Chapter 4

The Processor

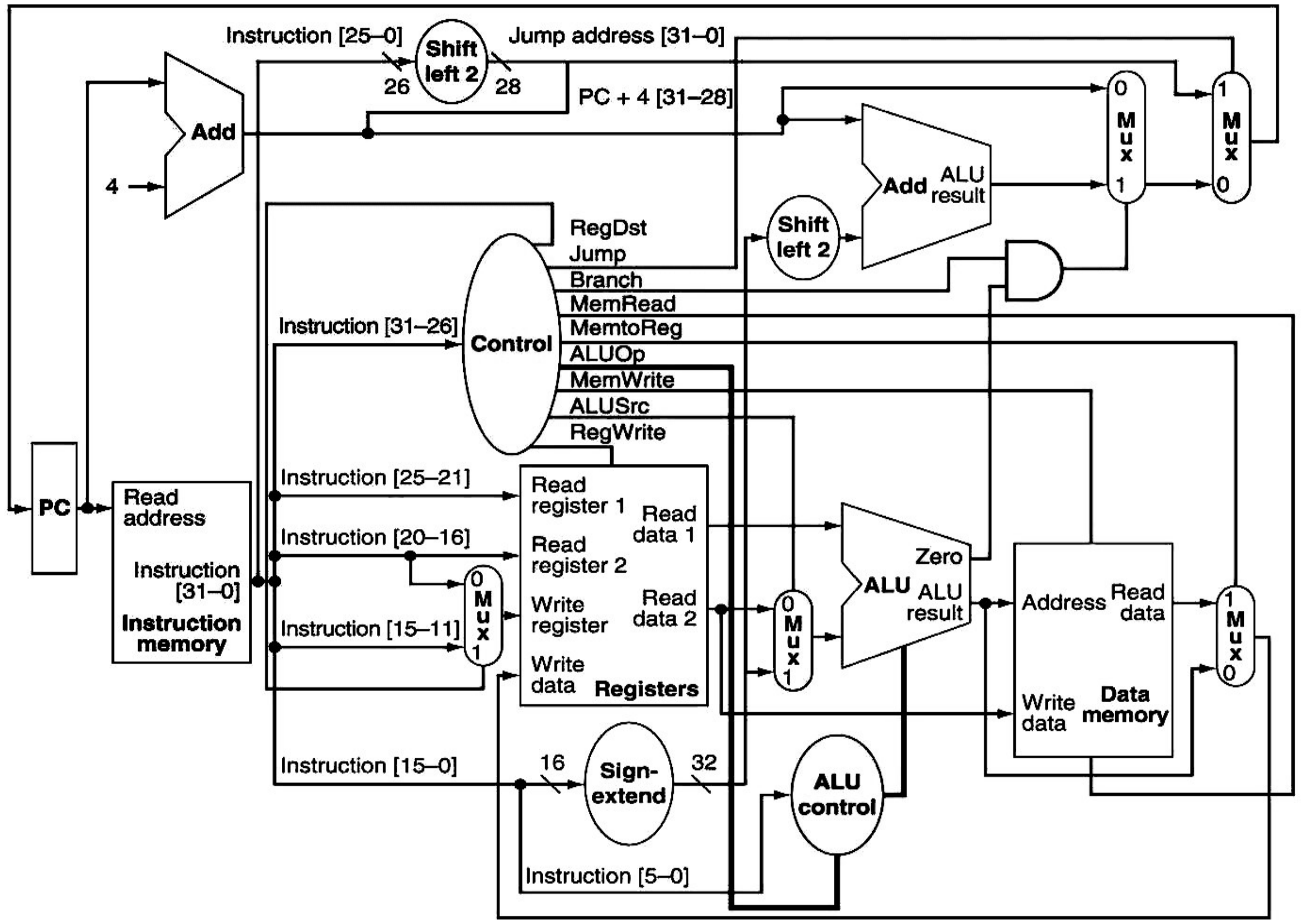
Lecture 7

Dr. Karim Emara

Agenda

- Introduction to Pipelining
- Pipelined Datapath
- Pipelined Control

Single Cycle MIPS



Single-cycle Performance

- In single-cycle datapath, different instructions have different execution paths. Look at the table below
- What is the suitable cycle time??
  - Longest delay

Instr	Instr fetch	Register read	ALU op	Memory access	Register write
lw	√	√	√	√	√
sw	√	√	√	√	
R-format	√	√	√		√
beq	√	√	√		

Performance Issues

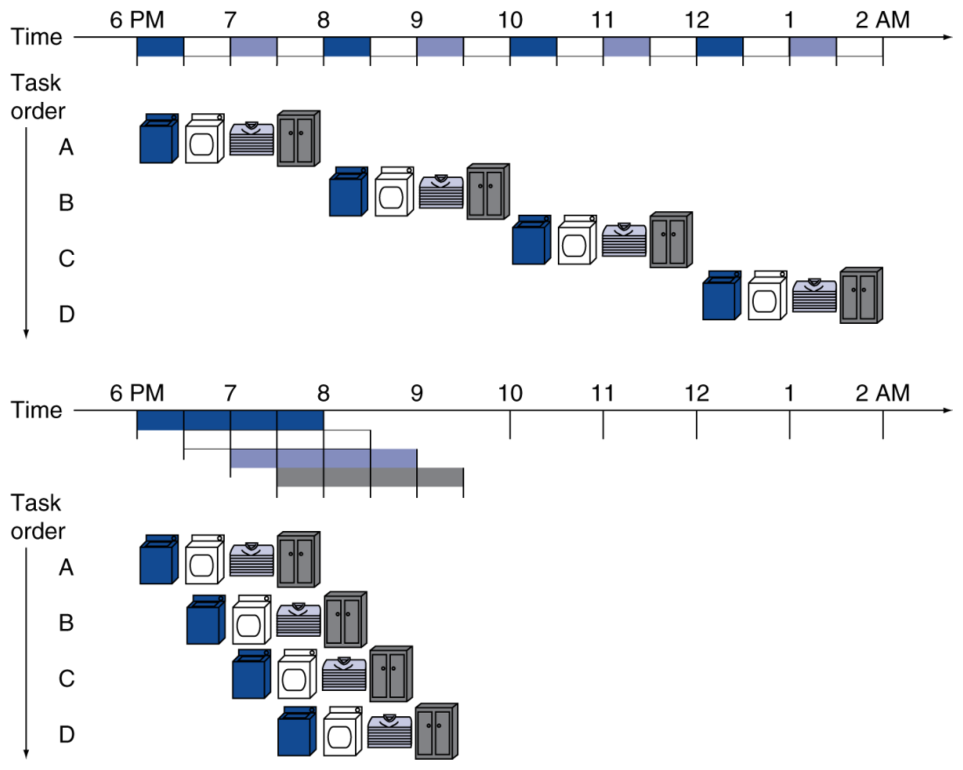
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- **Not feasible to vary period** for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

MIPS Pipeline

- The concept of pipeline is to divide the instruction into **N stages**
- Stages can run in parallel on **different** instructions
- Cycle time equals to the **longest stage**
  - Remember in single-cycle, cycle time equals to the longest instruction

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup

    = non-pipe. / pipelined

    = 8/3.5 = 2.3
- Non-stop:
  - Speedup

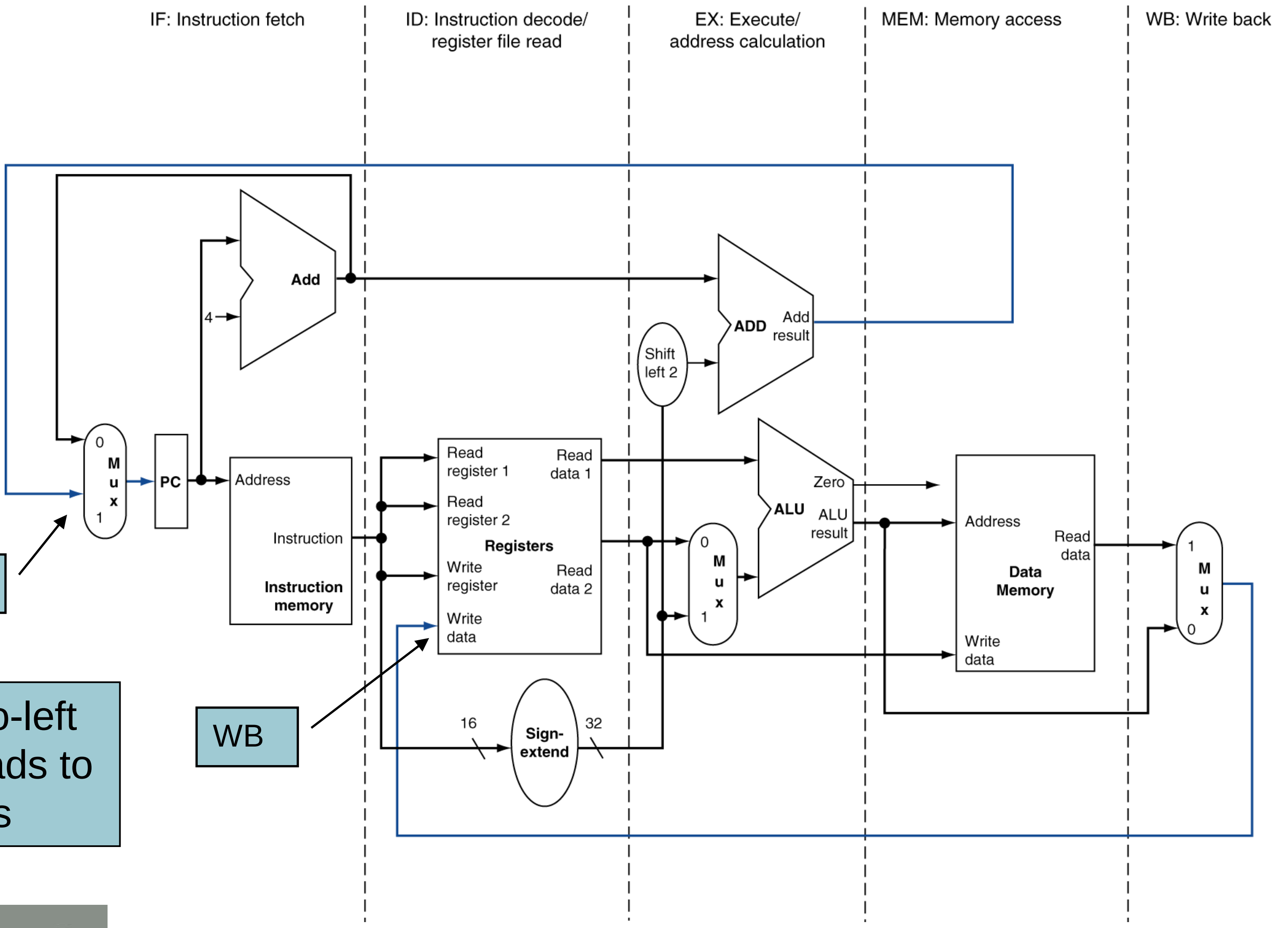
    = 2n/0.5n + 1.5
  - ≈ 4 (for large n)

    = number of stages

MIPS Pipeline

- Five stages, 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
- Each stage runs in a single cycle
  - A stage performs simpler task → shorter cycle is possible

MIPS Pipelined Datapath



MEM

WB

Right-to-left flow leads to hazards

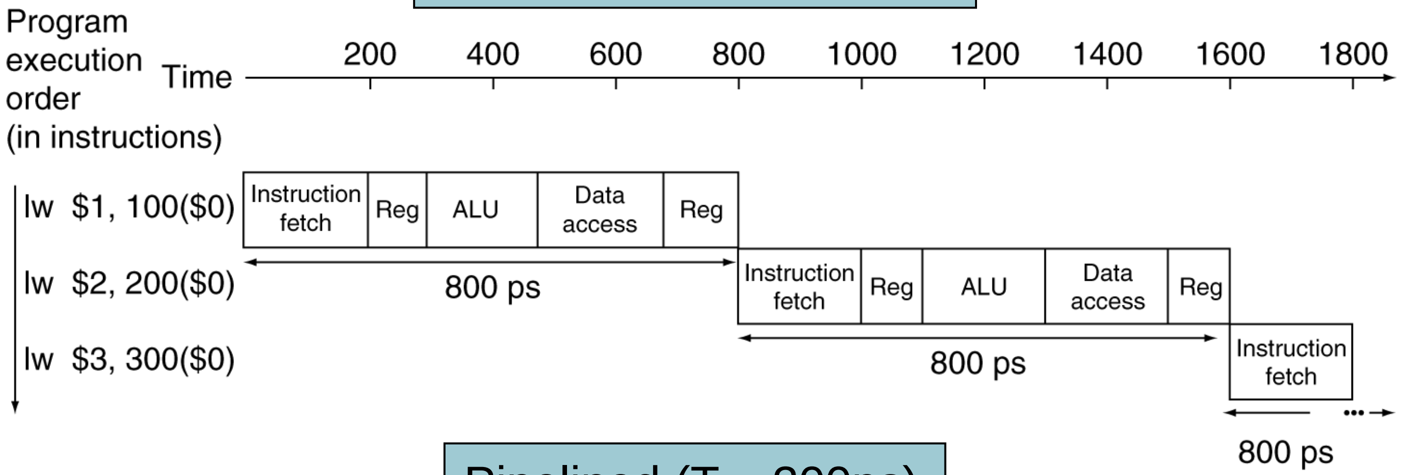
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

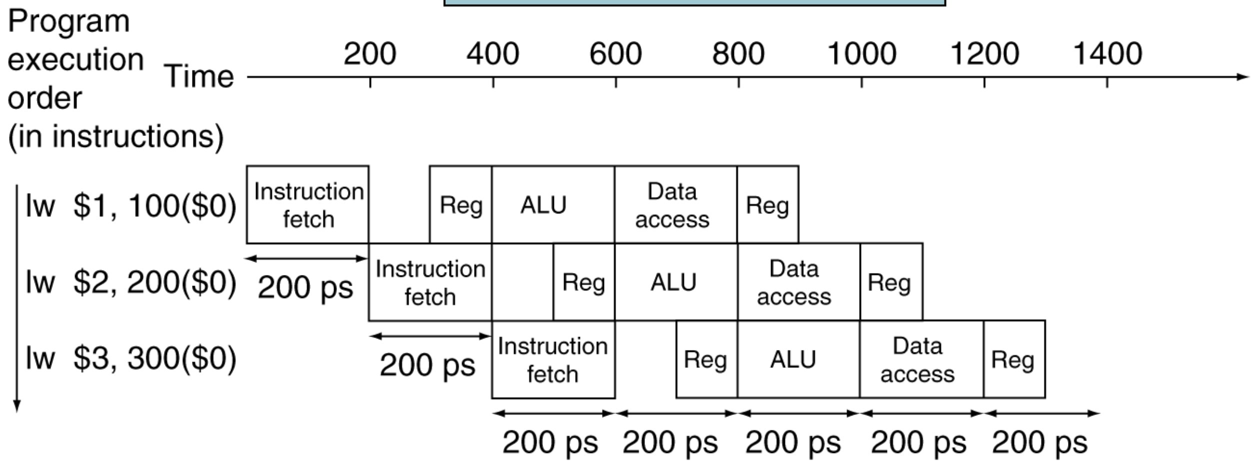
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

- If all stages are balanced
  - i.e., *all stages take the same time*
  - $$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Hazards

- Situations that **prevent** starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

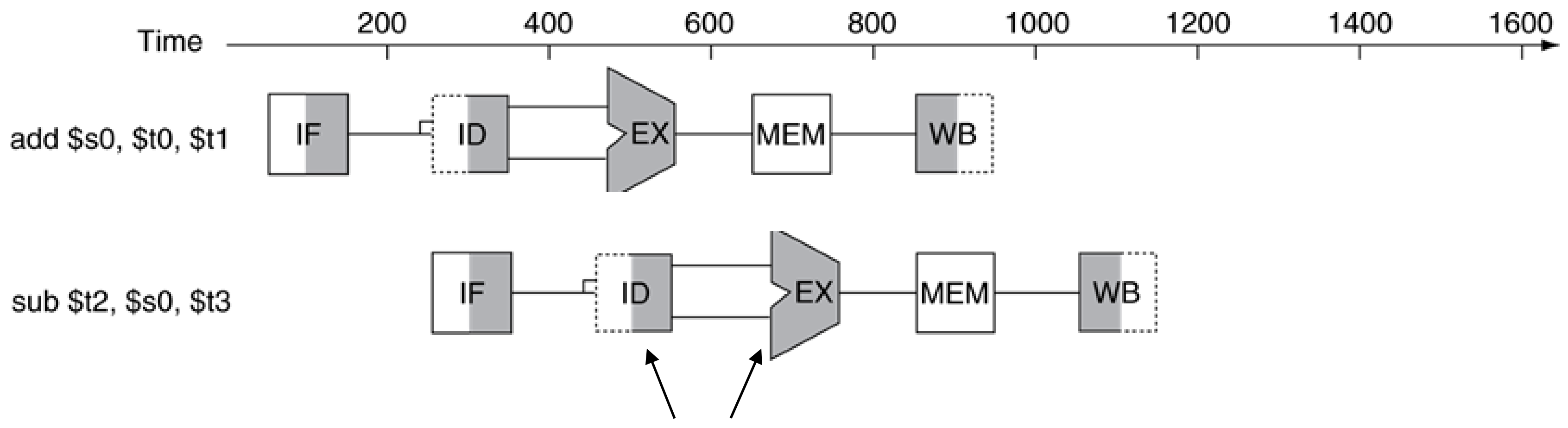
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

- An instruction depends on completion of data access by a previous instruction

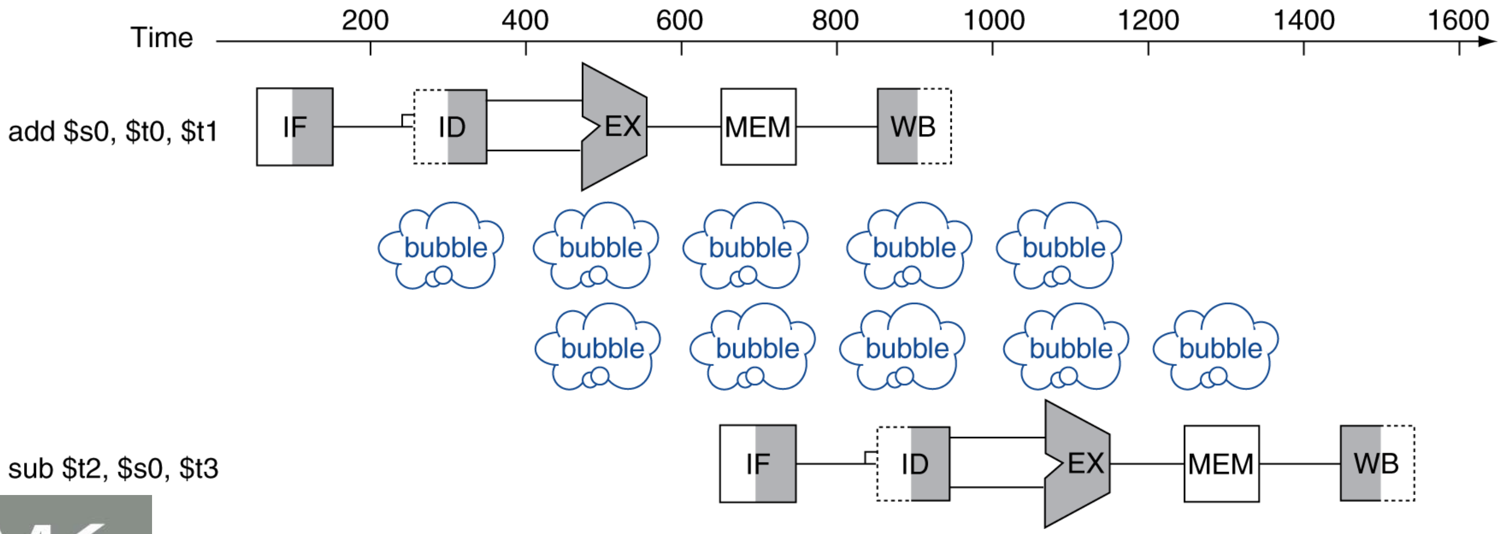
  -
    ```
    add  $s0, $t0, $t1
    sub  $t2, $s0, $t3
    ```



Old $s0 value

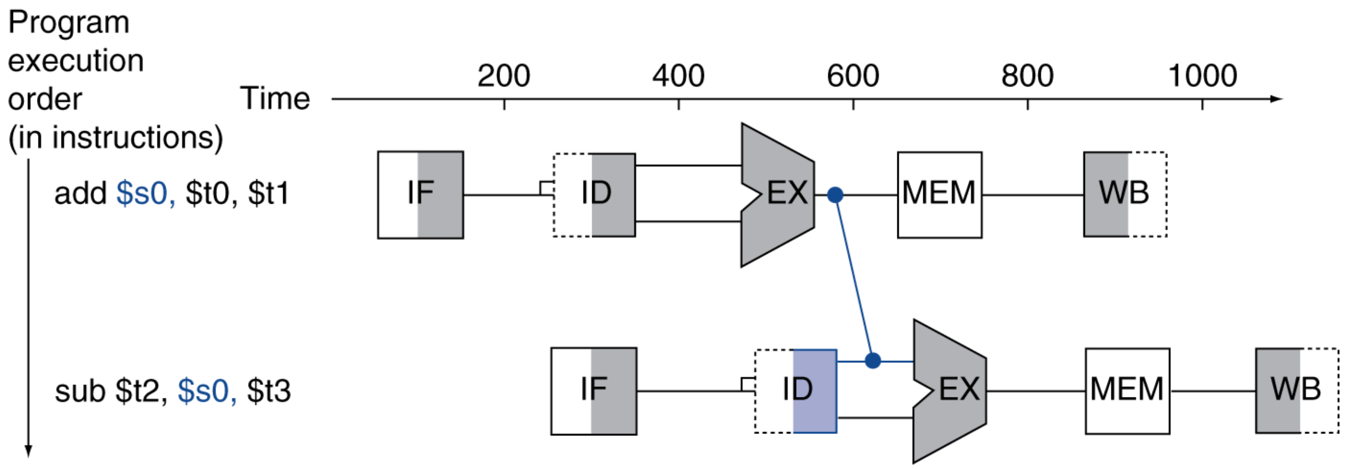
Data Hazards

- Data hazard can be eliminated by **stalling** the pipeline
  - `add    $s0, $t0, $t1`
  - `Nop`
  - `Nop`
    `sub    $t2, $s0, $t3`



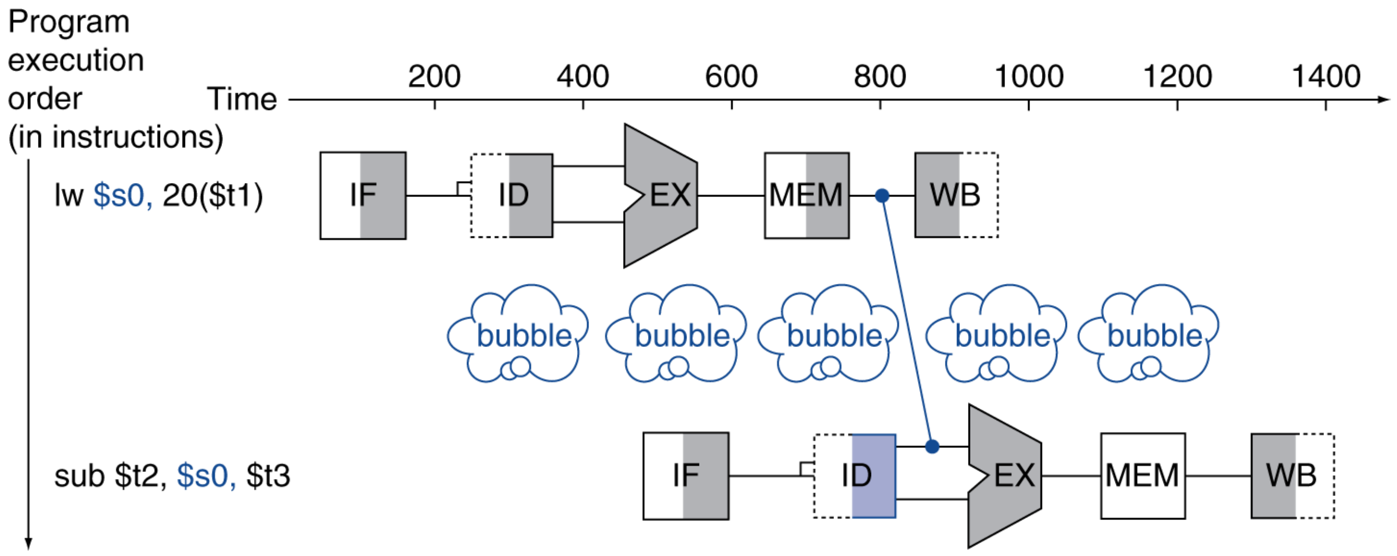
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



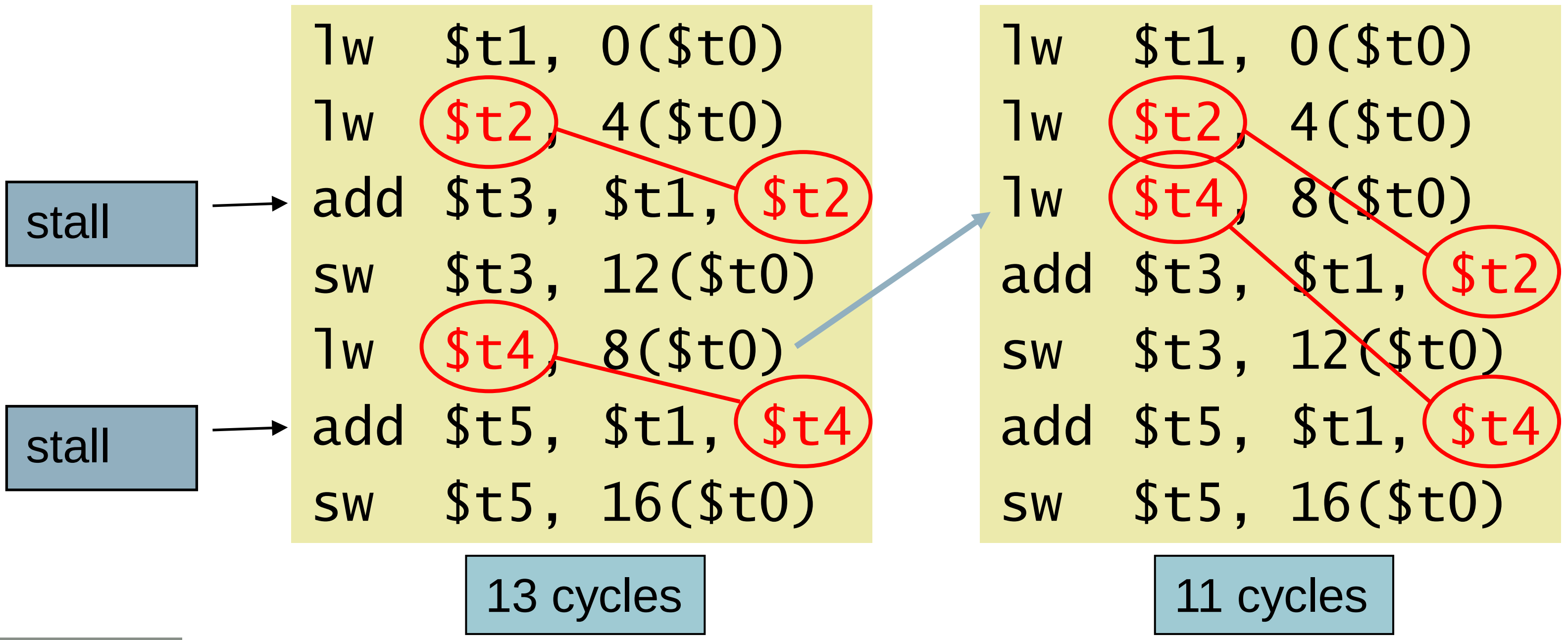
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A `A = B + E; C = B + F;`



```
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)
```

13 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

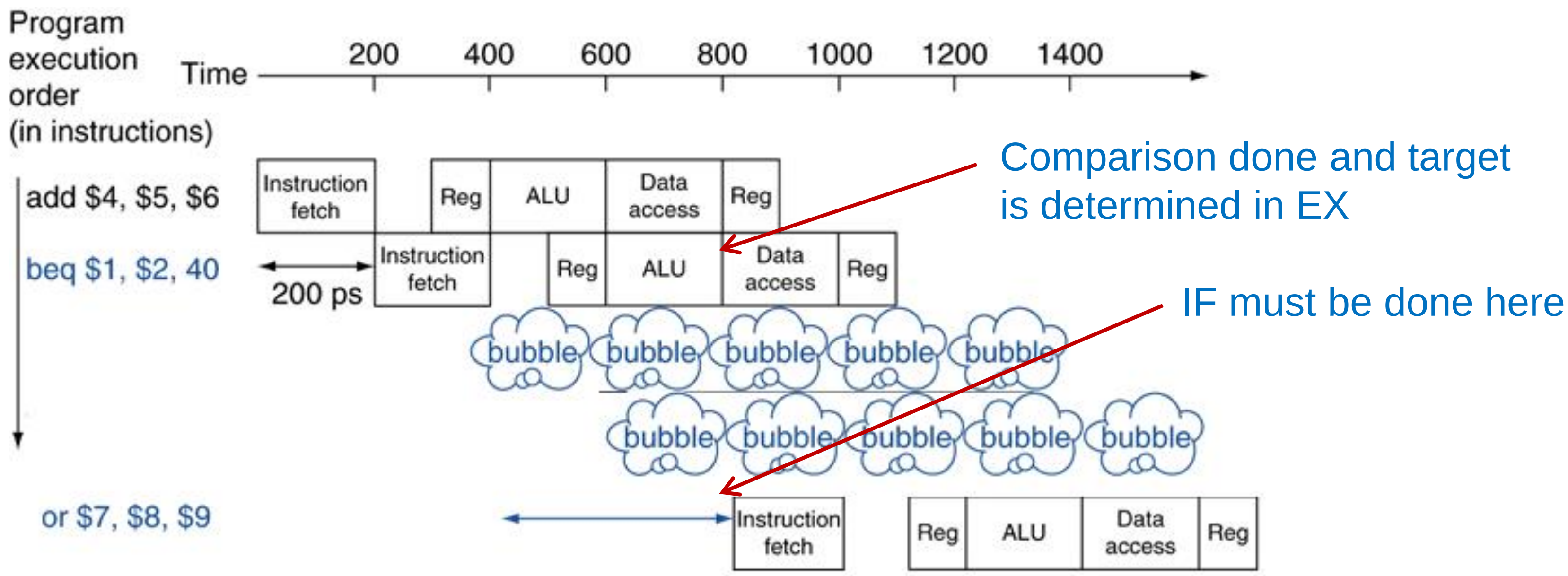
11 cycles

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

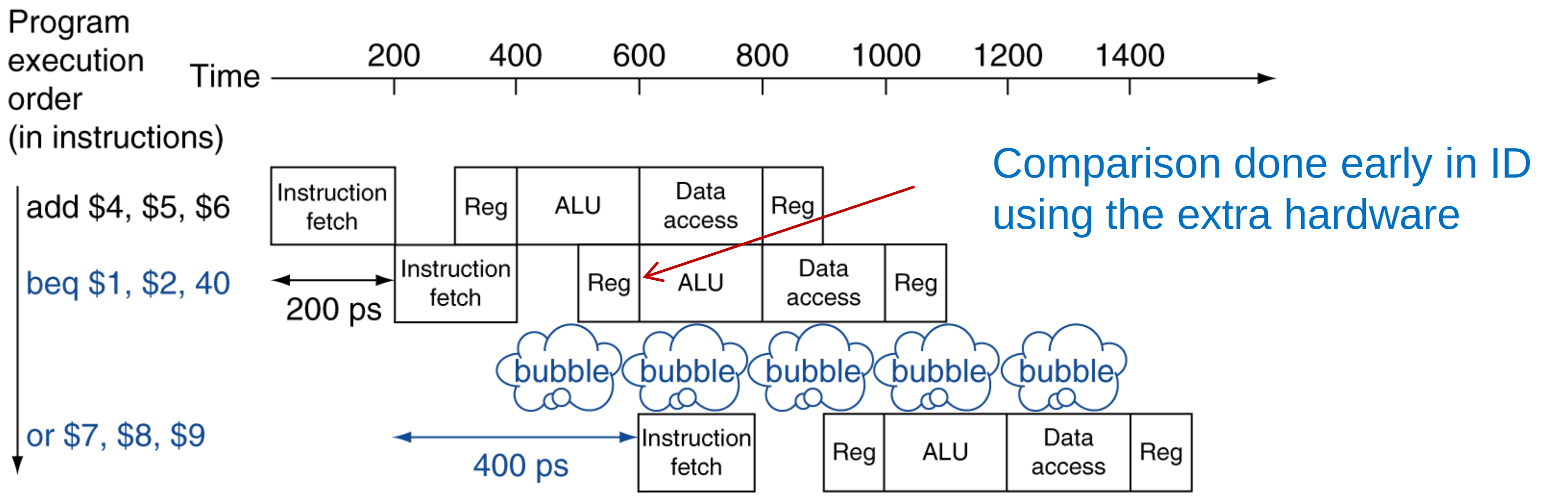
Stall on Branch

- Without extra hardware, 2 stalls are needed to fetch the correct instruction after branch instruction



Stall on Branch

- Wait until branch outcome determined before fetching next instruction



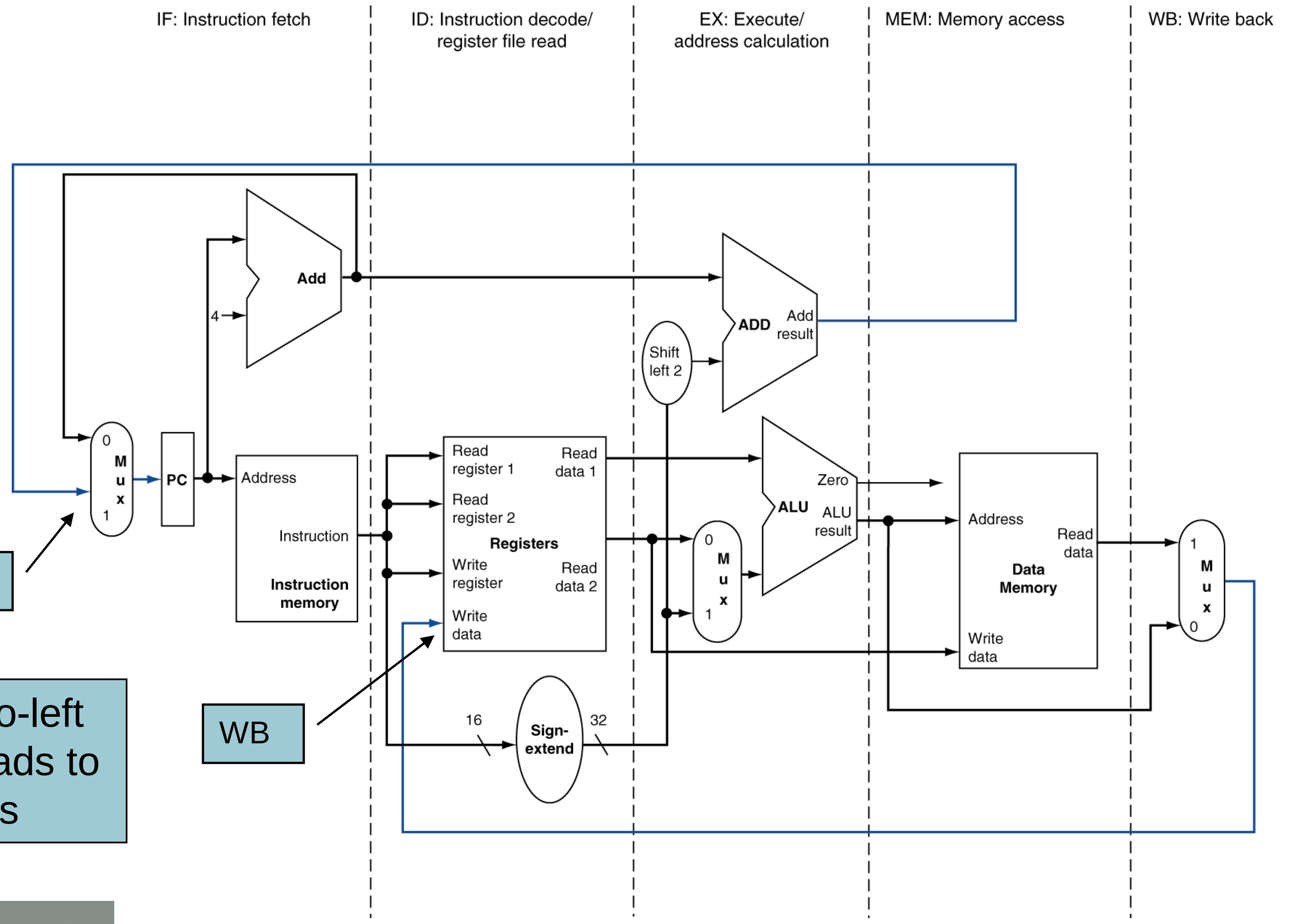
Agenda

- Introduction to Pipelining
- Pipelined Datapath
- Pipelined Control

Pipelined Datapath

- 5 stages → upto 5 instructions running in parallel
- Instruction and data move from left to right except in
  - write-back stage and
  - updating the PC value (+4 or branch)
- Right-to-left flow can lead to data hazard (WB) or control hazards (PC update)

MIPS Pipelined Datapath



MEM

WB

Right-to-left flow leads to hazards

Pipeline registers

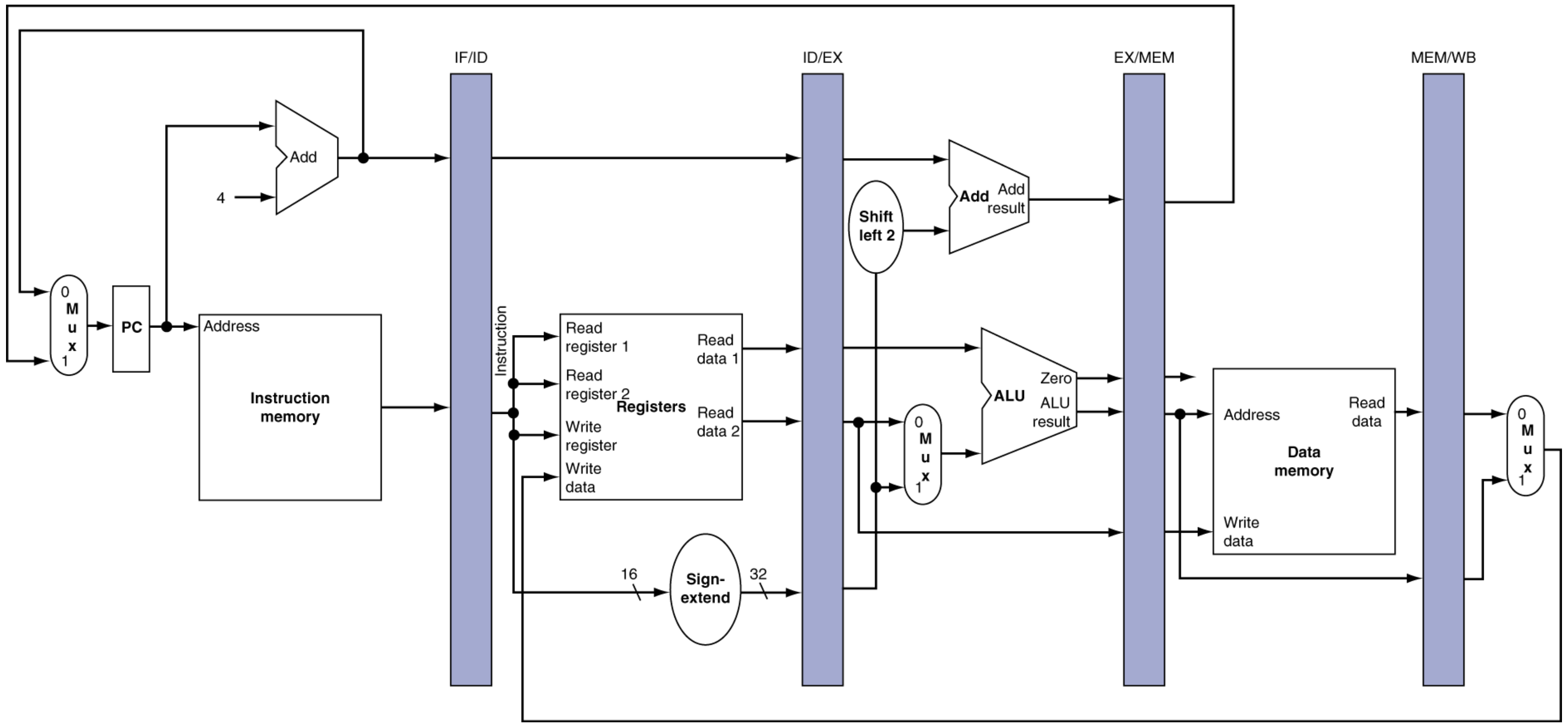
- Since the datapath is **combinational**, data/control of the previous instruction disappear next cycle!
- Thus **pipeline registers** are needed to keep them to serve the instruction in the later stage
- All instructions advance from one pipeline register to the next every cycle.

Pipeline registers

- The registers are named for the two stages separated by that register.
- For example, the pipeline register between the IF and ID stages is called IF/ID, and so on.
- There are no pipeline register after
  - WB because the state will be already written in a CPU register anyway
  - PC because it is updated **every cycle**

Pipeline registers

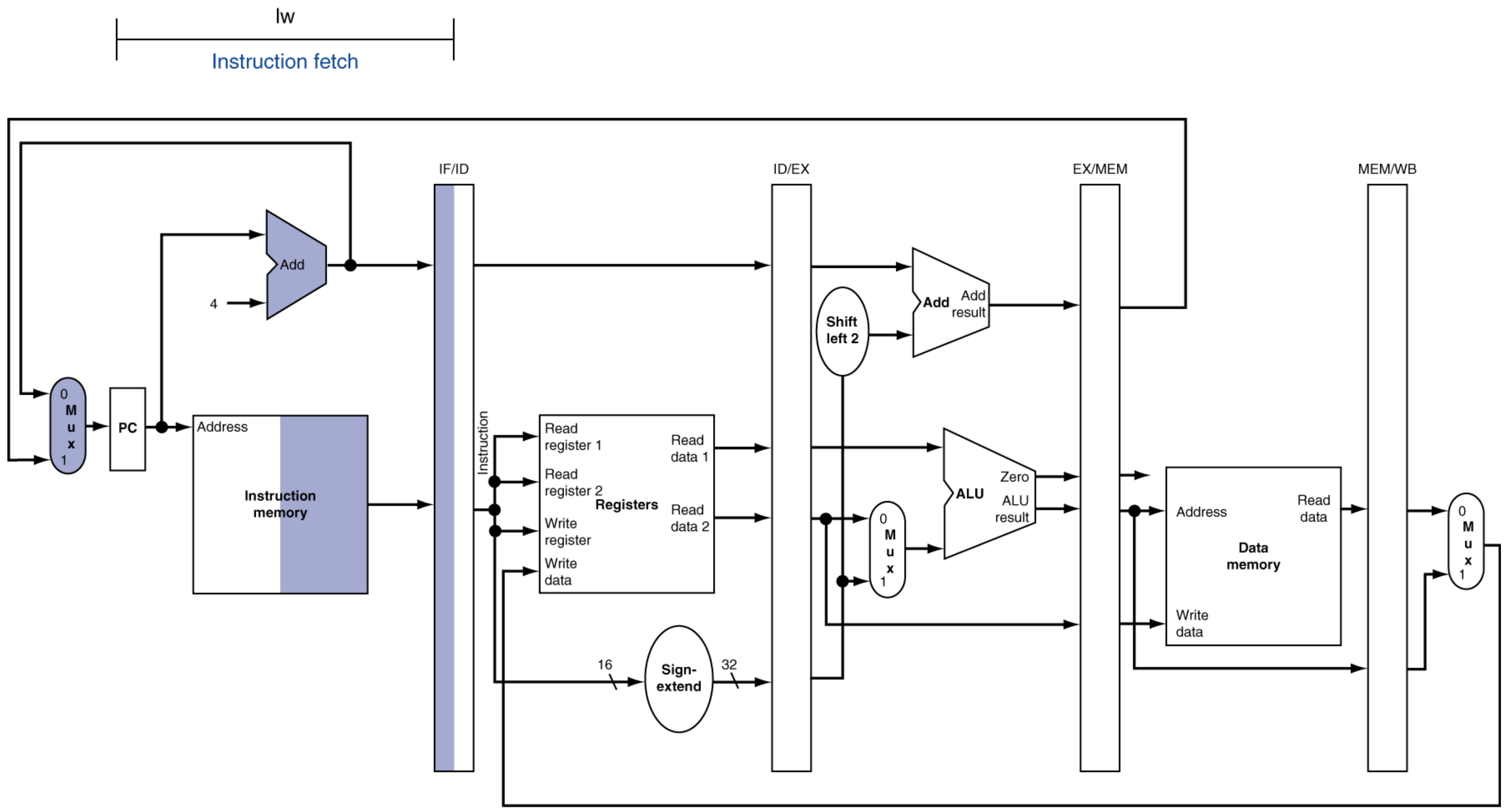
- Need registers between stages
  - To hold information produced in previous cycle



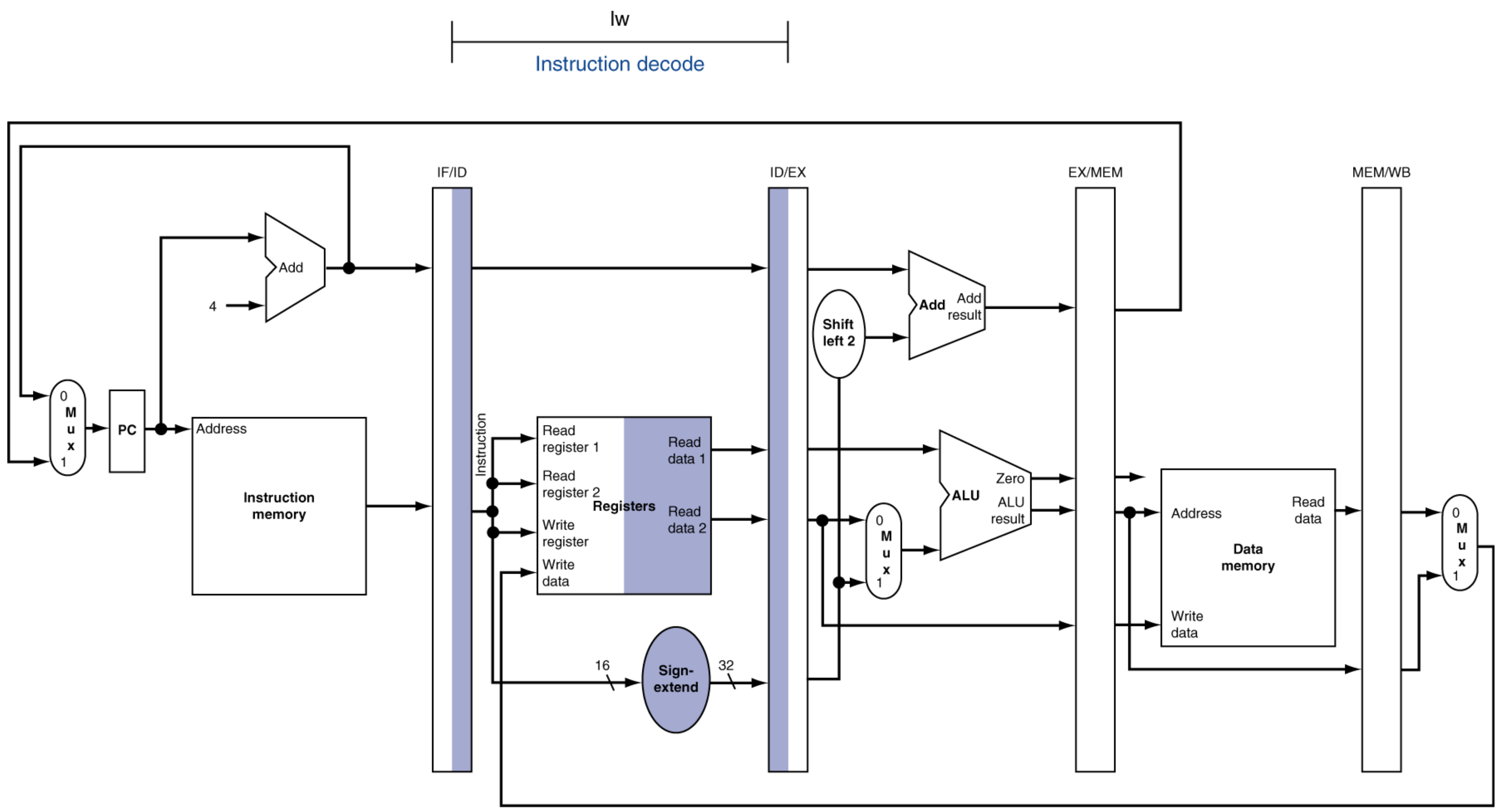
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store

IF for Load (single-clock-cycle)

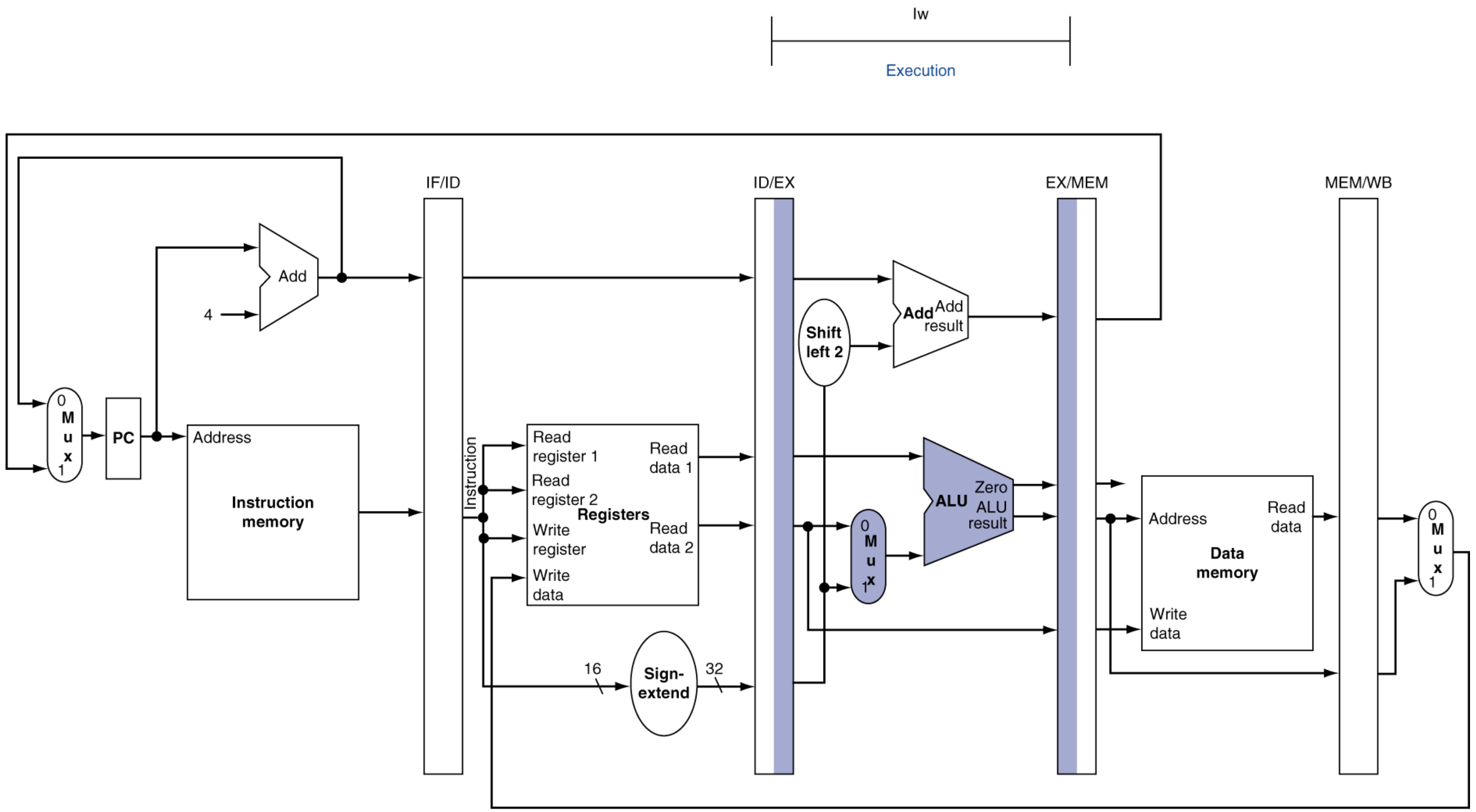


ID for Load

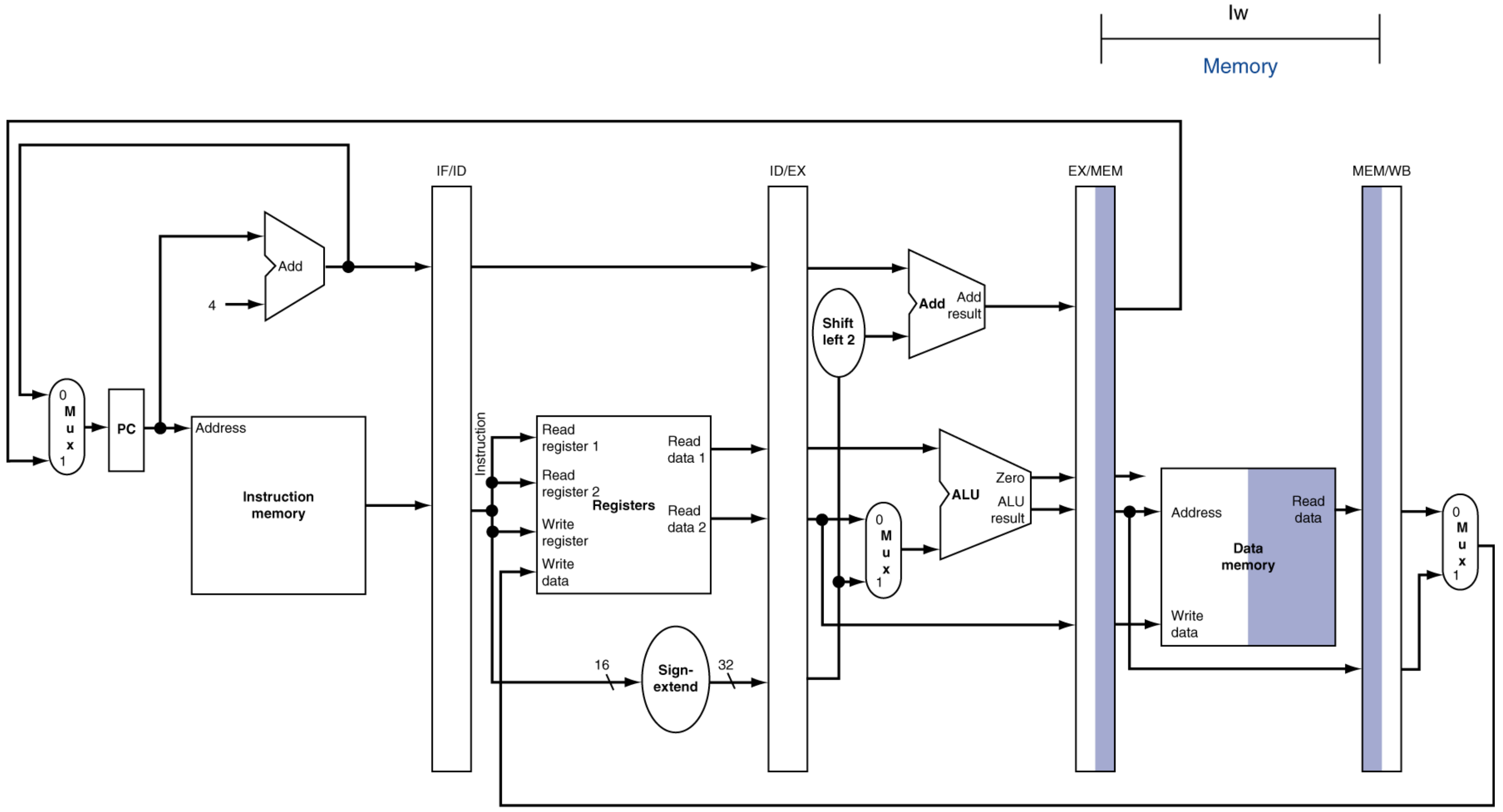


lw
Instruction decode
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

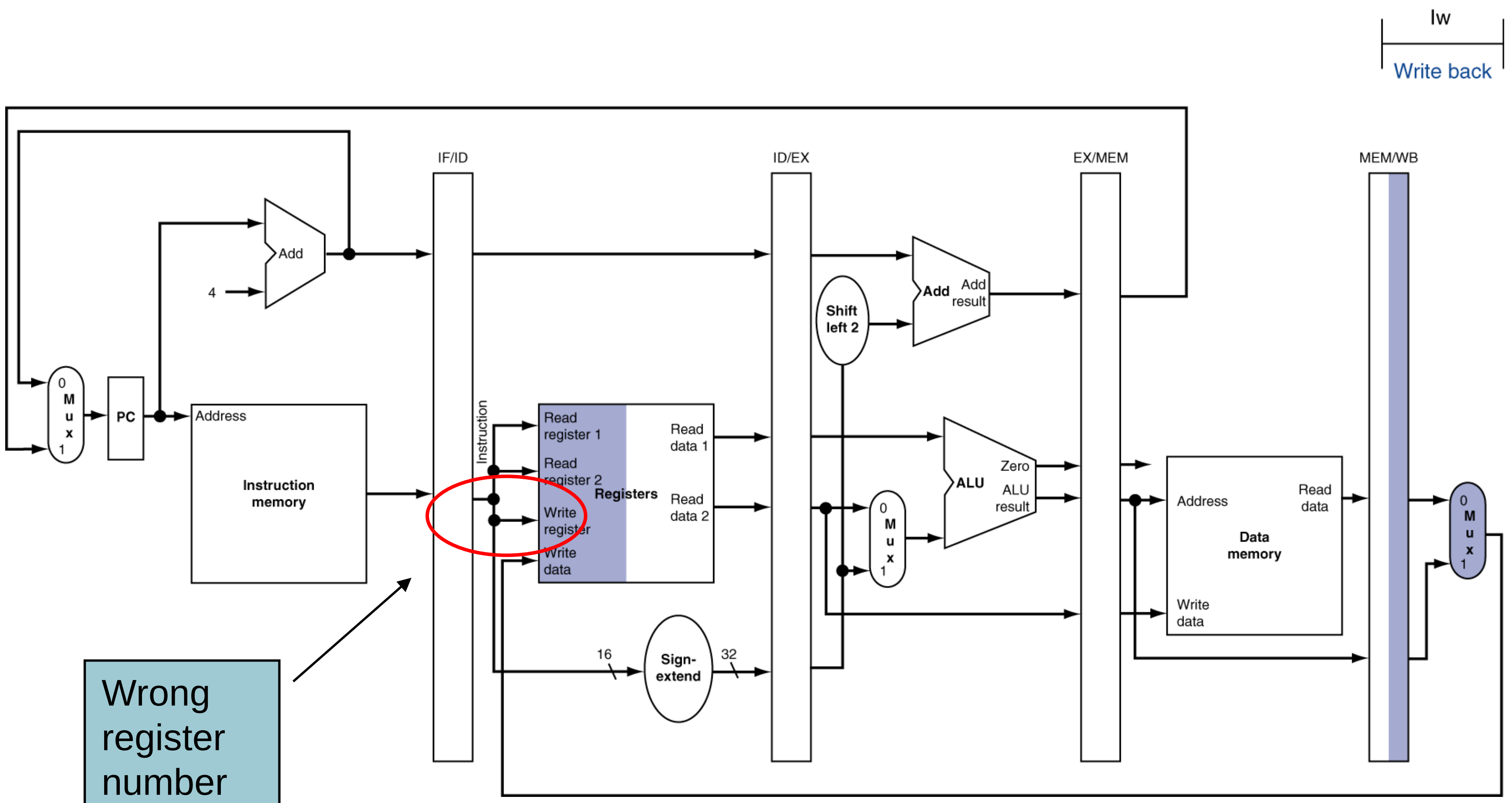
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Write data
Read data

MEM for Load

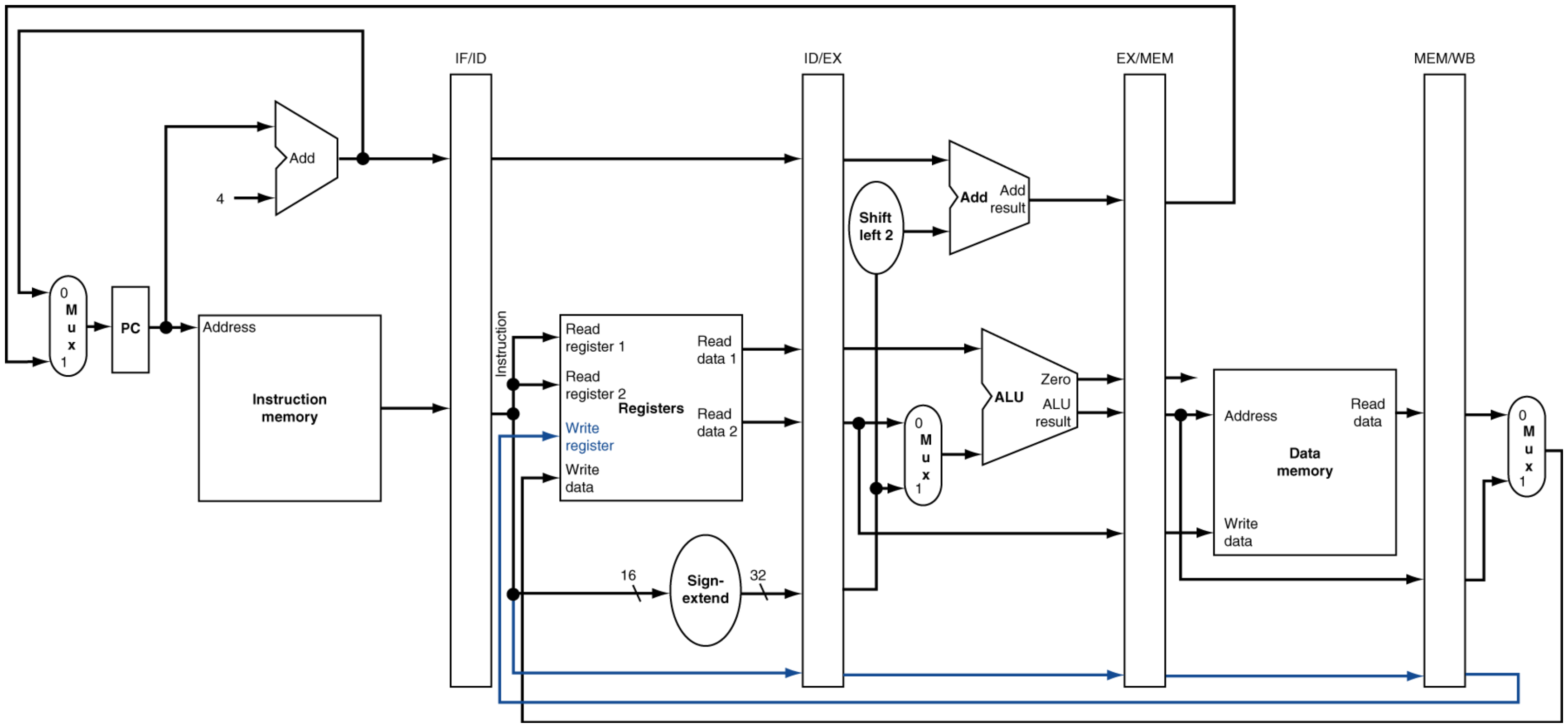

lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

WB for Load



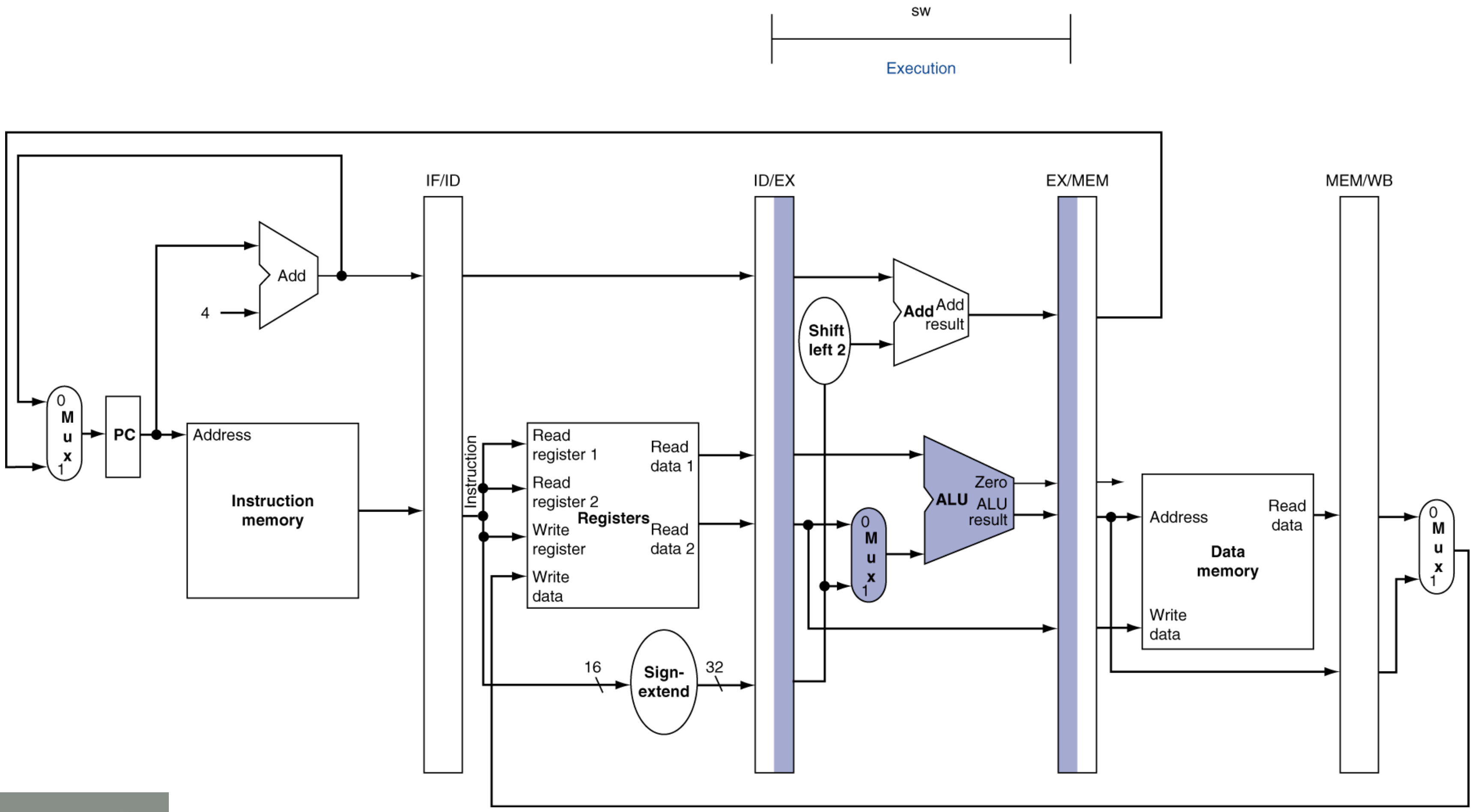
Wrong register number

Corrected Datapath for Load

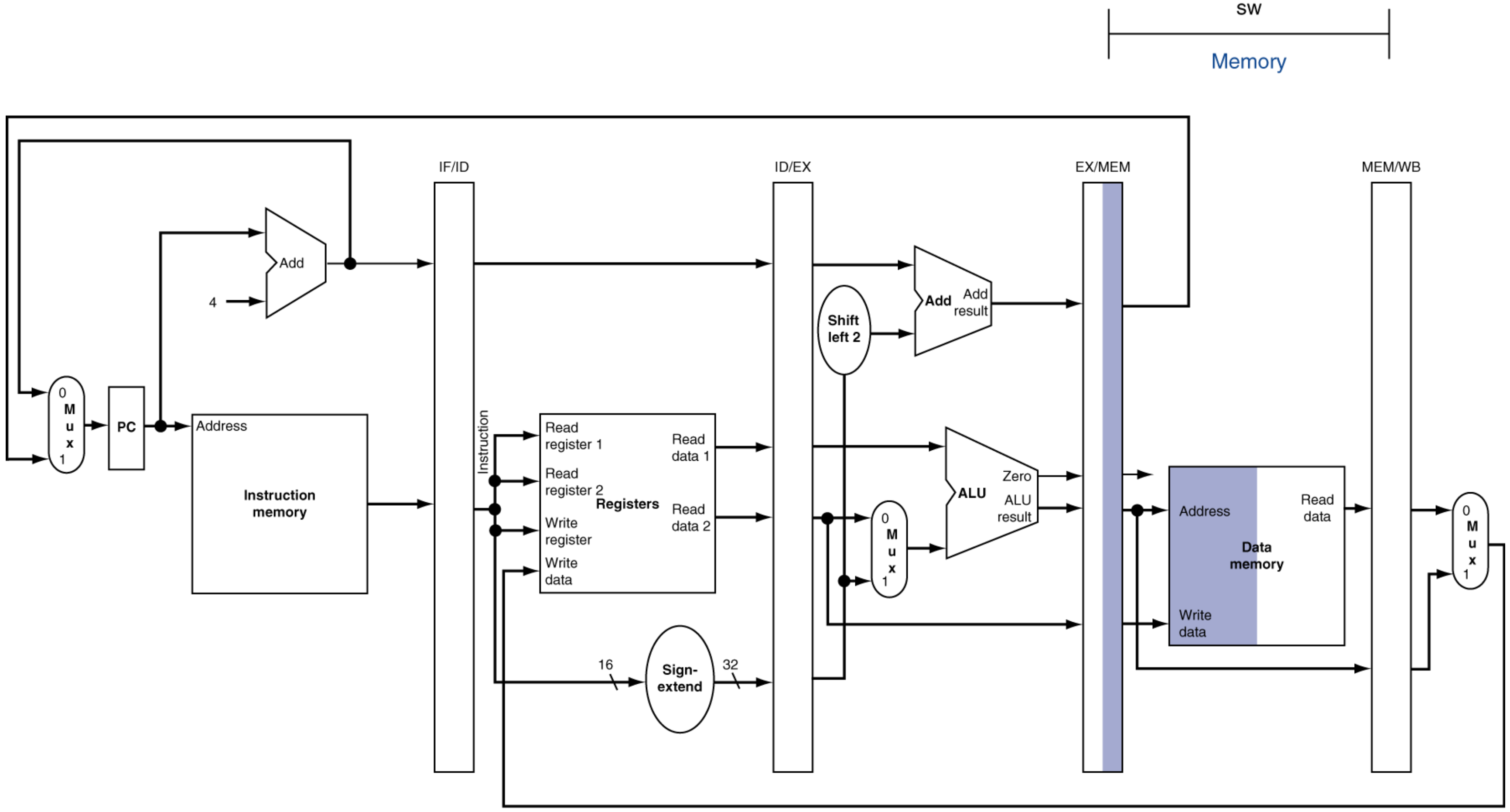


EX for Store

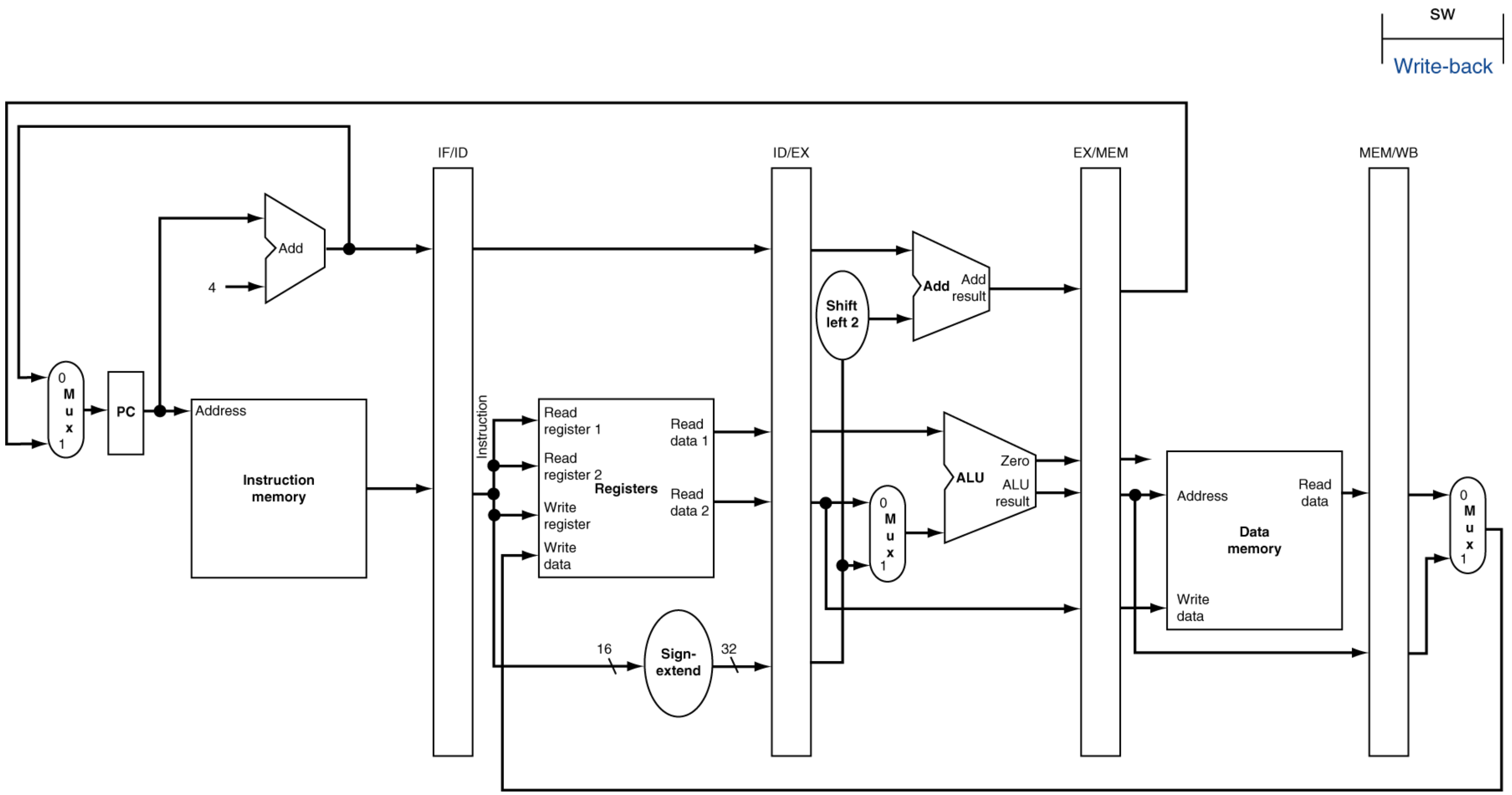
Note: IF and ID for store are exactly the same as Load



MEM for Store



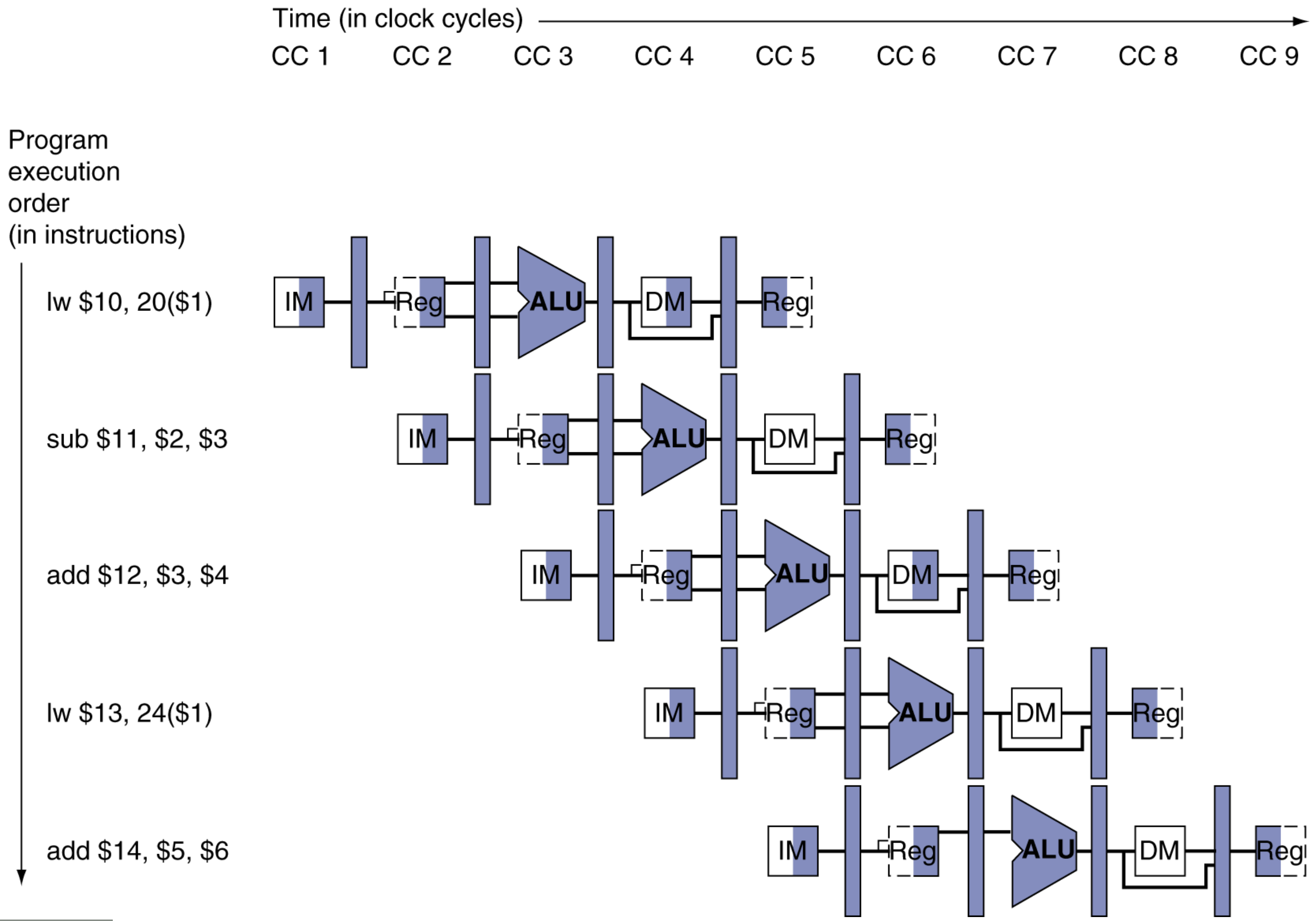
WB for Store



sw
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
0 M u x 1

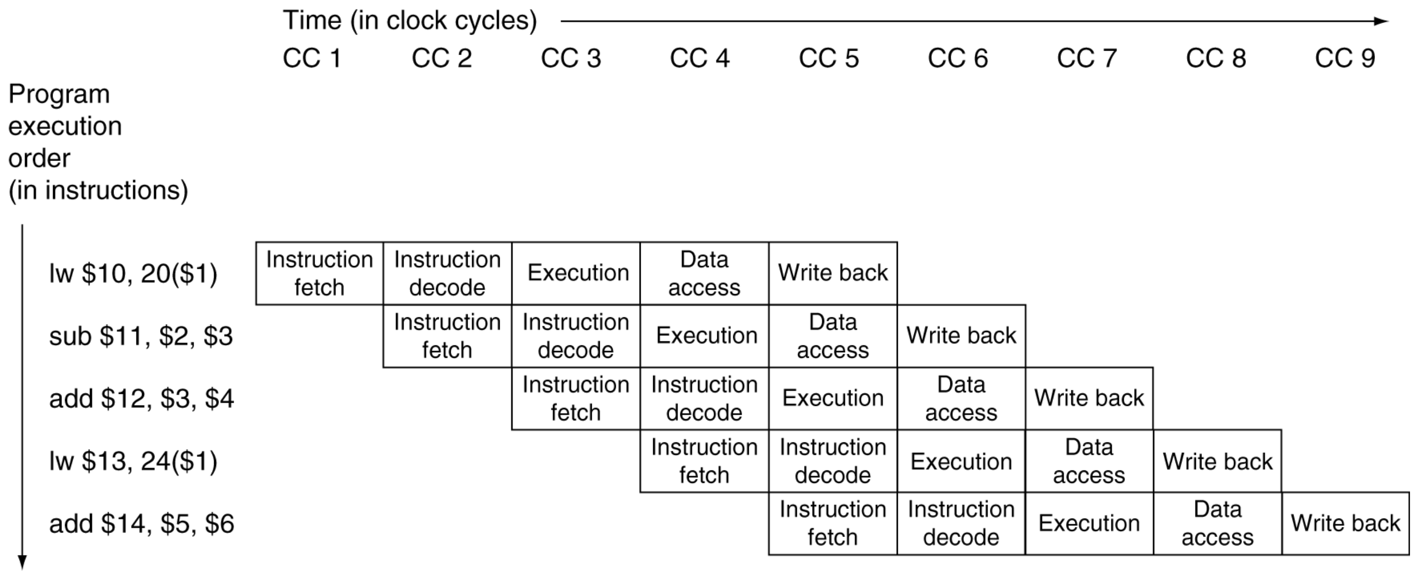
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

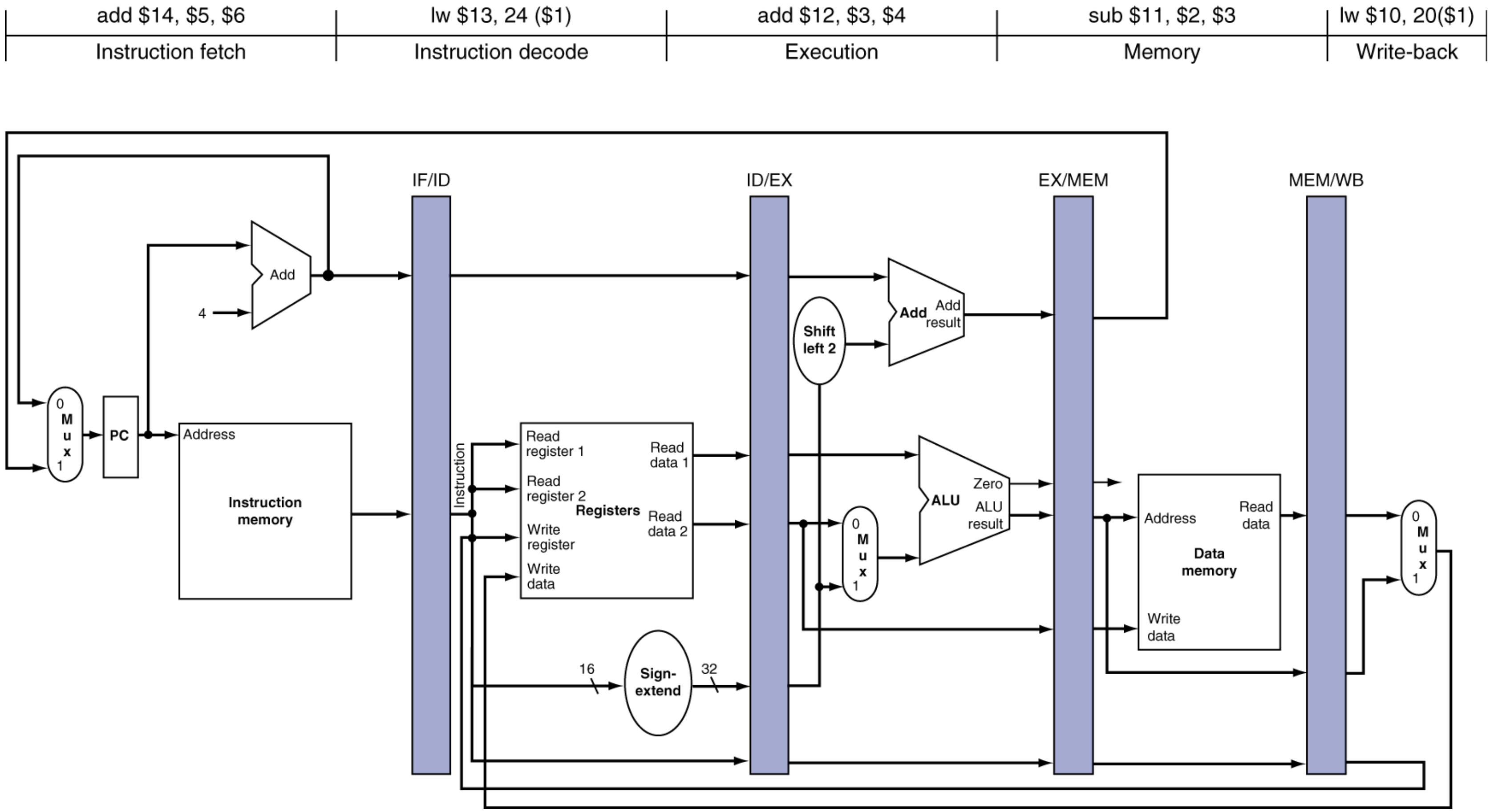
- Traditional form



	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Single-Cycle Pipeline Diagram

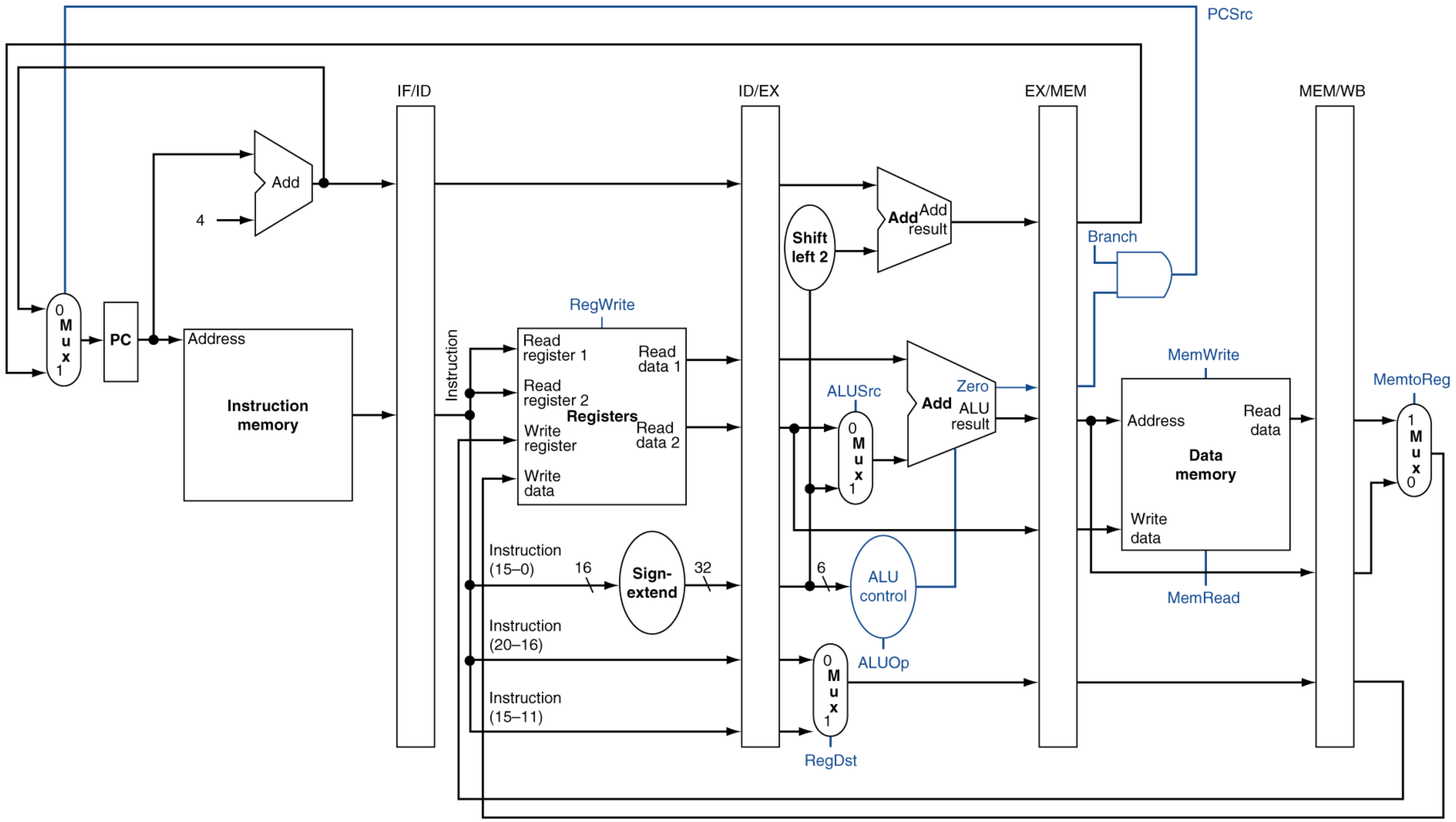
- State of pipeline in a given cycle (Cycle 5)



Agenda

- Introduction to Pipelining
- Pipelined Datapath
- Pipelined Control

Pipelined Control Overview



Pipelined Control

- There are no control signal for PC and pipeline registers. Why?
  - Updated every clock cycle regardless the instruction
- Control lines are divided into groups according to the pipeline stage
- We can use the same values of control lines of single-cycle datapath
- The difference: control lines come from an early instruction

Pipelined Control

- Single-cycle

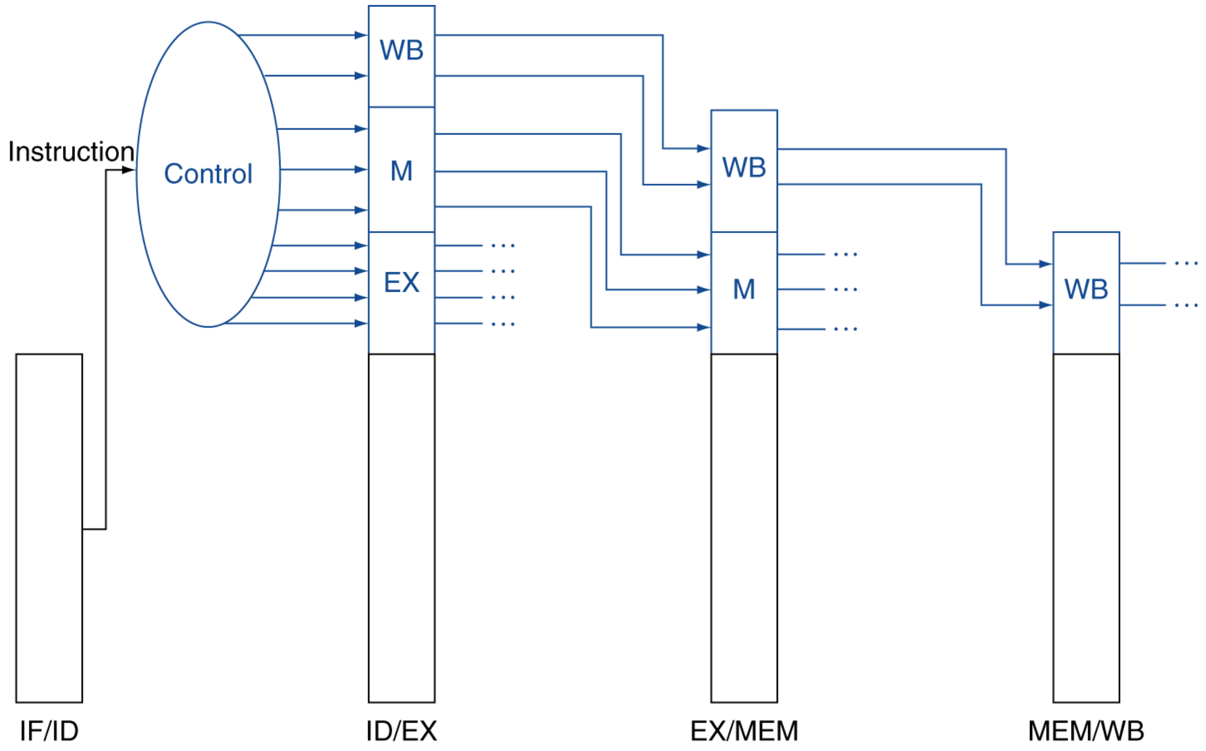
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- Pipelined

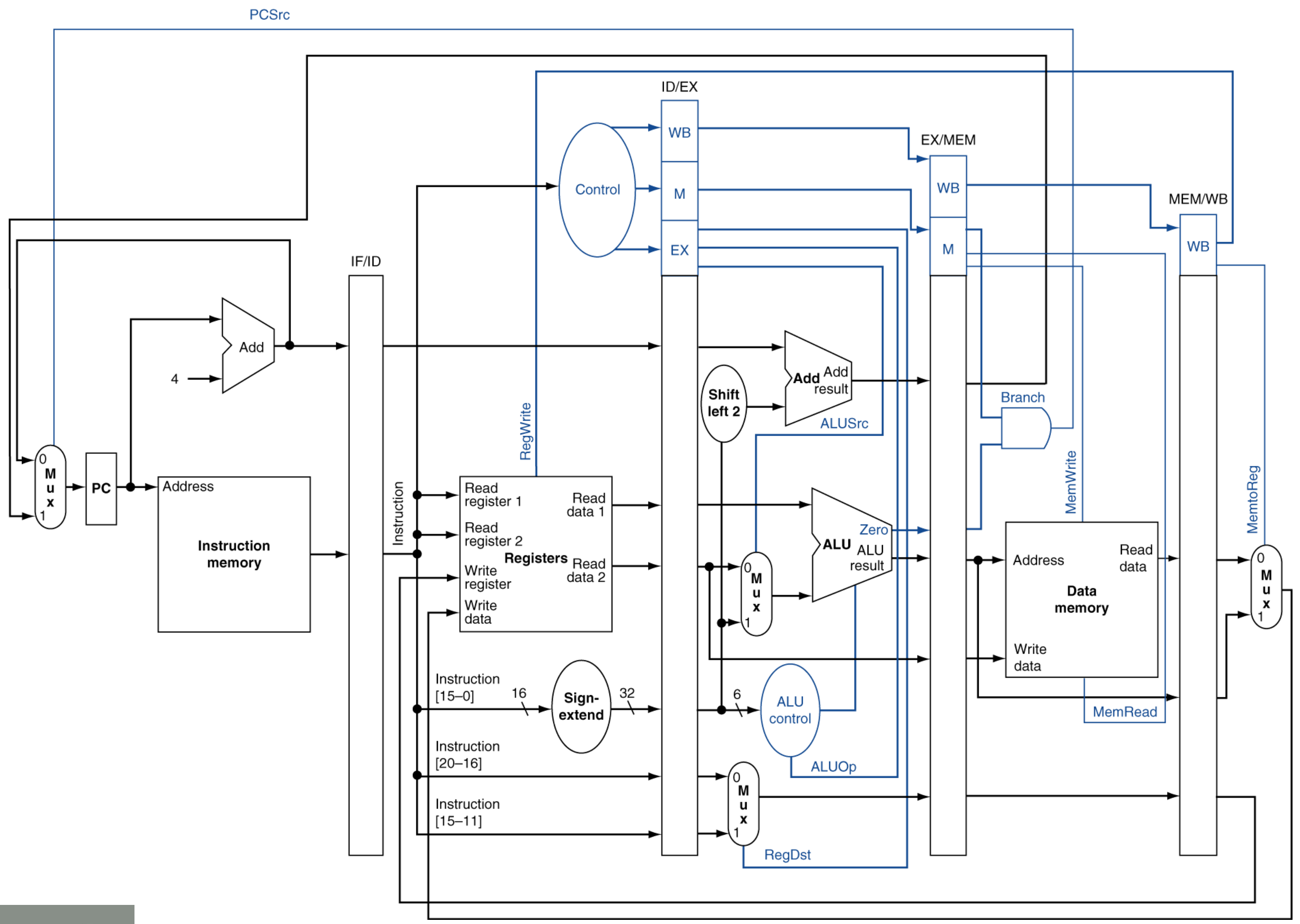
	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Control
Add
4
Add
Add result
Shift left 2
ALUSrc
Branch
RegWrite
MemWrite
MemtoReg
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Zero
ALU
ALU result
Address
Data memory
Read data
Write data
Mux
Instruction [15–0]
16
Sign-extend
32
6
ALU control
MemRead
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst