



ECE/CS 472/572

Computer Architecture:

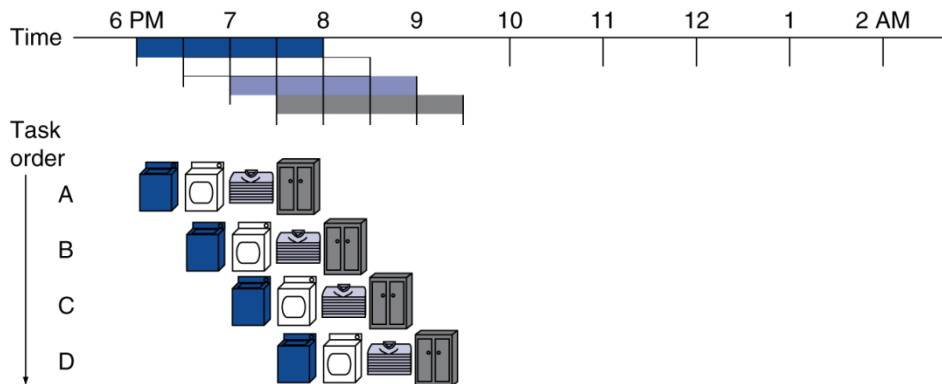
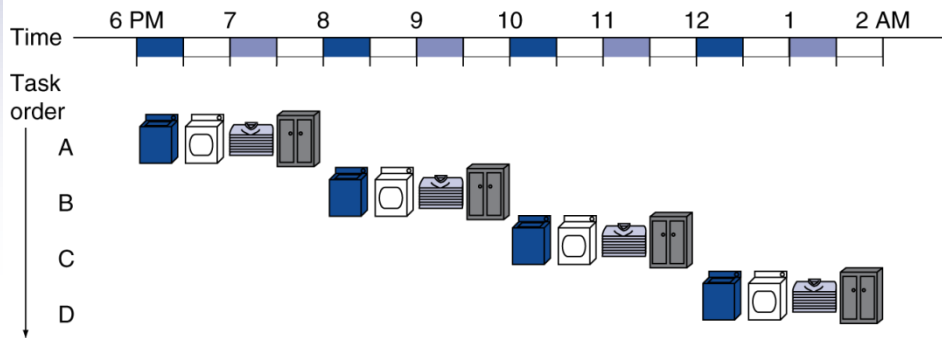
Pipeline Essentials

Prof. Lizhong Chen

Spring 2019

Pipelining Analogy

- Pipelined laundry: overlapping execution
 - Parallelism improves performance
 - Improve **throughput** rather than latency



- Four loads:
 - Speedup
 $= 8 / 3.5 = 2.3$
- n loads (non-stop):
 - Speedup
 $= 2n / (0.5n + 1.5) \approx 4$
 $= \text{number of stages}$

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

MIPS Pipelined Datapath

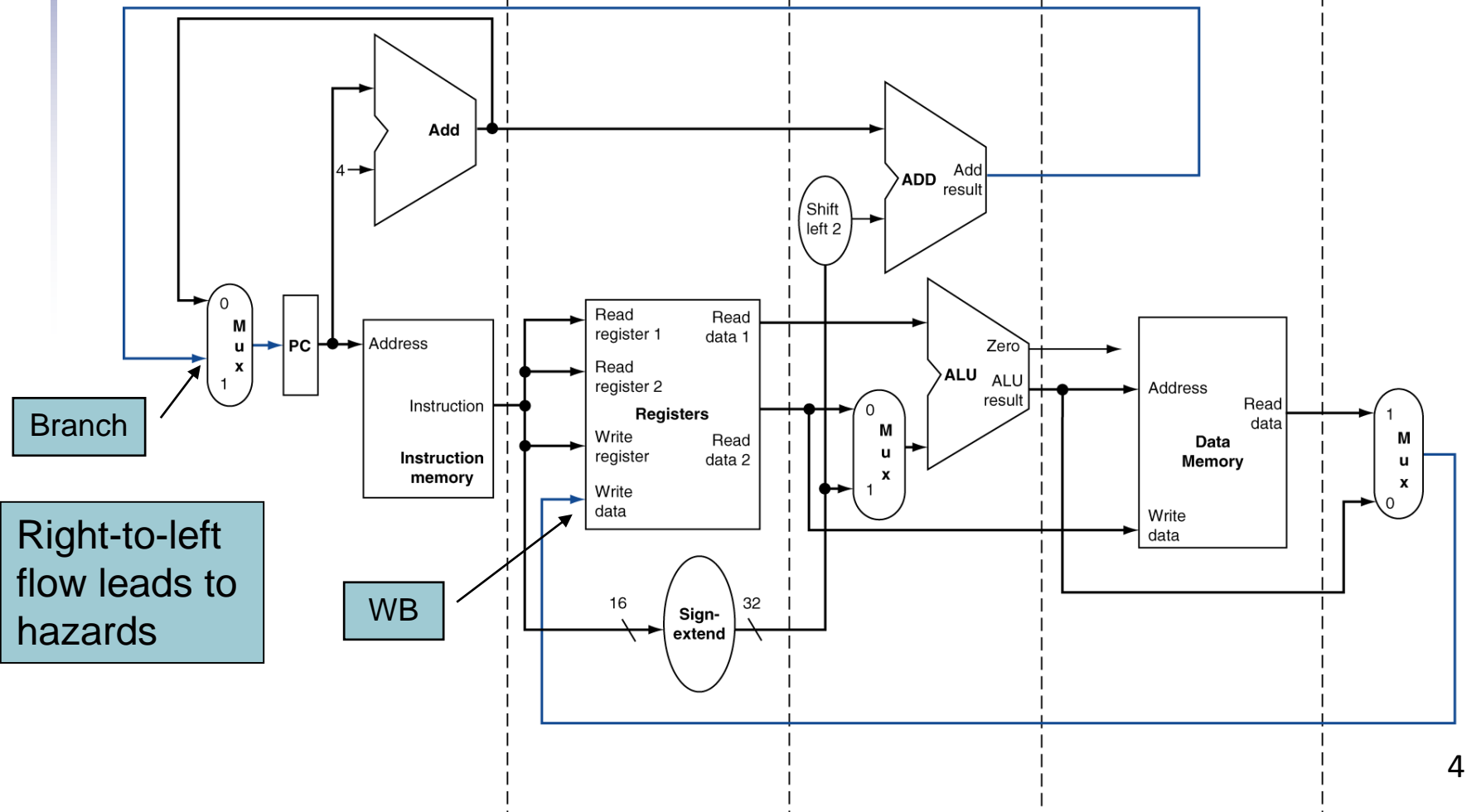
IF: Instruction fetch

ID: Instruction decode/
register file read

EX: Execute/
address calculation

MEM: Memory access

WB: Write back

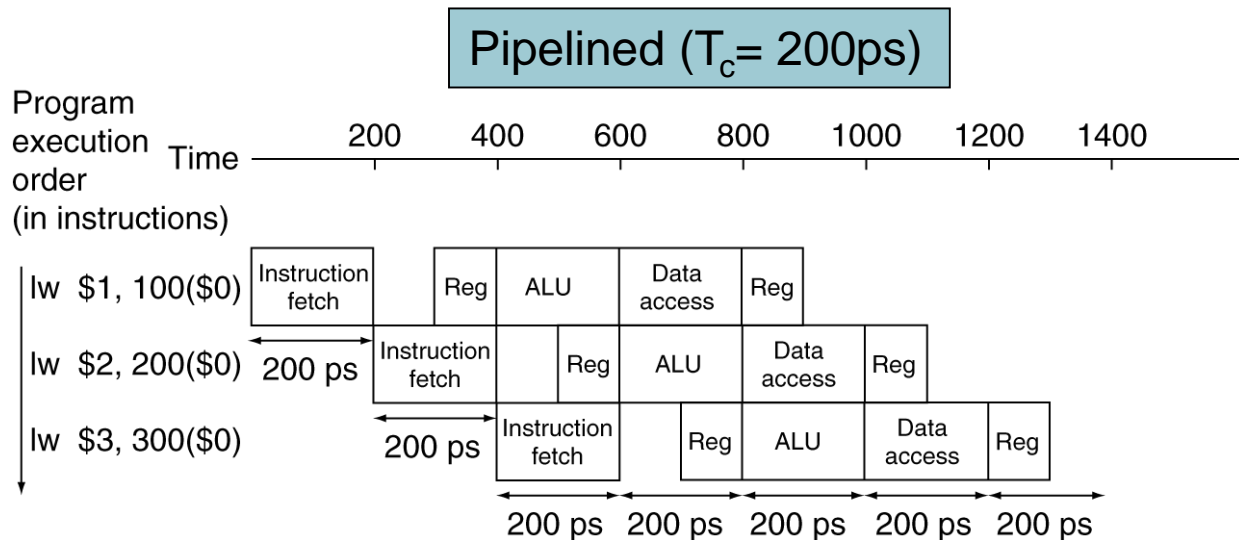
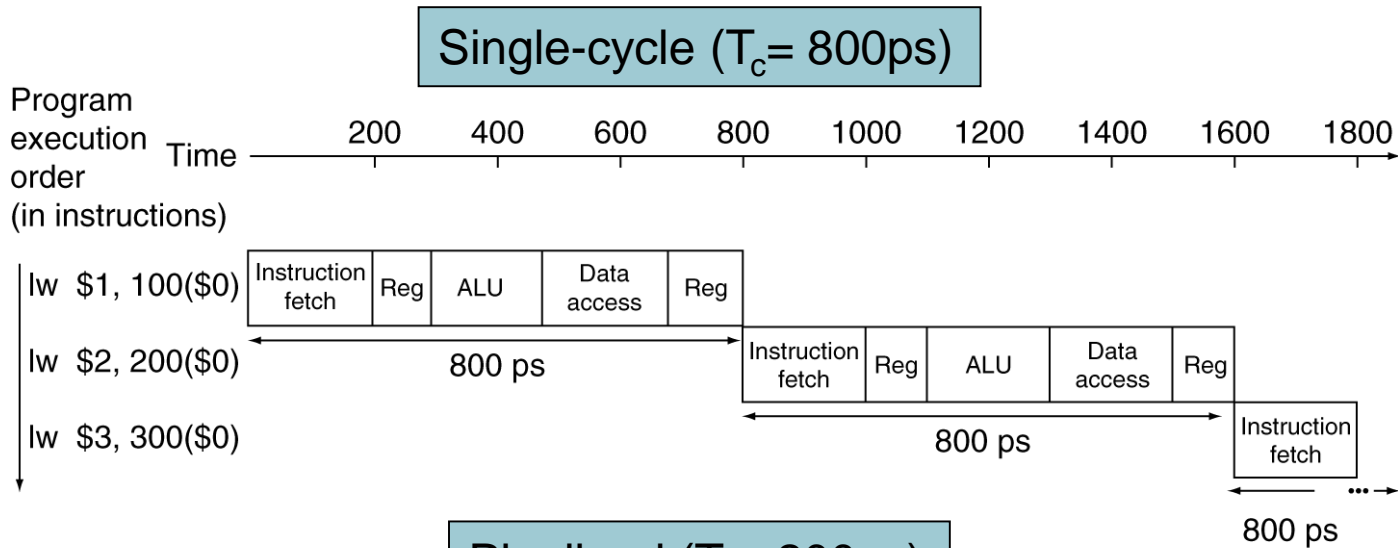


Pipeline Performance

- 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



Pipeline Speedup

- If all stages are balanced
 - Speedup is the 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
 - x86: 1- to 15-byte instructions
 - 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 stall the execution of 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

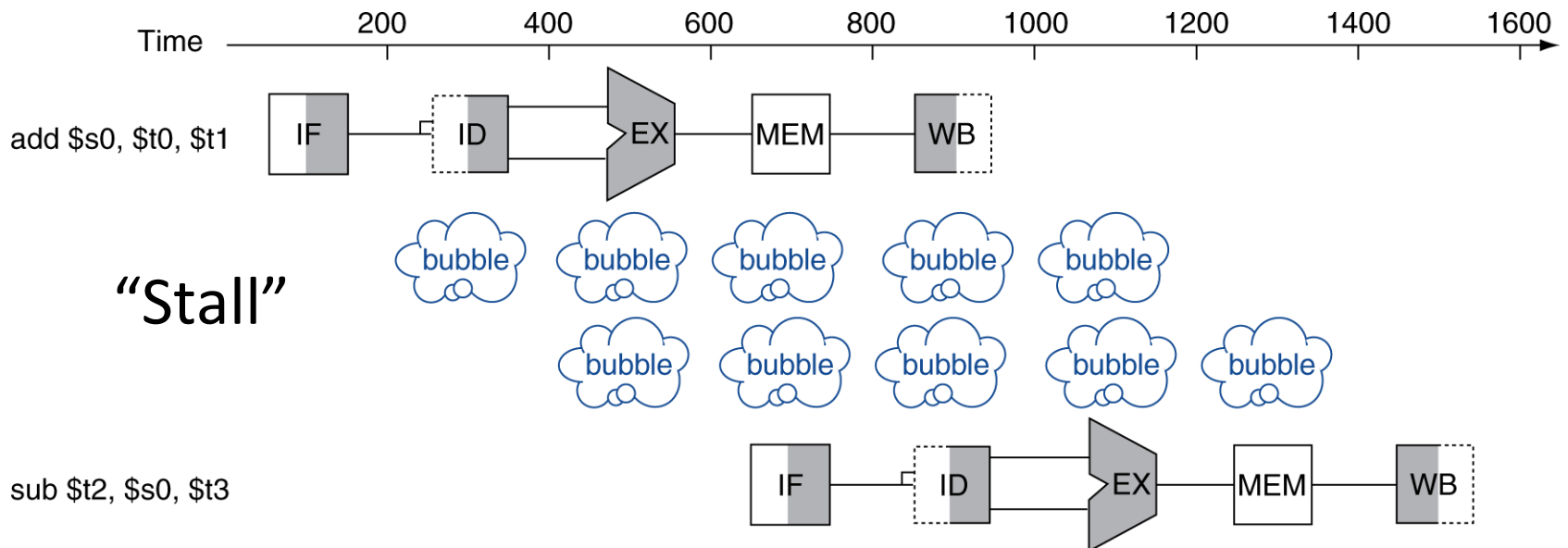
Structure Hazards

- 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” (no-op)
- Hence, pipelined datapaths require separate instruction/data memories
 - Or separate instruction/data caches

Data Hazards

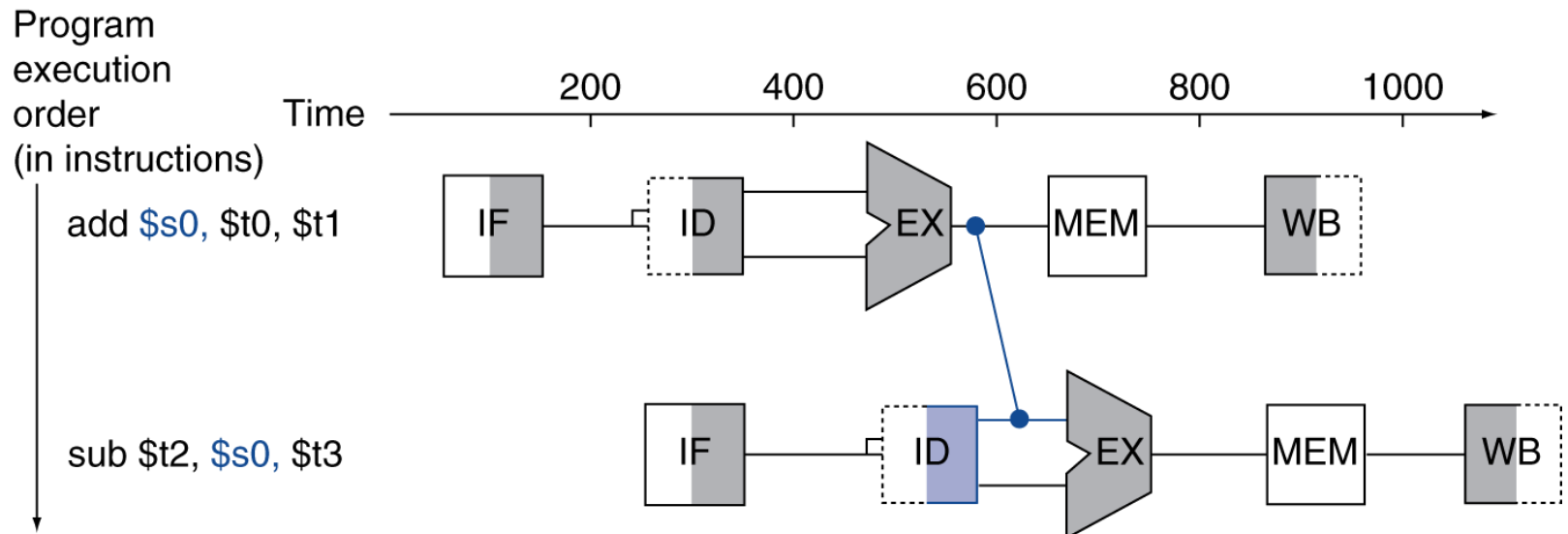
- An instruction depends on completion of data access by a previous instruction

- add **\$s0**, \$t0, \$t1
sub \$t2, **\$s0**, \$t3



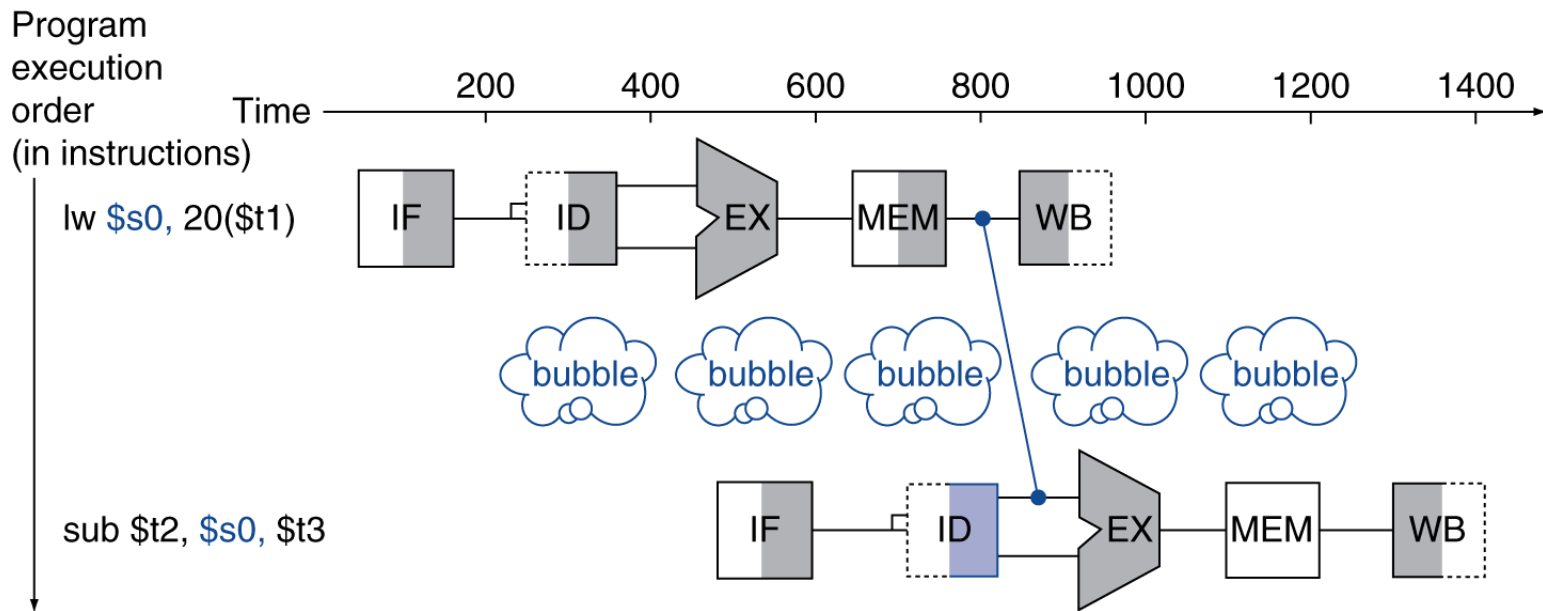
Forwarding (aka Bypassing)

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



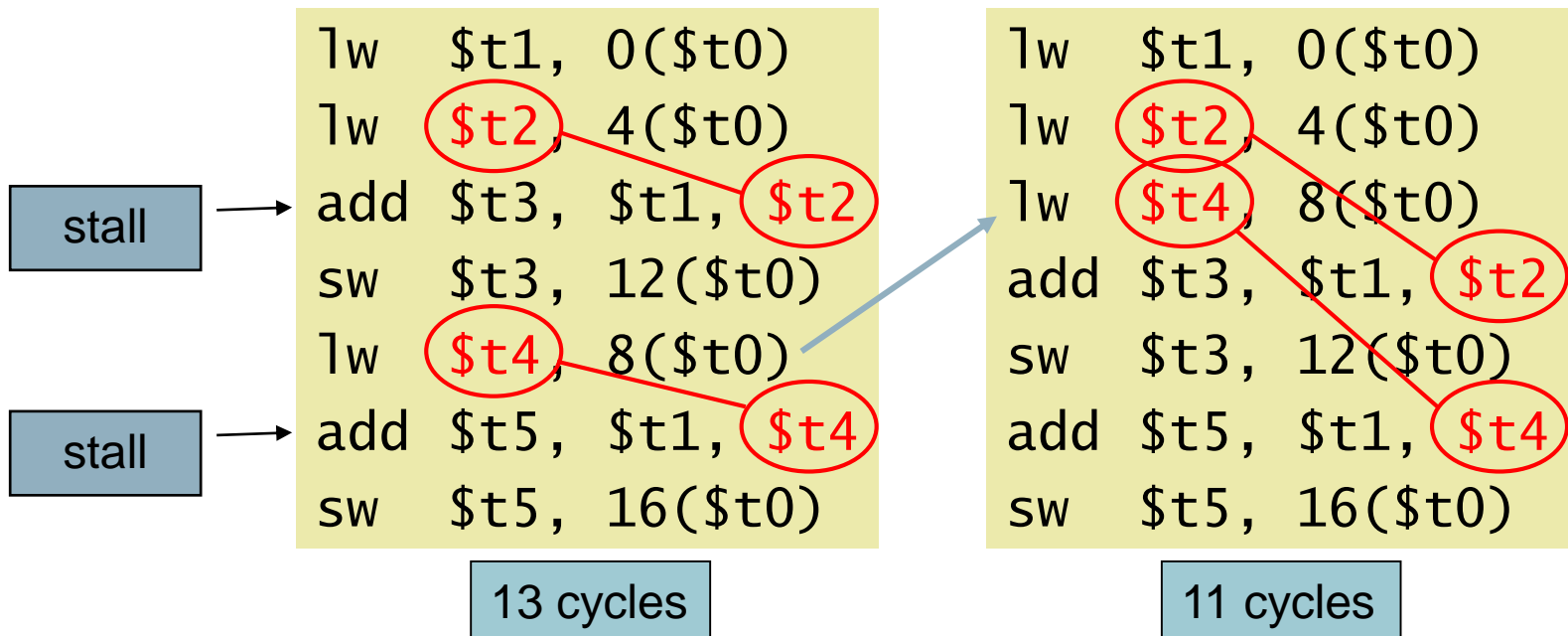
Load-Use Data Hazard

- 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 $A = B + E$; $C = B + F$;

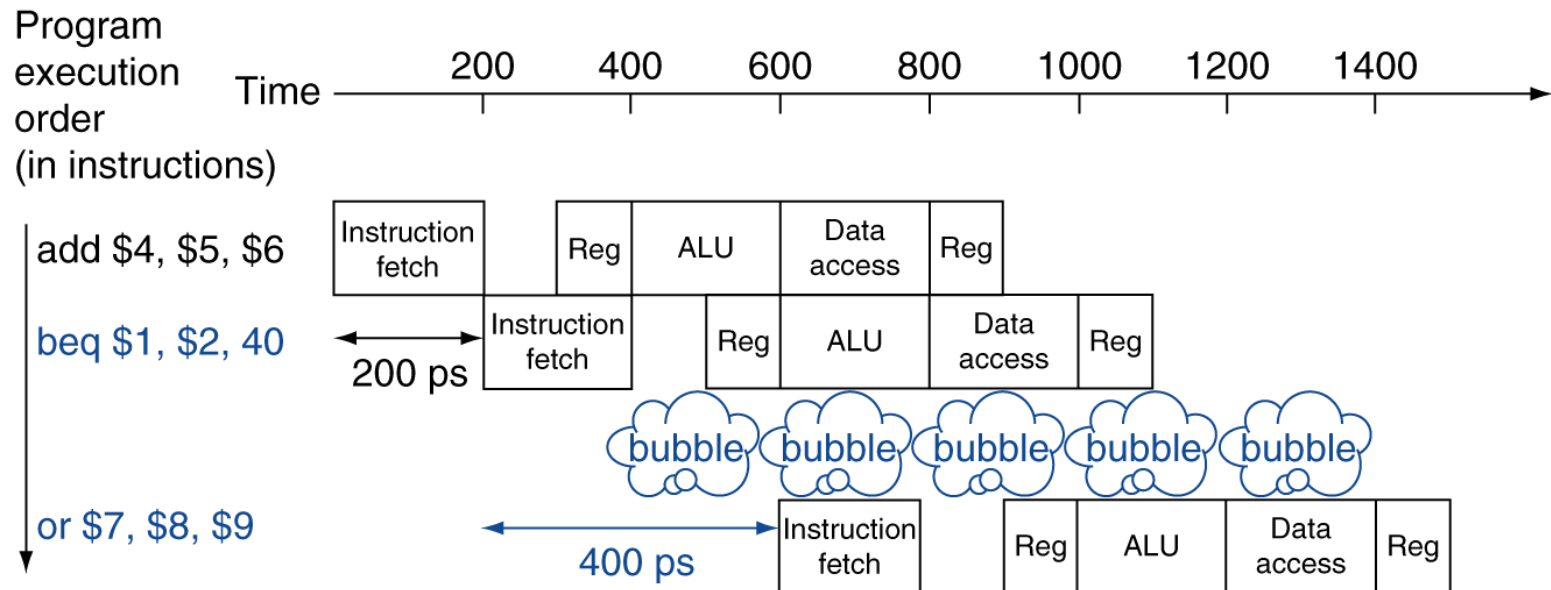


Control Hazards

- 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
 - Needs two bubbles
- In MIPS pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to do it in ID stage
 - Still needs one bubble

Stall on Branch

- Resolving Branch in ID
 - Wait until branch outcome determined before fetching next instruction

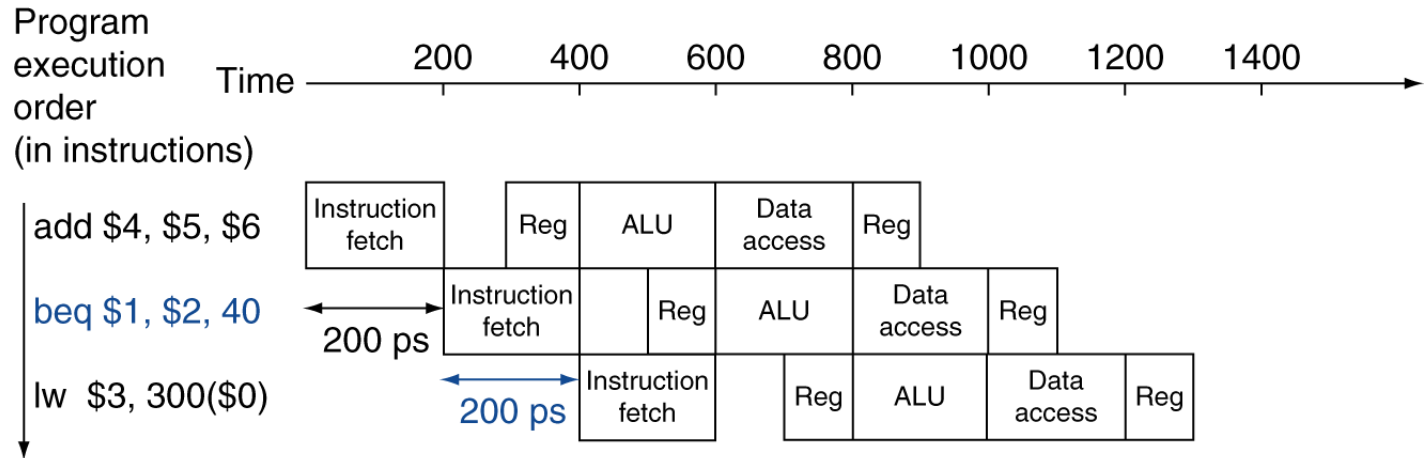


Branch Prediction

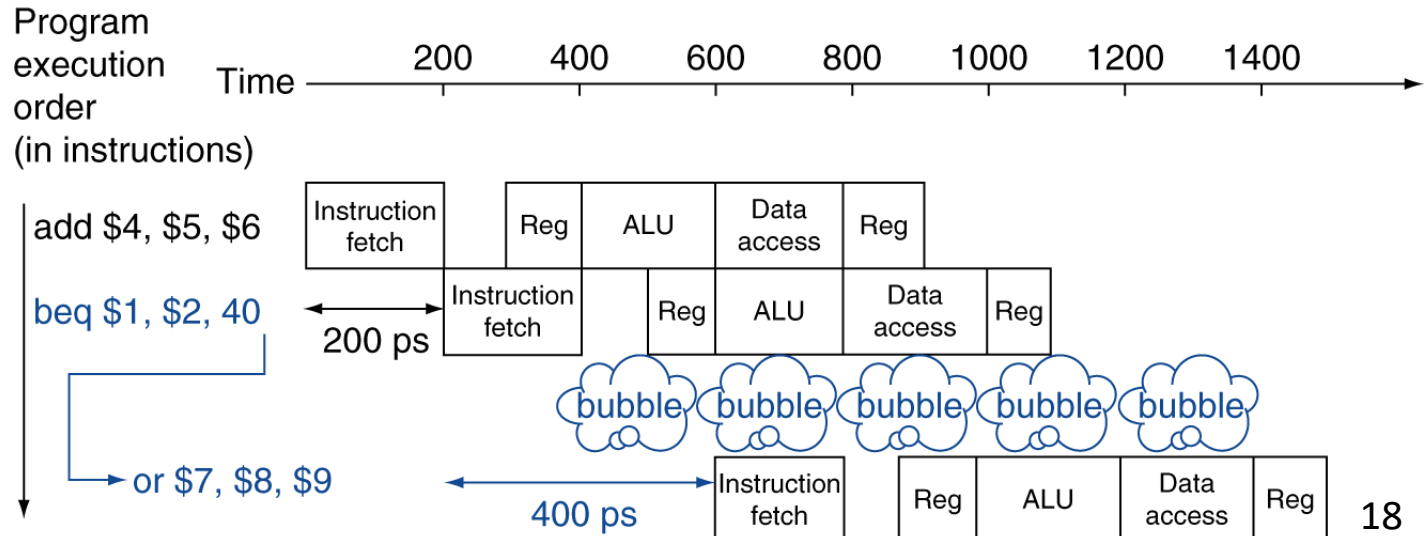
- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken (built-in)
 - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history

Pipeline Basics Summary

- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

MIPS Pipelined Datapath

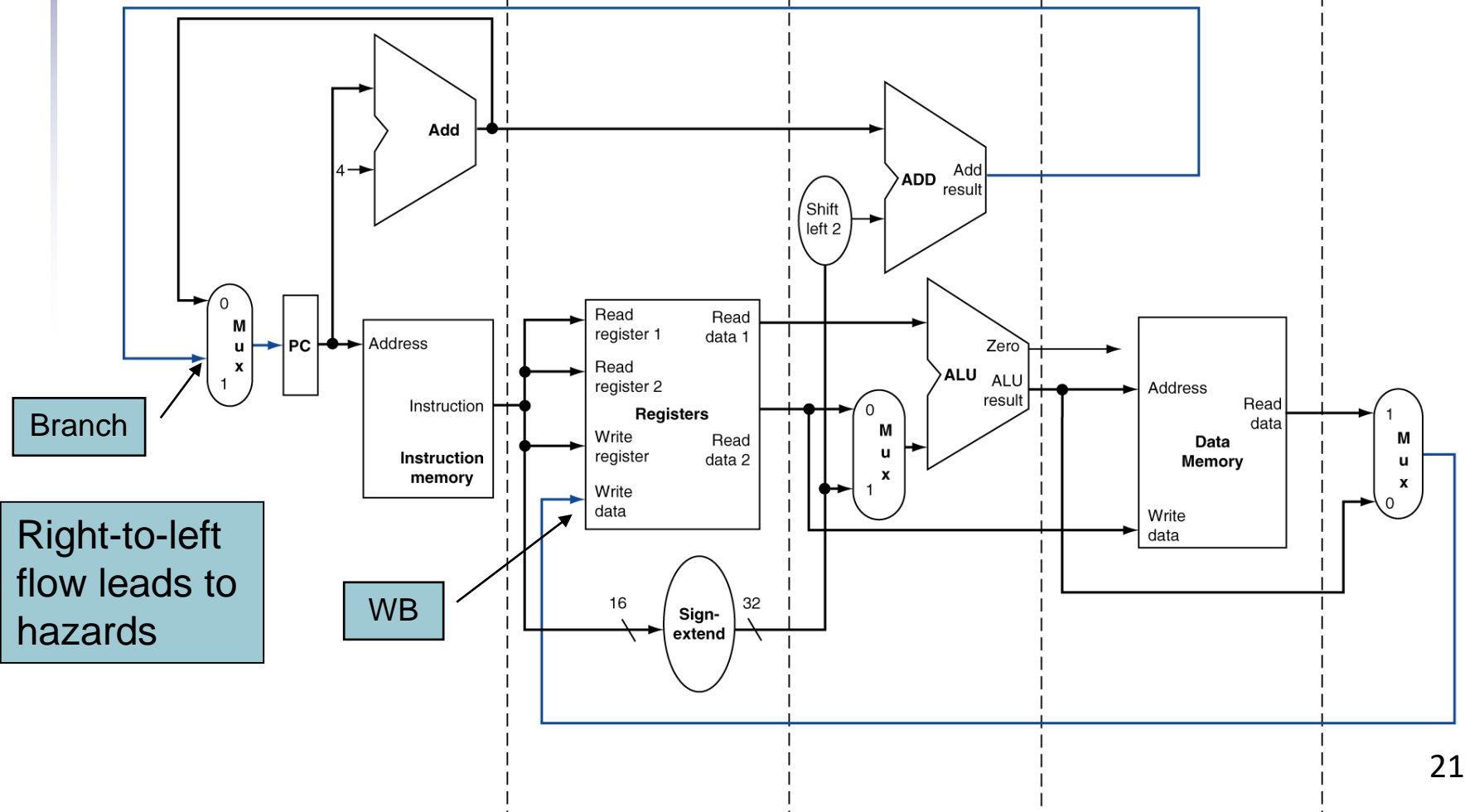
IF: Instruction fetch

ID: Instruction decode/
register file read

EX: Execute/
address calculation

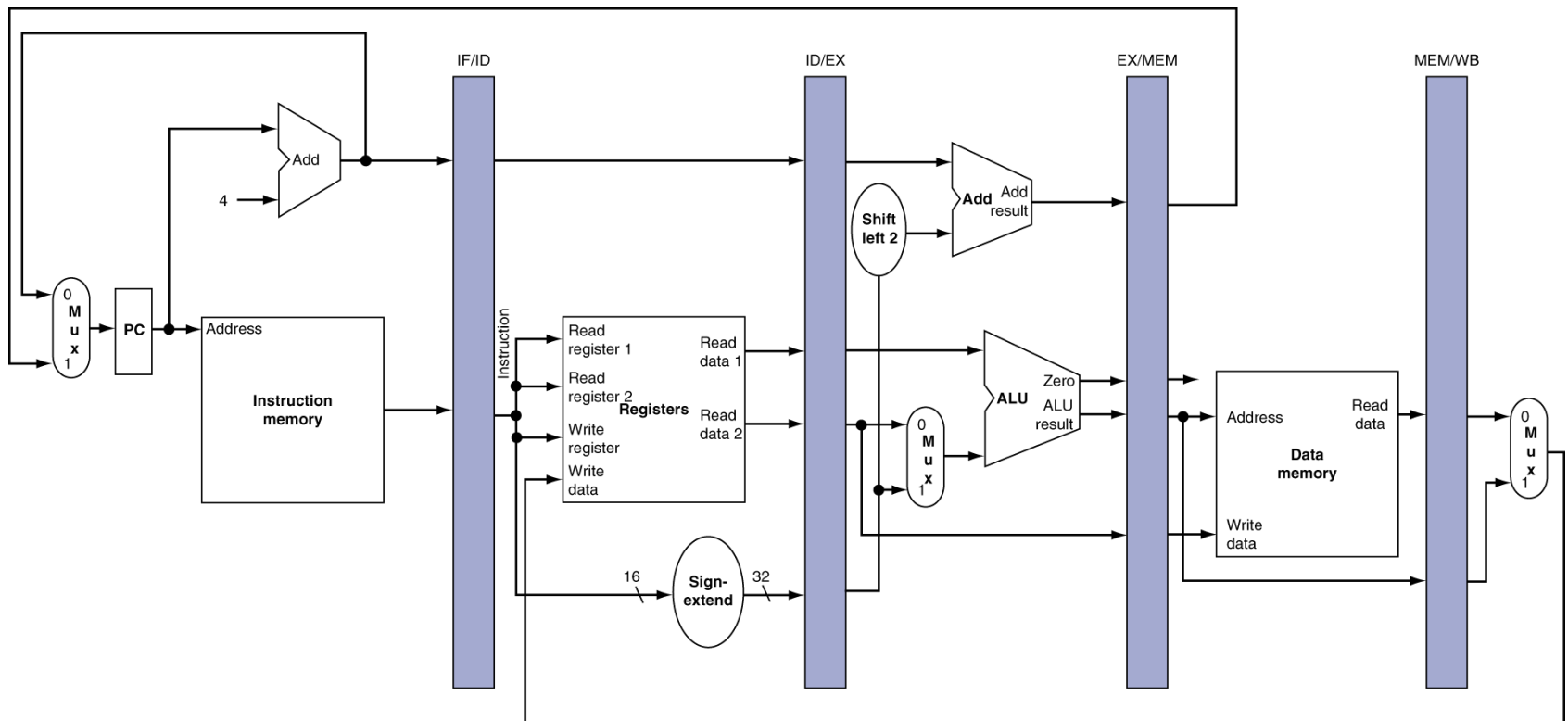
MEM: Memory access

WB: Write back



Pipeline registers

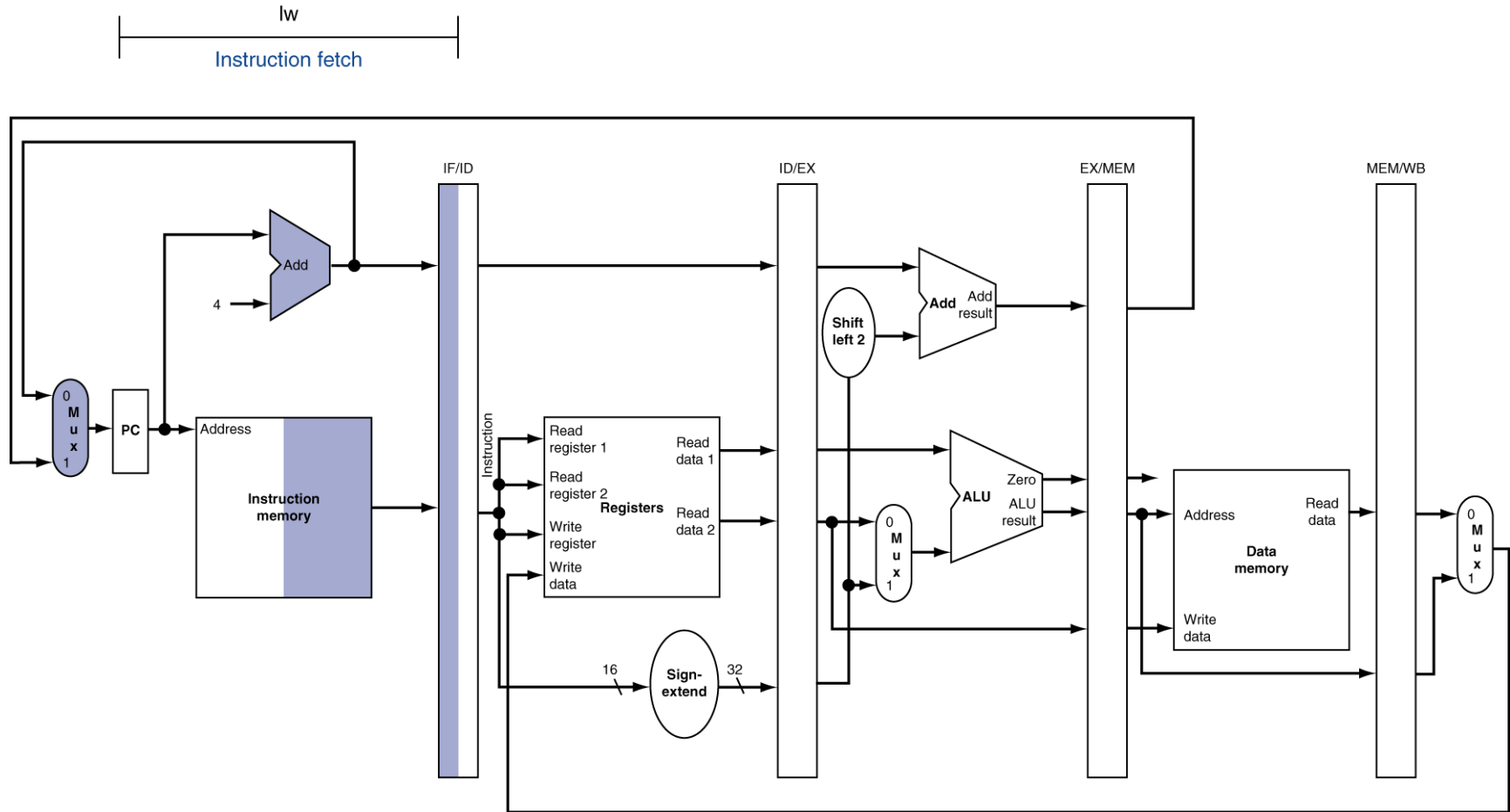
- Need registers between stages
 - To hold information produced in previous cycle



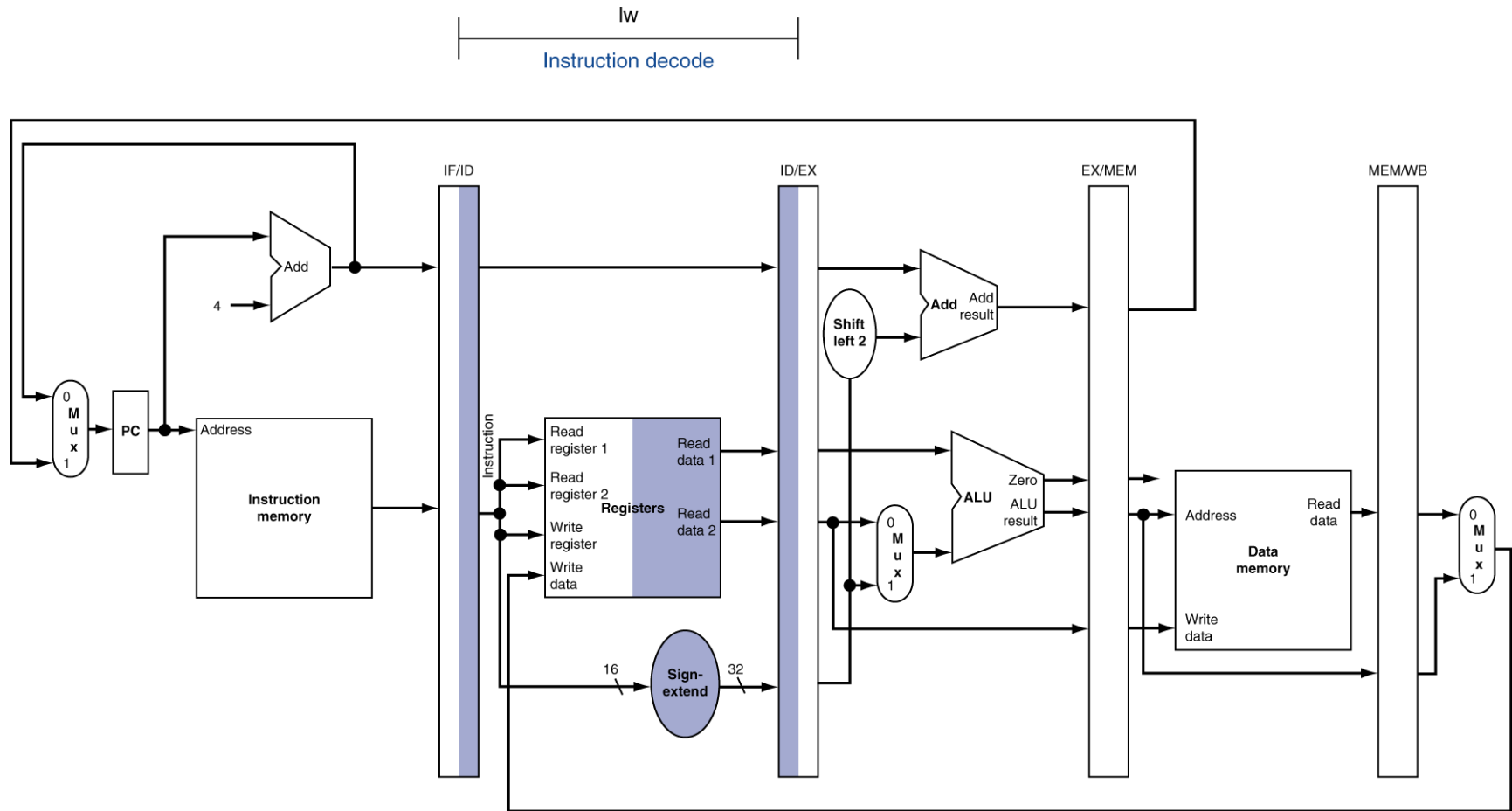
Pipeline Operation

- 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
 - “Multi-clock-cycle” diagram
 - Graph of operation over time
- We’ll first look at “single-clock-cycle” diagrams for load & store

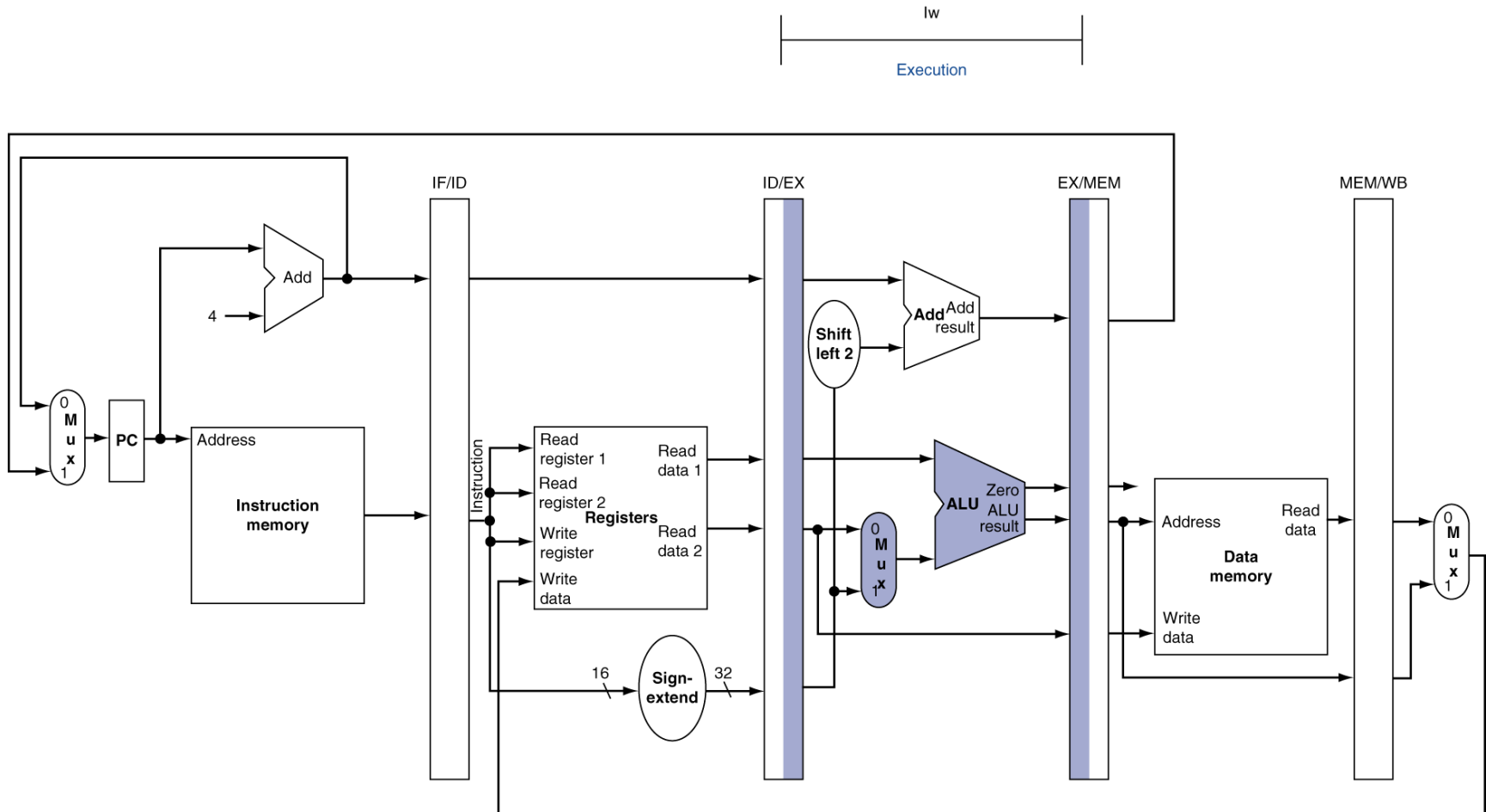
IF for Load, Store, ...



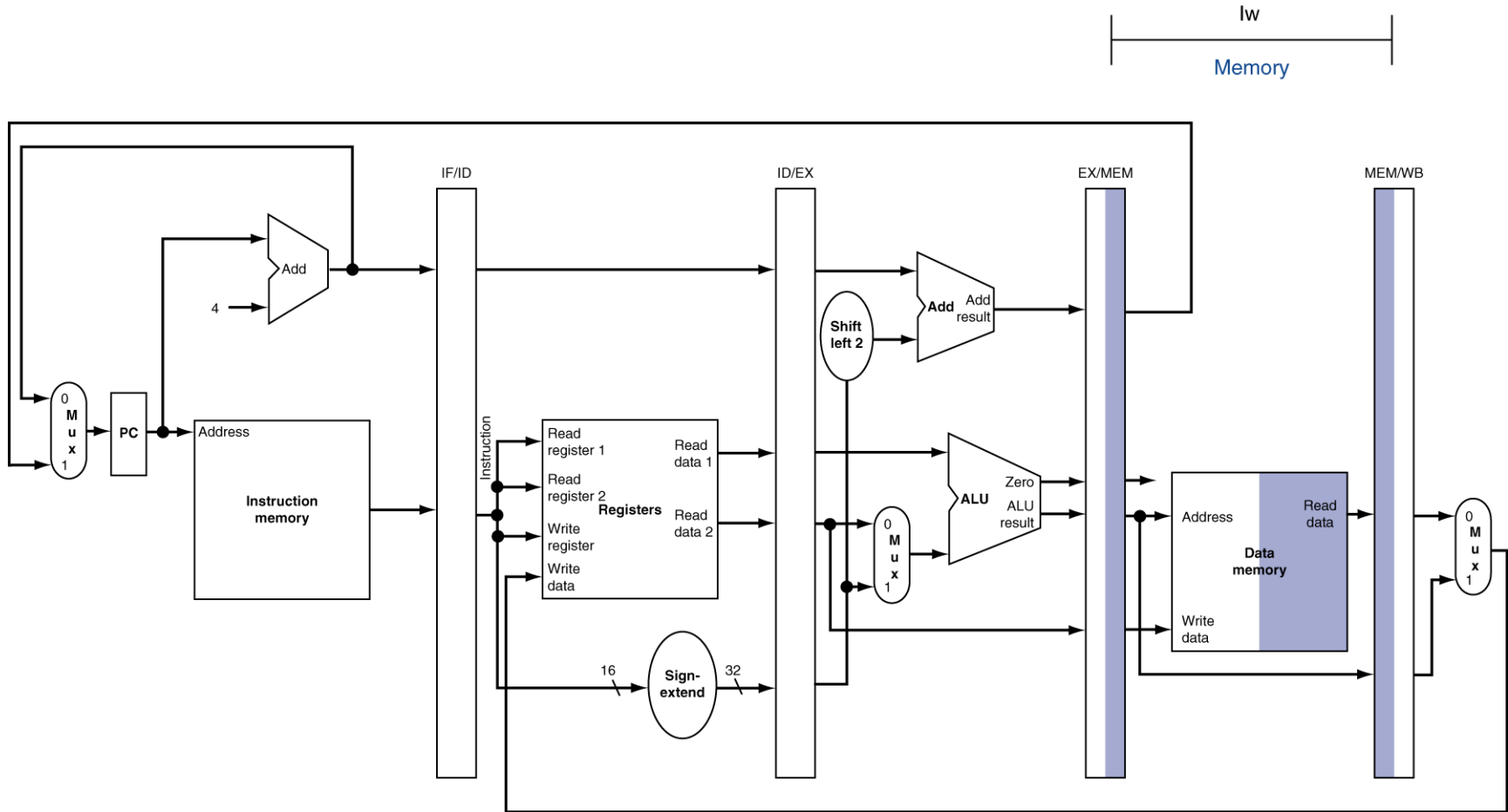
ID for Load, Store, ...



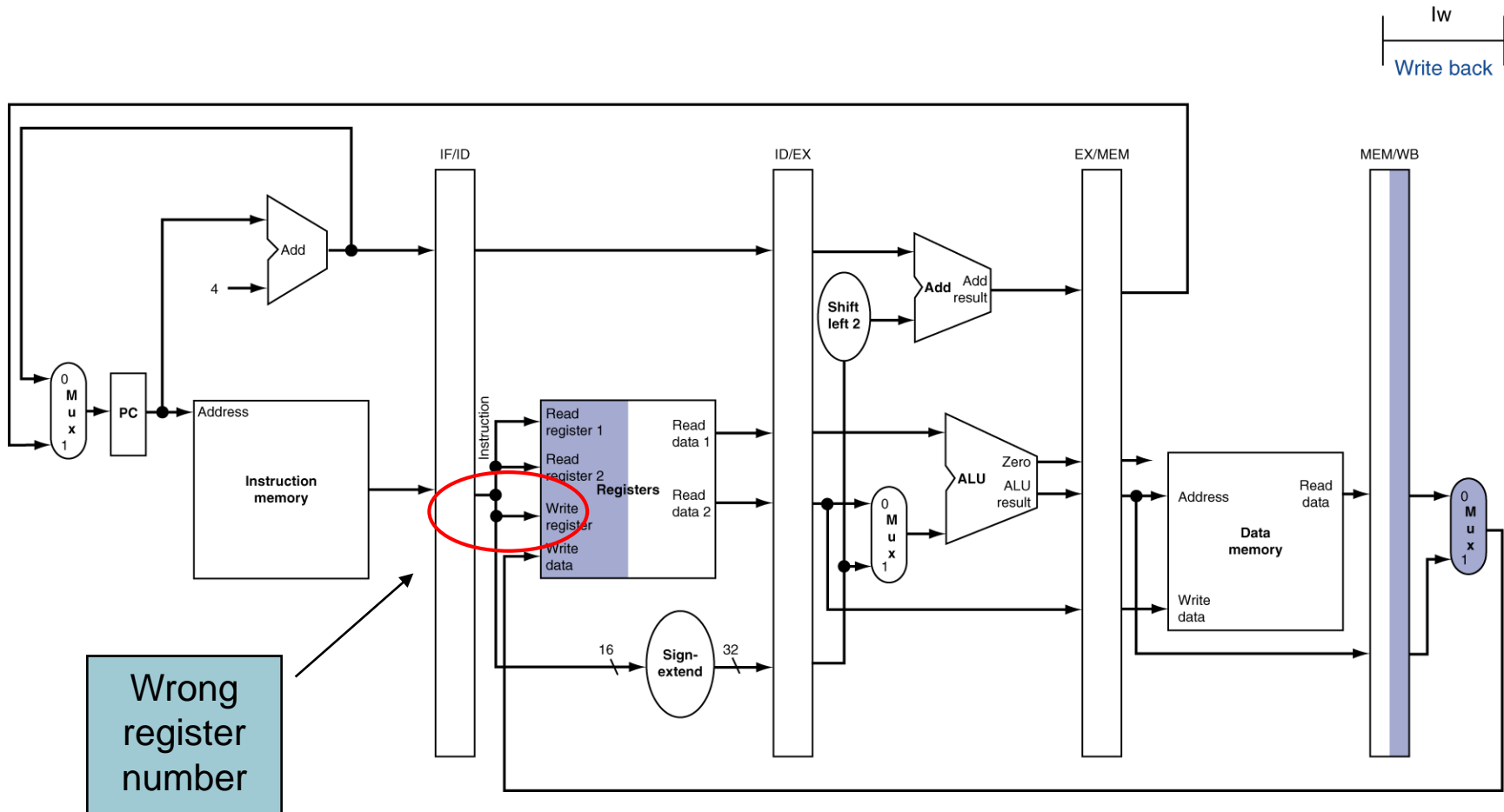
EX for Load



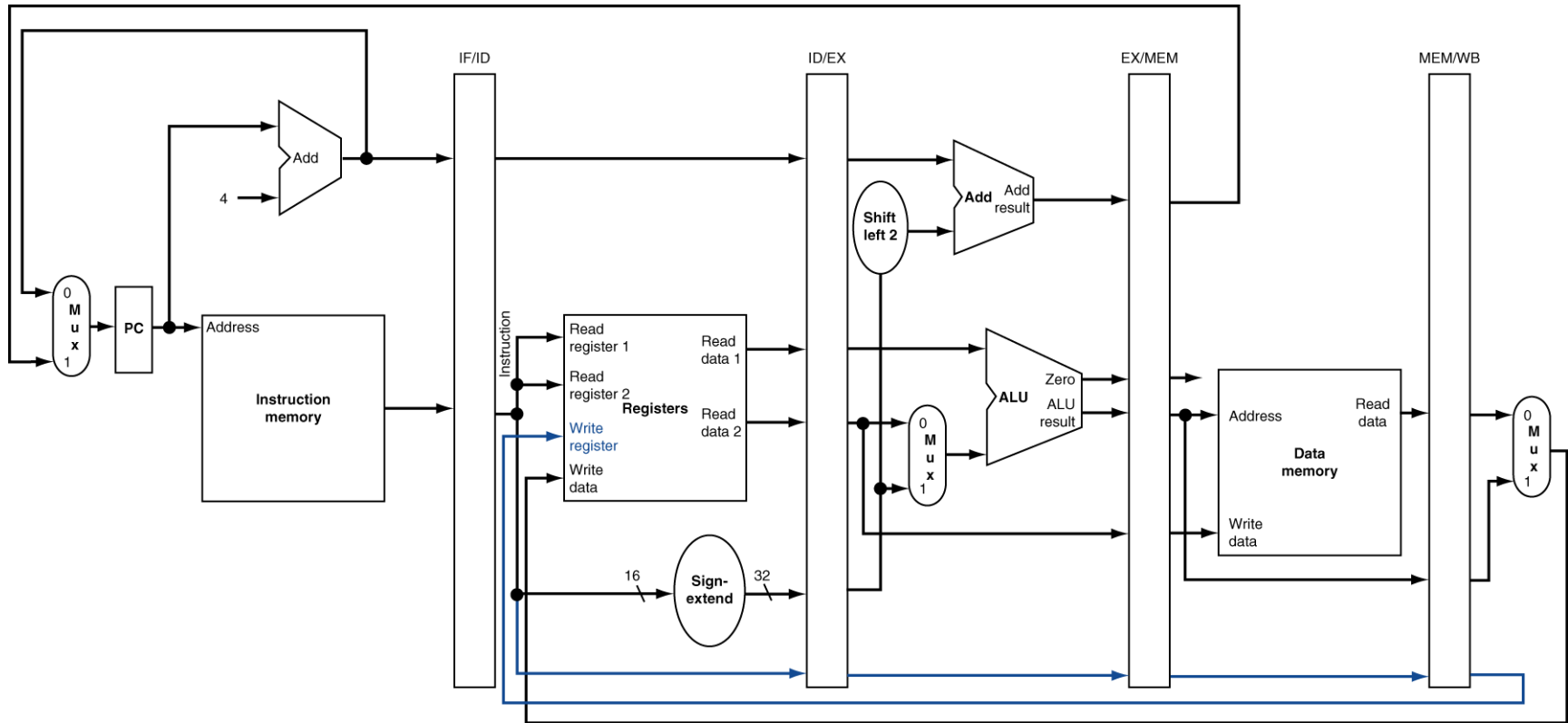
MEM for Load



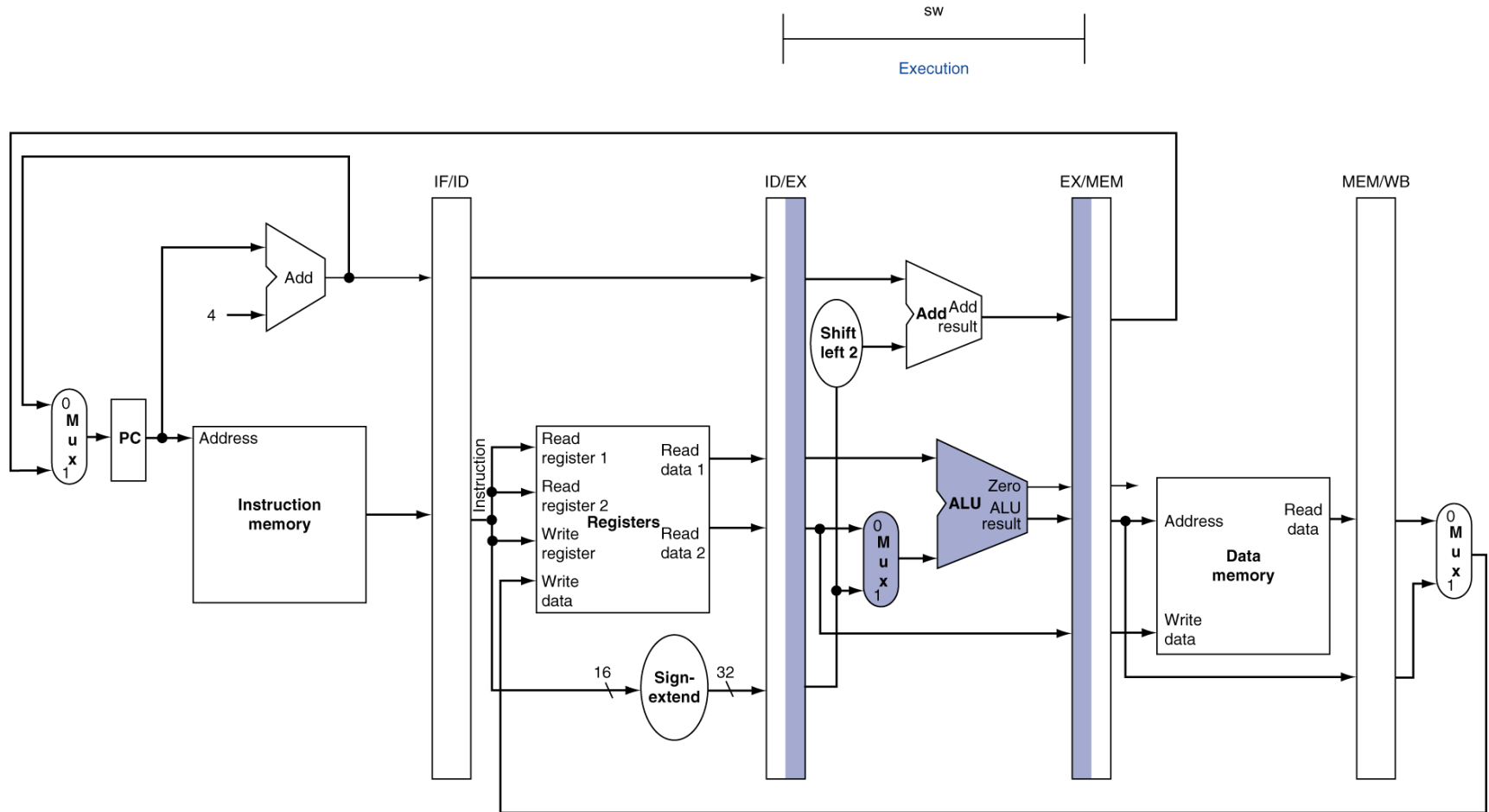
WB for Load



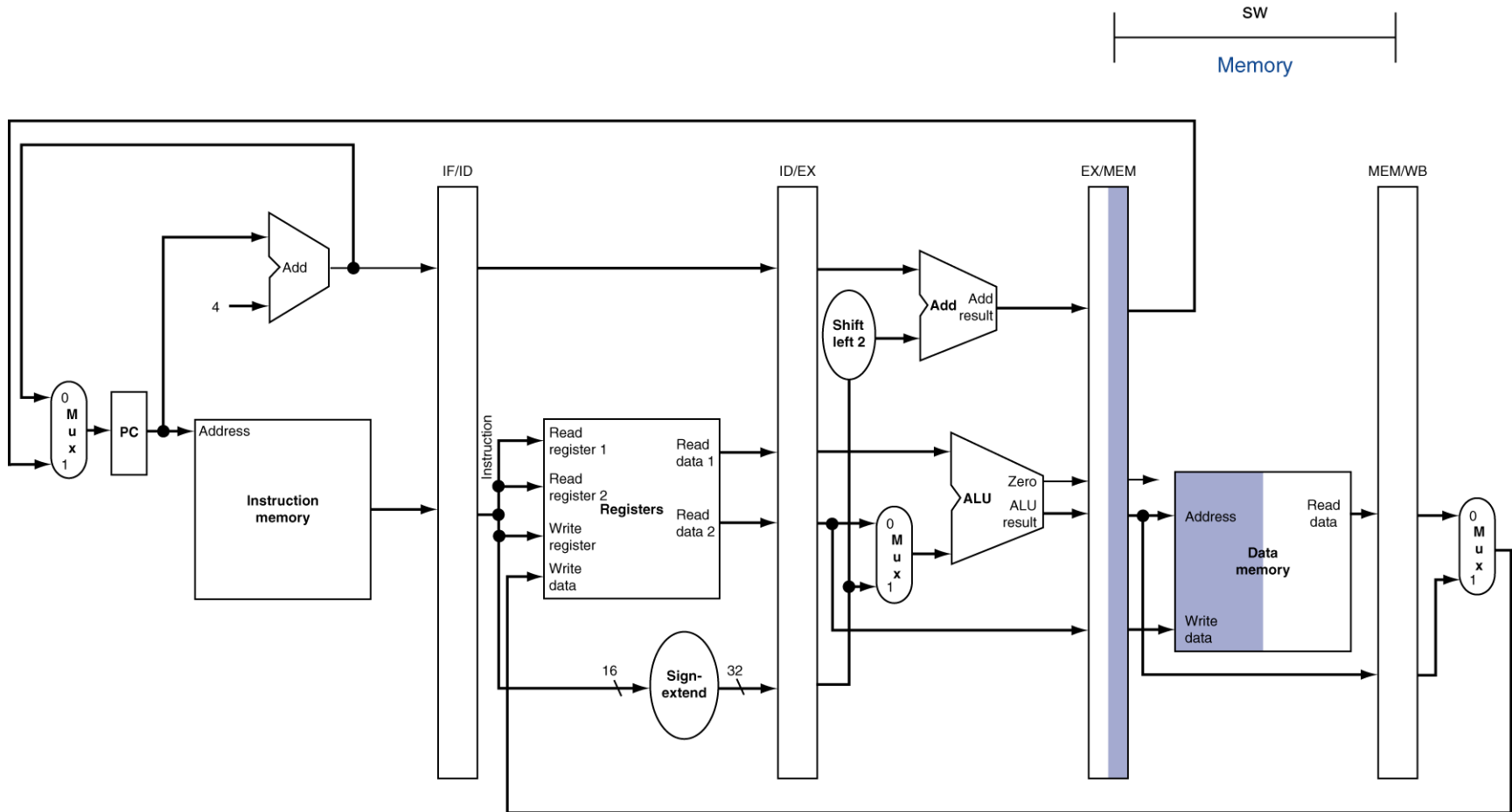
Corrected Datapath for Load



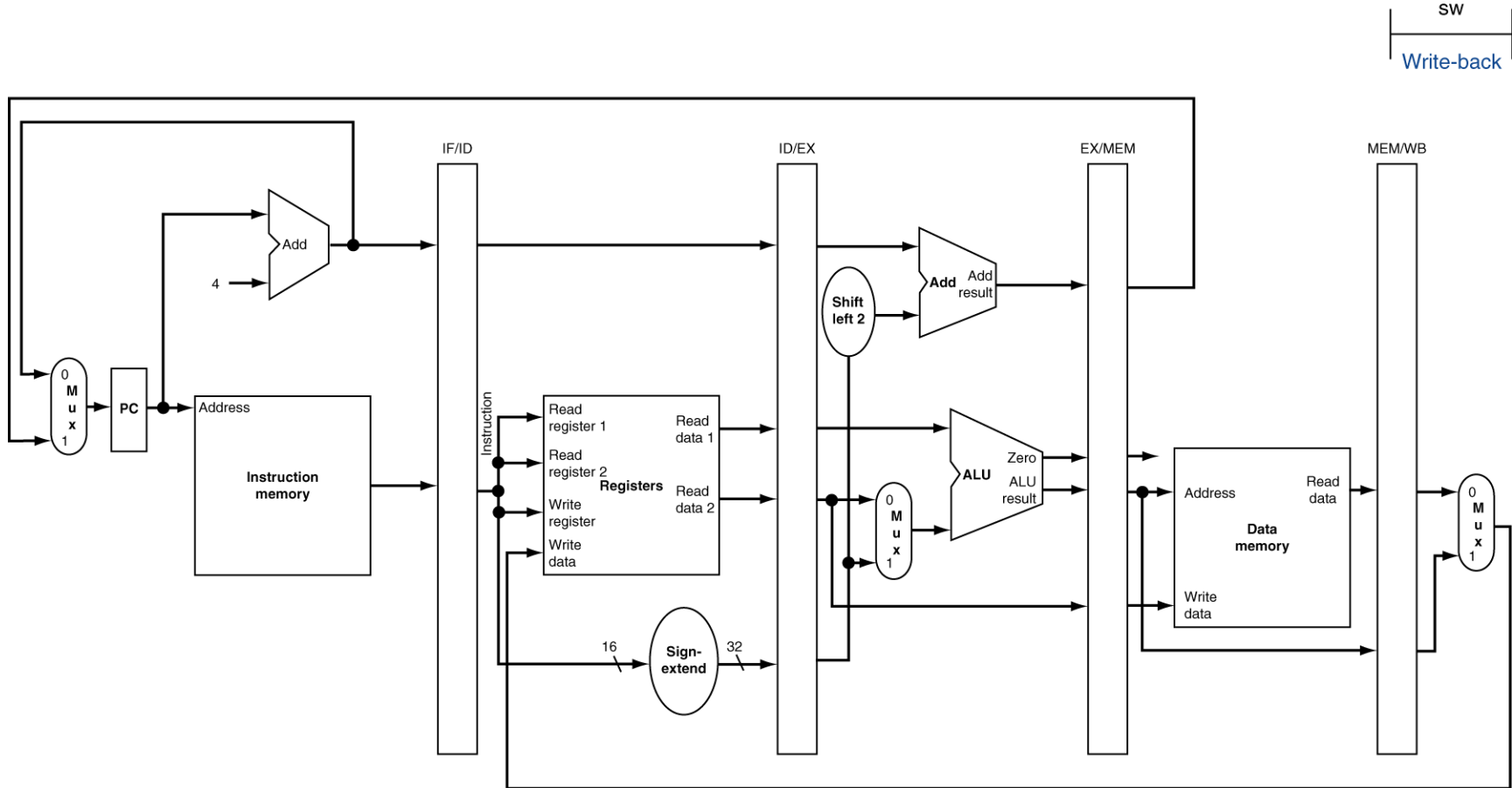
EX for Store



MEM for Store

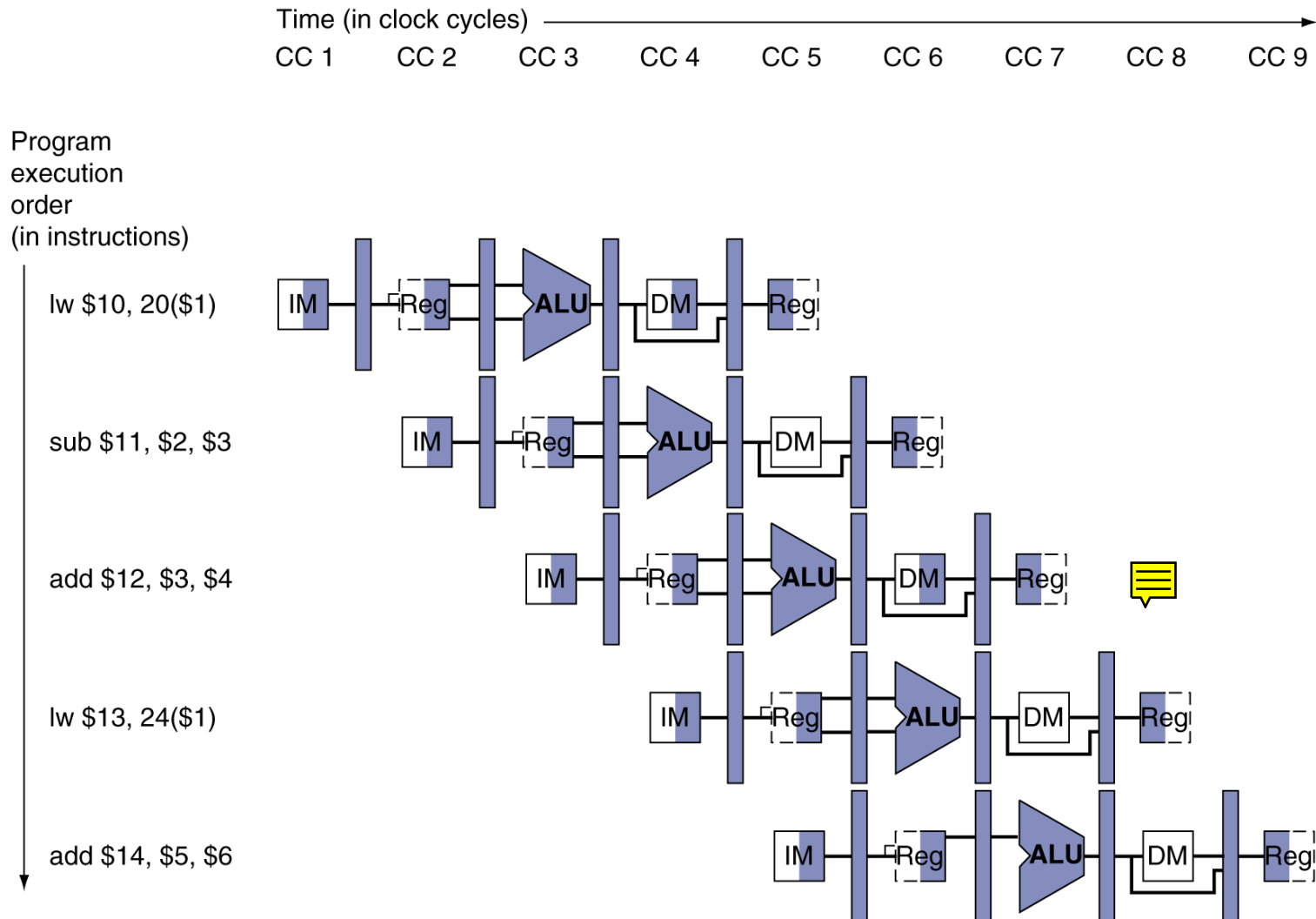


WB for Store



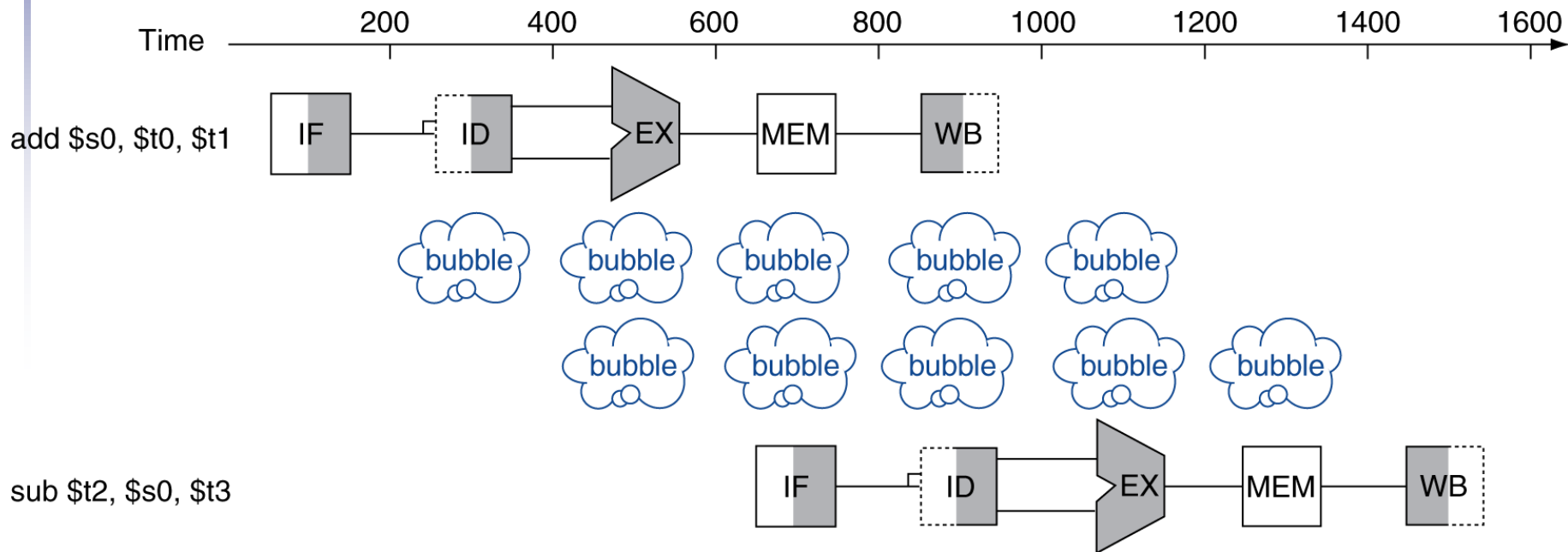
Multi-Clock-Cycle Diagram

■ Form showing resource usage



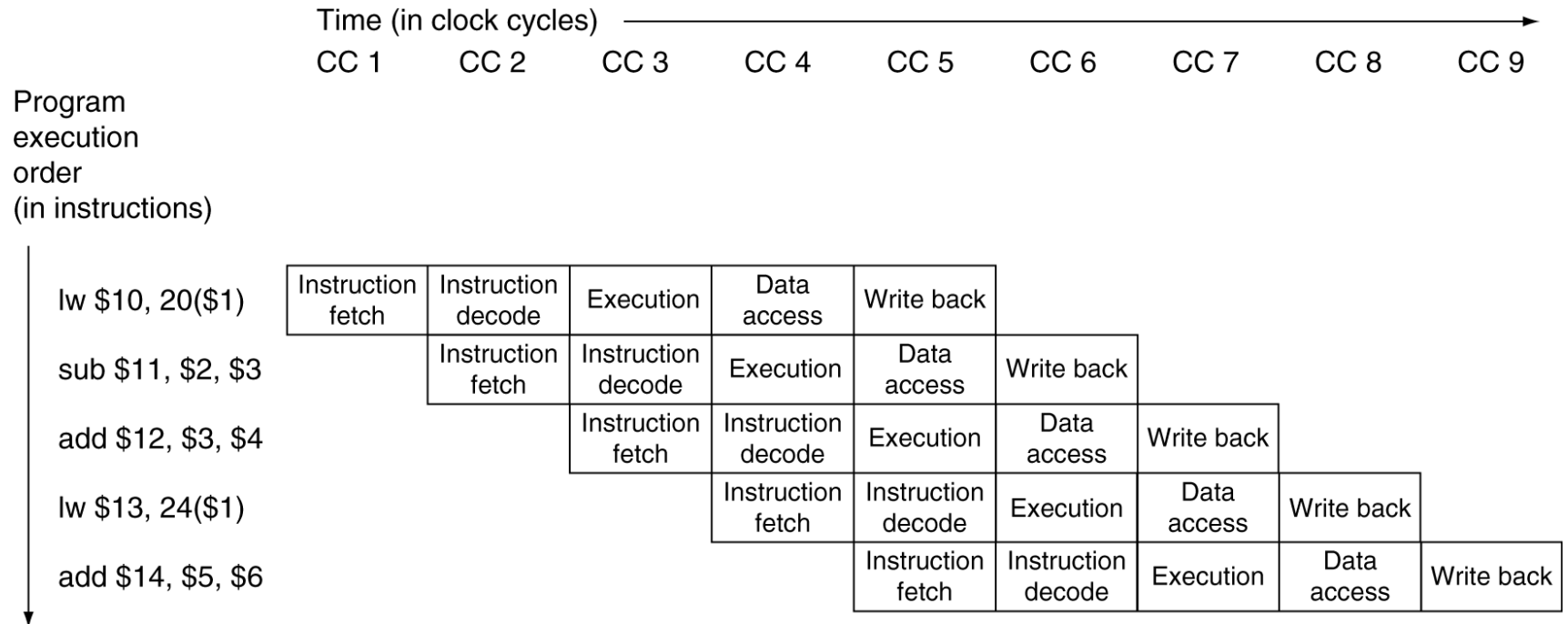
Multi-Clock-Cycle Diagram

■ Form showing stage names



Multi-Clock-Cycle Diagram

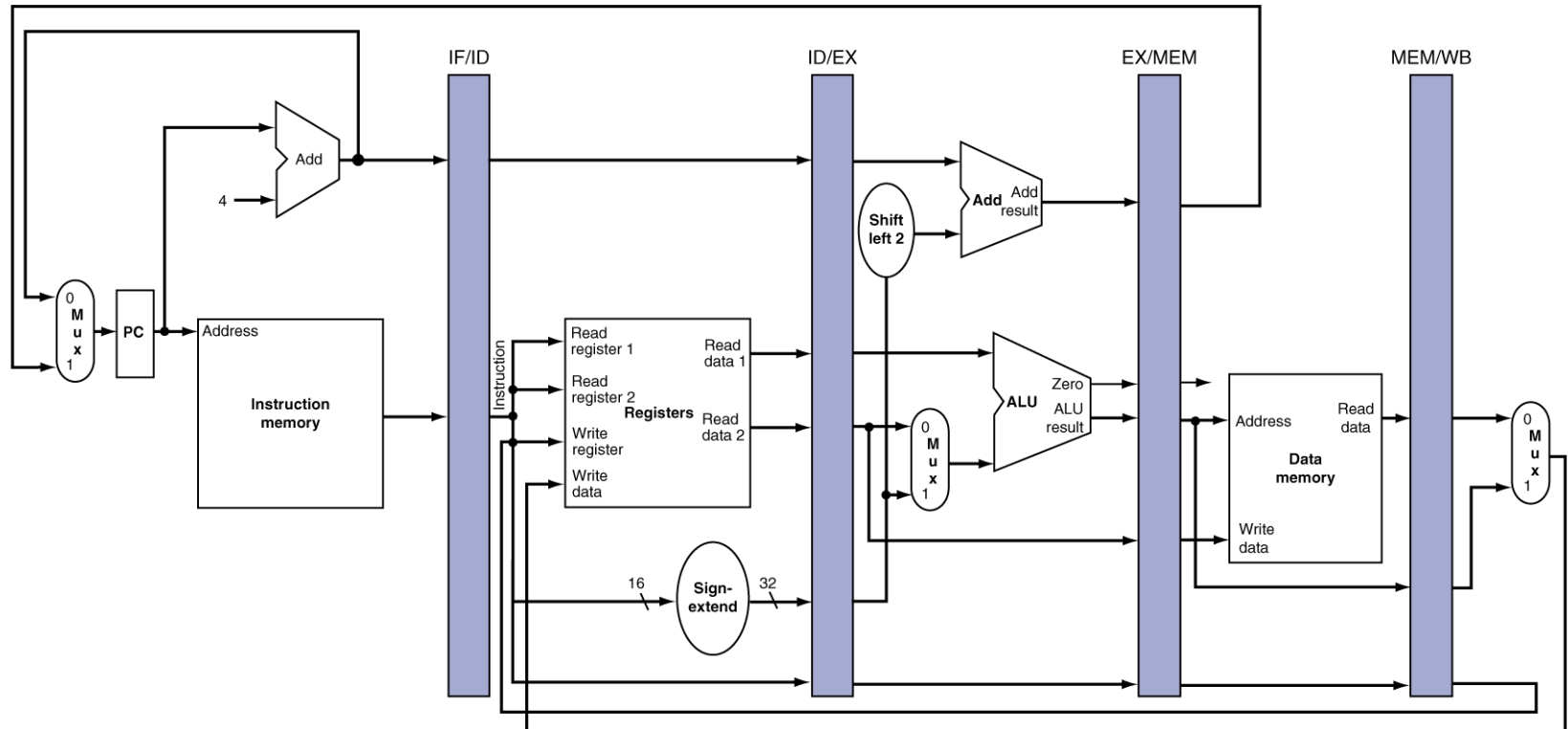
■ Traditional form



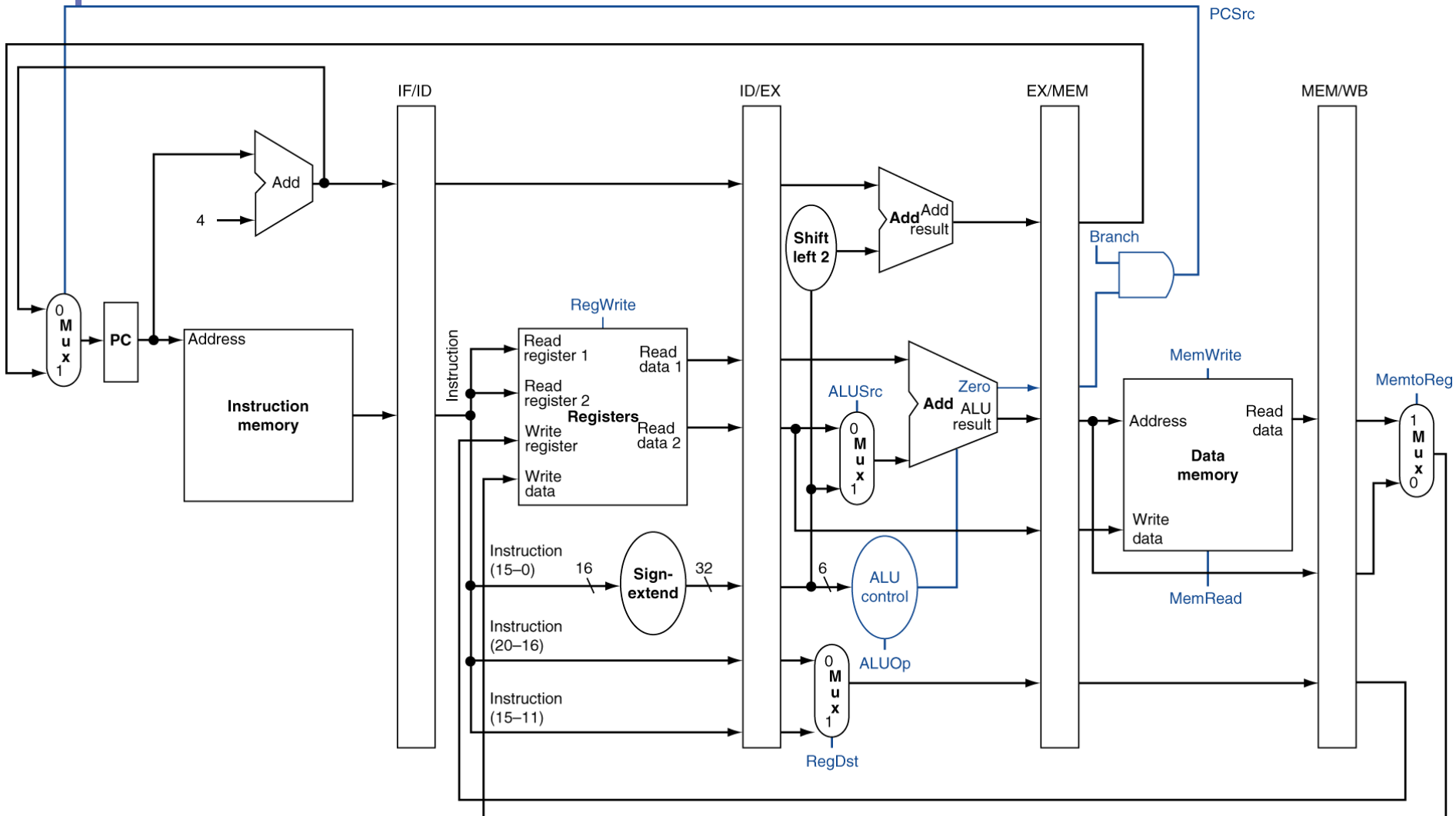
Single-Clock-Cycle Diagram

■ State of pipeline in a given cycle (CC5)

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 Signals



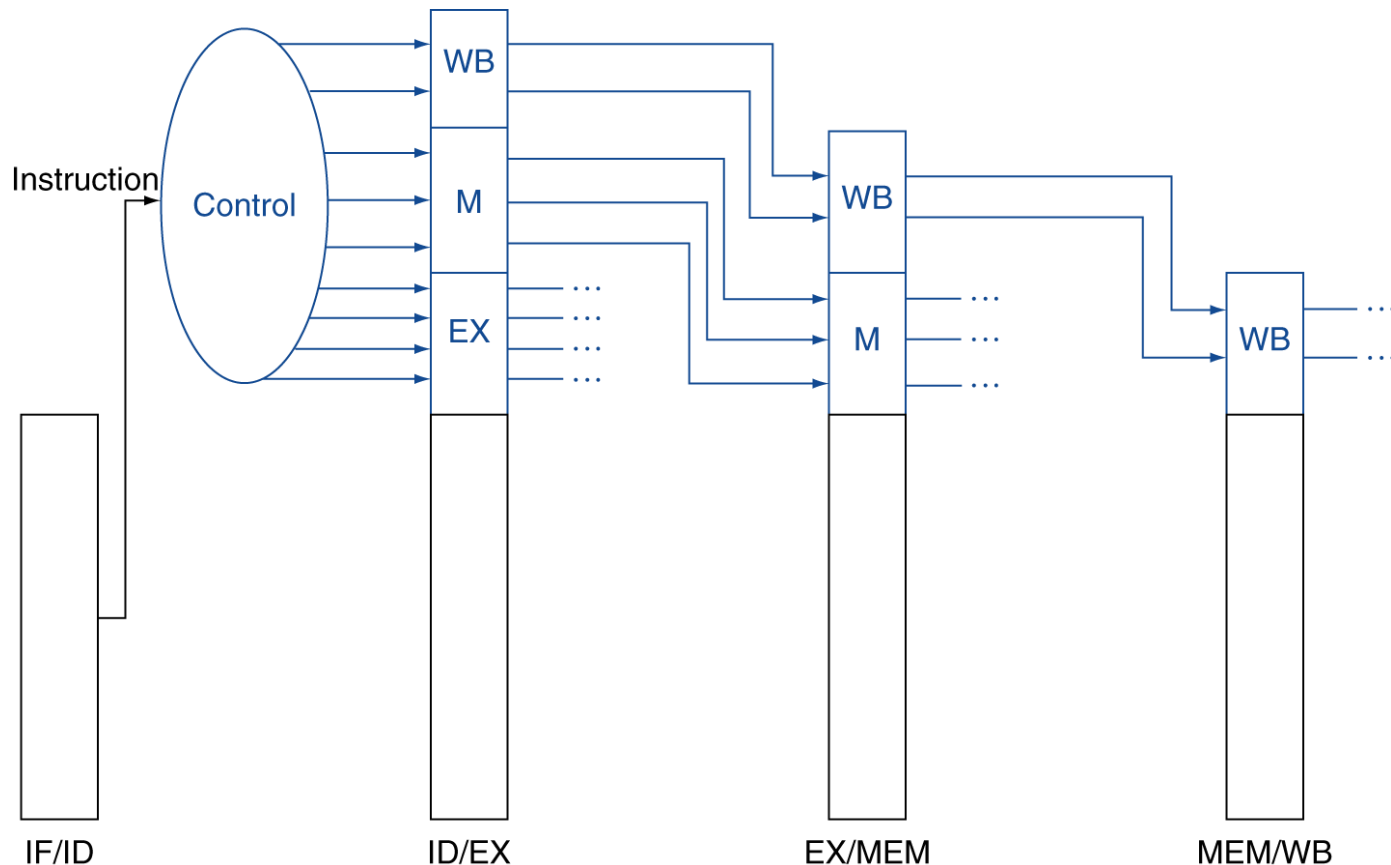
Pipelined Control Signals

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

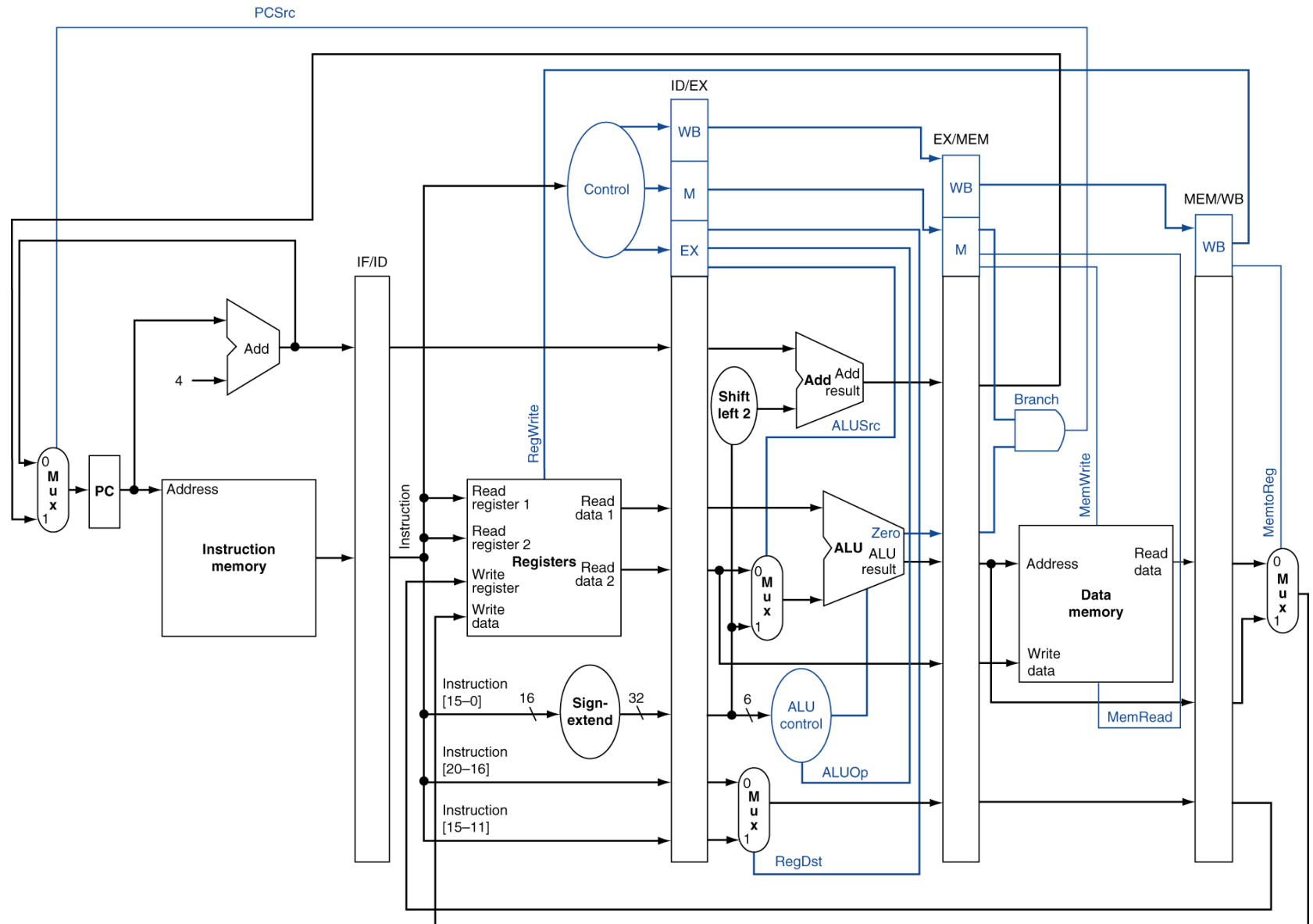
Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Pipelined Control

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



Pipelined Control



RAW Dependence

- Read-After-Write dependence
 - add **\$s0**, \$t0, \$t1
 - sub \$t2, **\$s0**, \$t3
- If reversed
 - Read the old value!

WAR Dependence

- Write-After-Read dependence
 - `sub $t2, $s0, $t3`
 - `add $s0, $t0, $t1`
- If reversed
 - Read a future value!

RAR Dependence?

- Read-After-Read dependence
 - add \$t0, \$s0, \$t1
 - sub \$t2, \$s0, \$t3
- If reversed
 - No problem at all ^_^

WAW Dependence

- Write-After-Write dependence
 - add **\$s0**, \$t0, \$t1
 - sub **\$s0**, \$t2, \$t3
- If reversed
 - The final **\$s0** value is wrong!