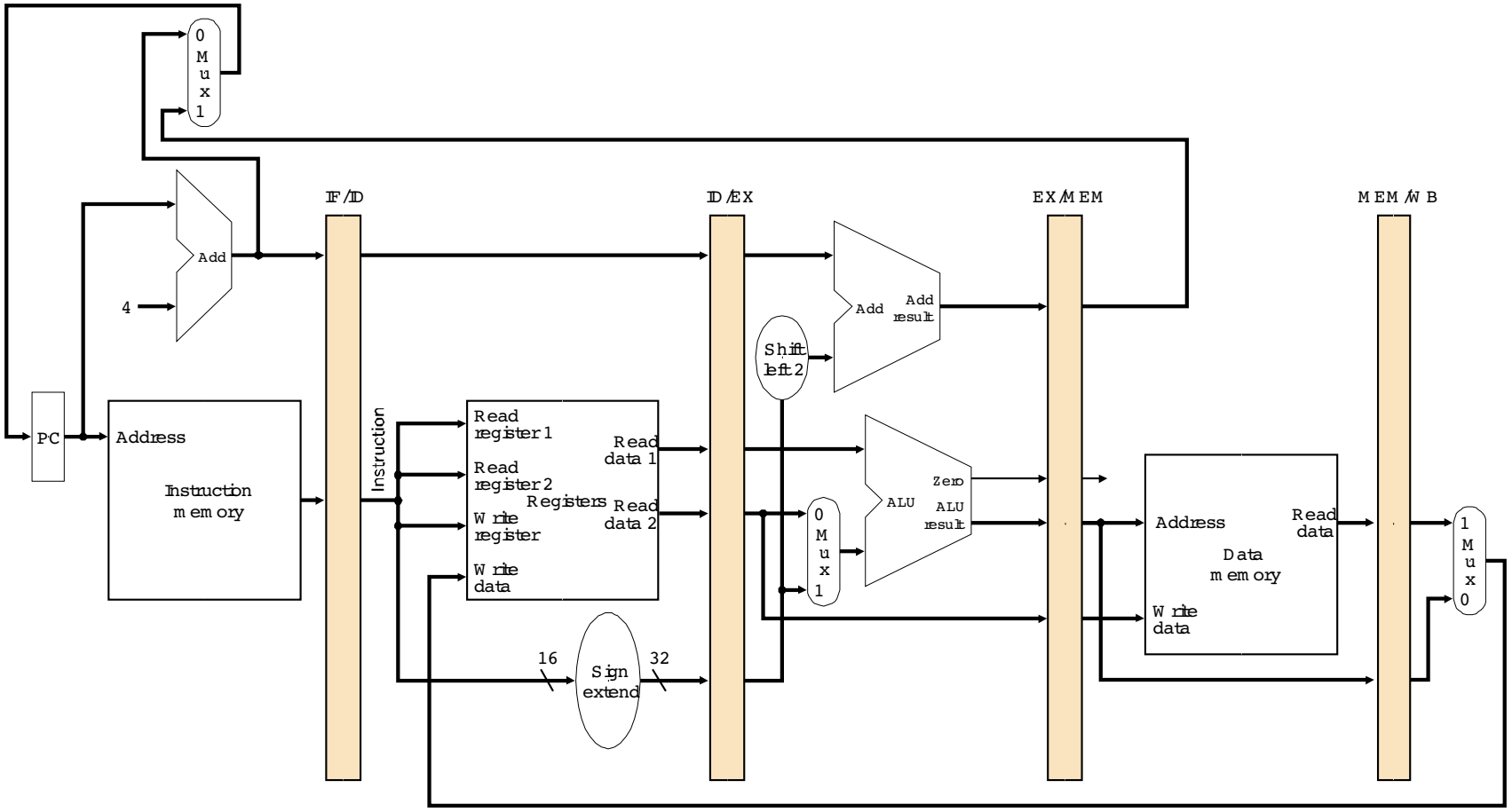
  add $3, $10, $11

  lw $8, 1000($3)

  sub $11, $8, $7

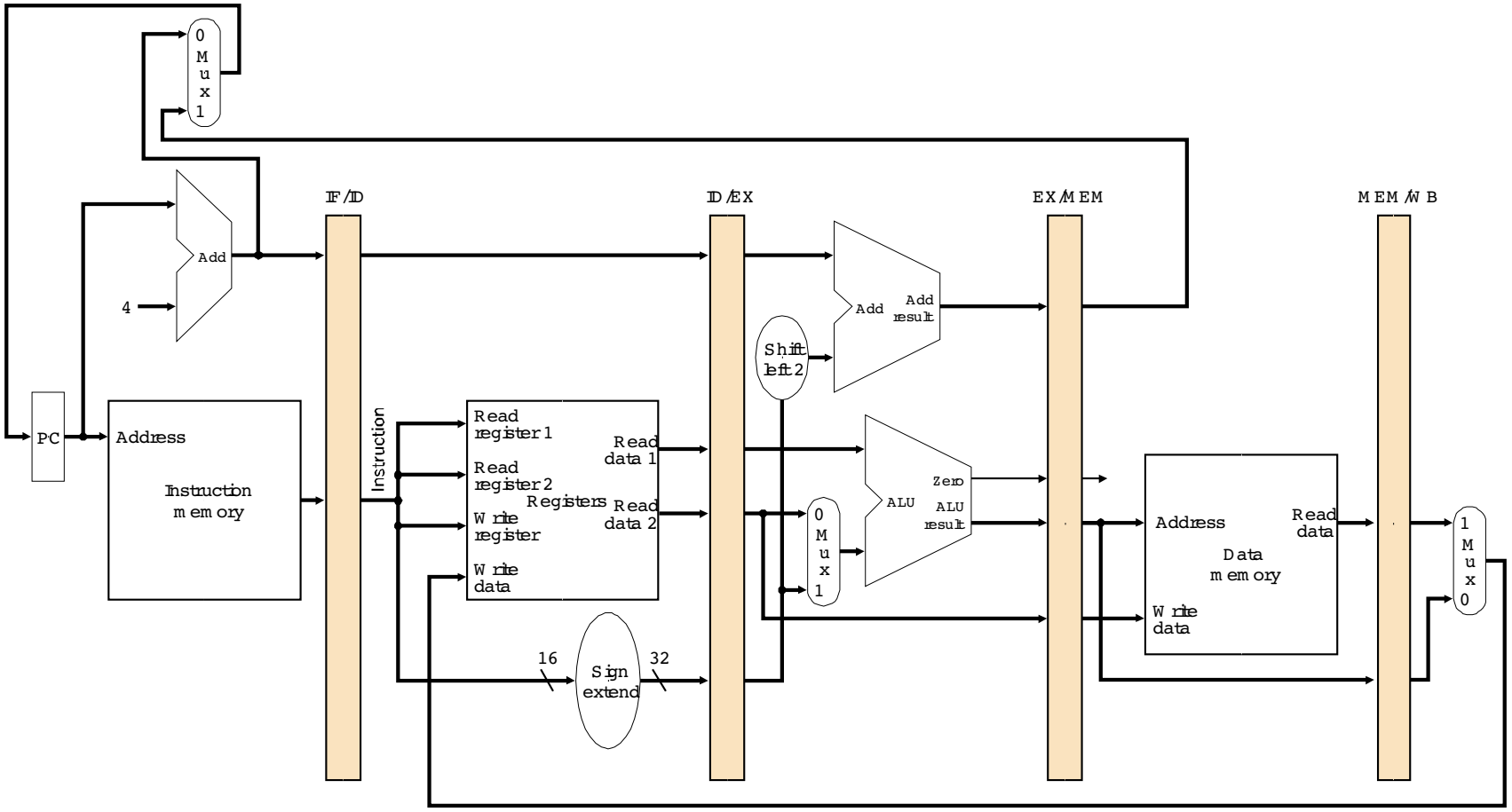
The Pipeline in Execution

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



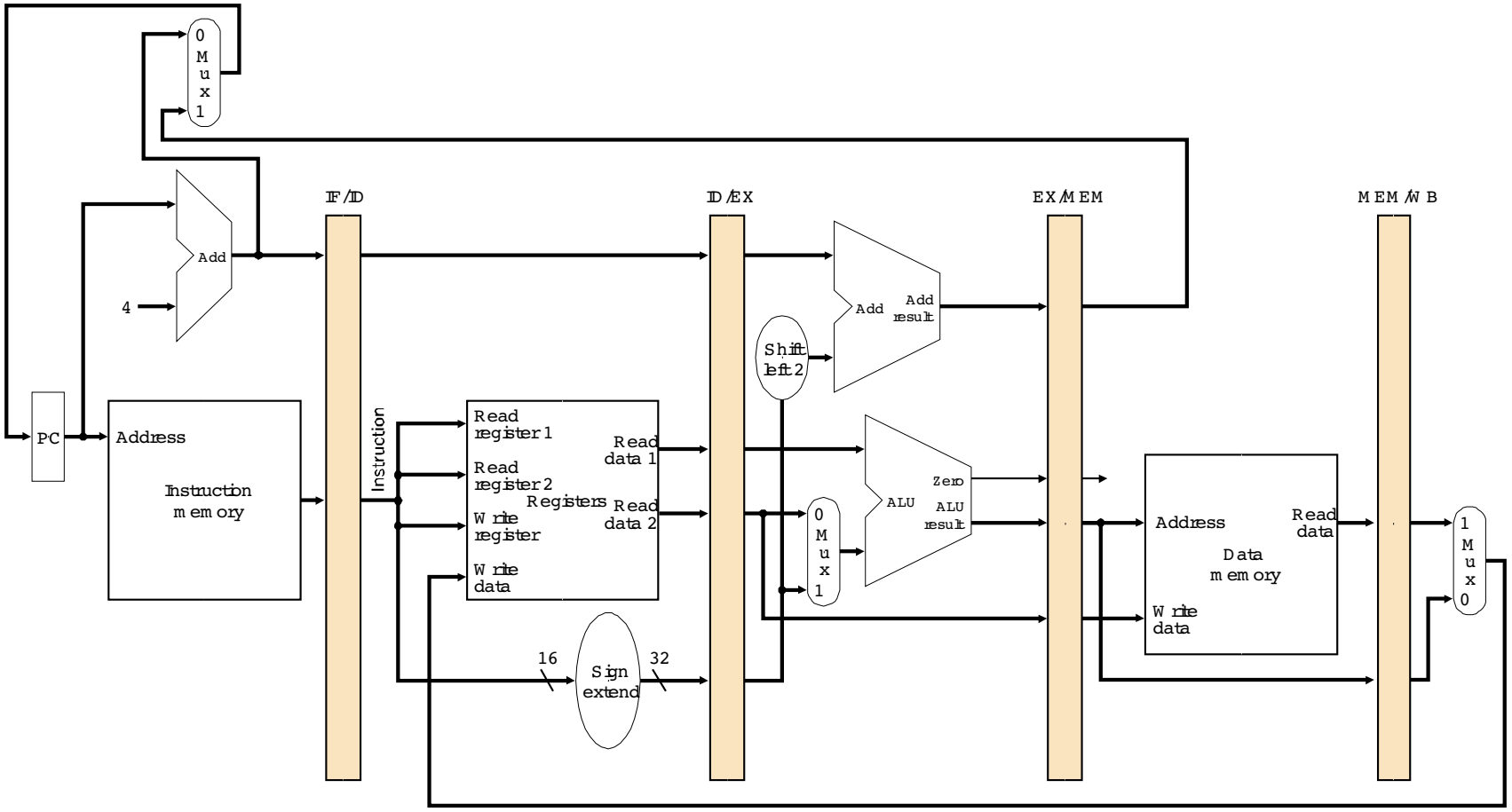
The Pipeline in Execution

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



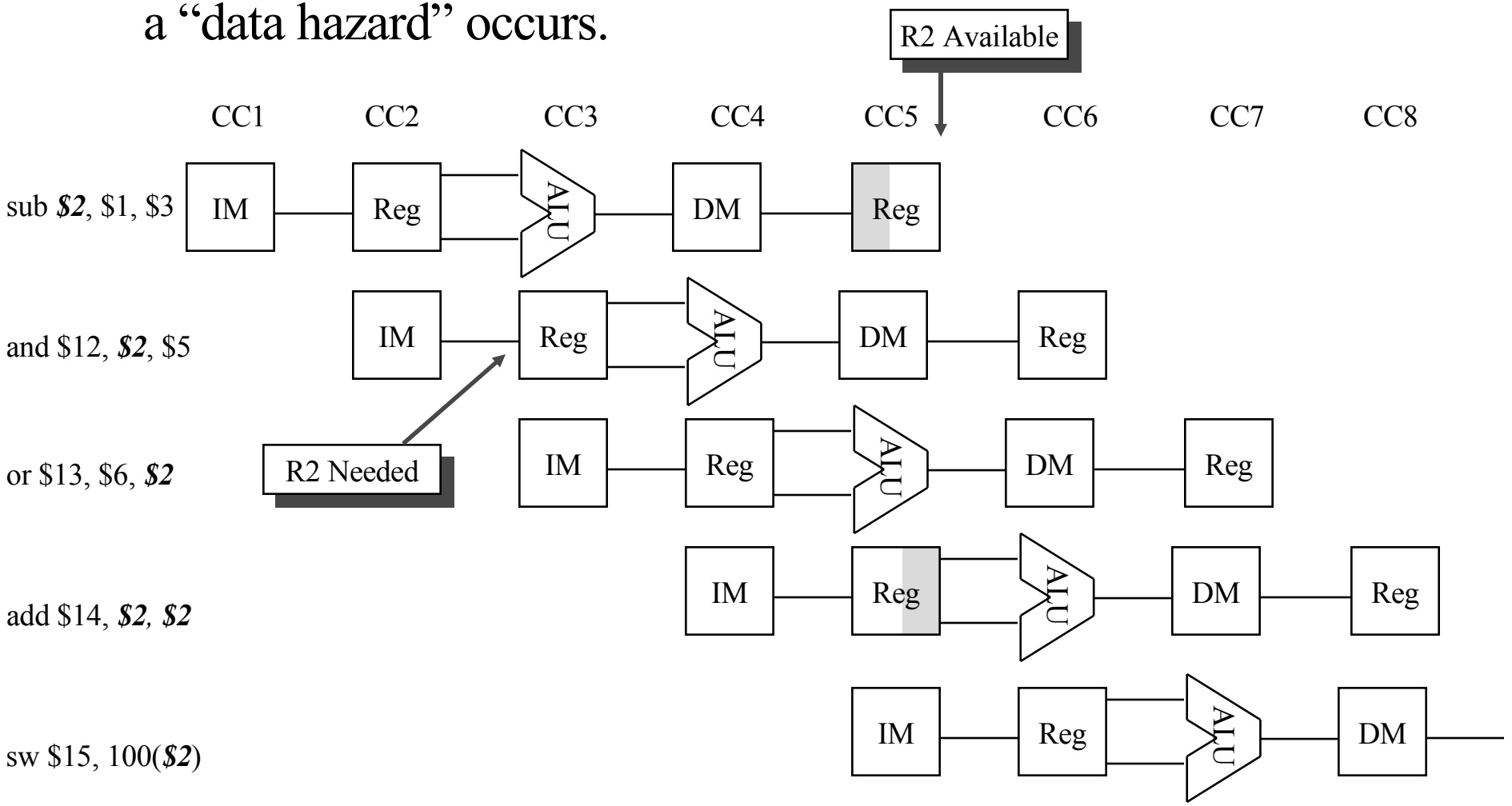
The Pipeline in Execution

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.



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).
- Pipelining uses combinational logic (and registers to propogate) to generate control signals.
- Pipelining creates potential hazards.