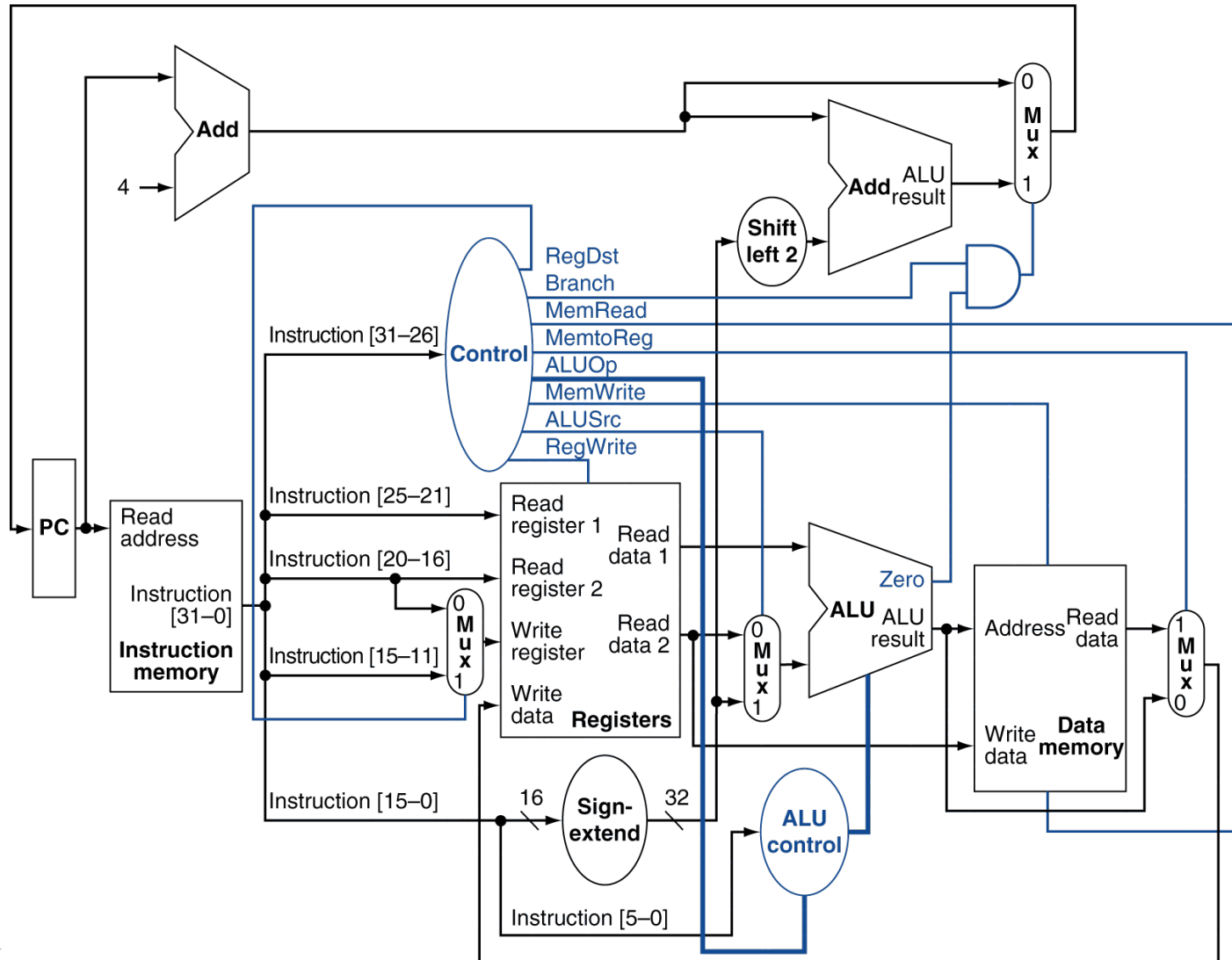




Topic 7

Pipelined Processor

Single Cycle Implementation

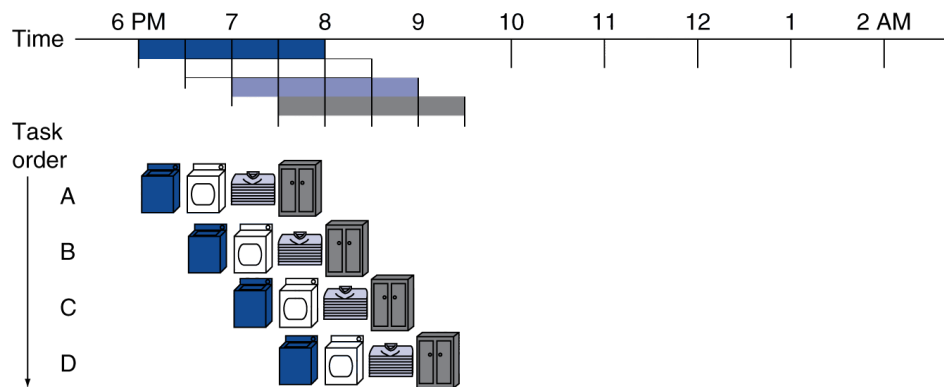
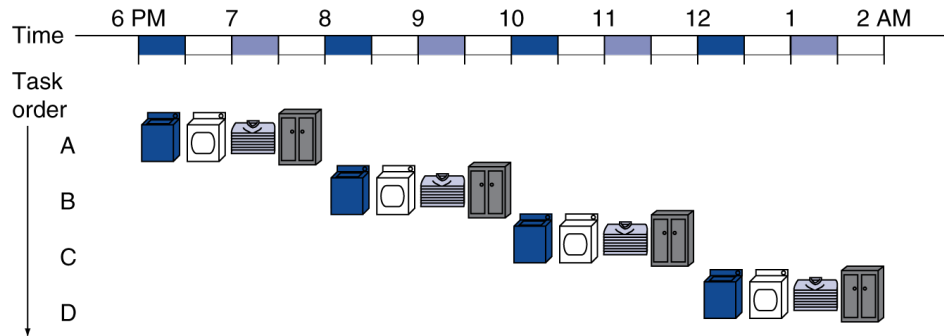


Performance Issues

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

Pipelining Analogy

- Pipelined laundry: overlapping execution
 - Parallelism improves performance



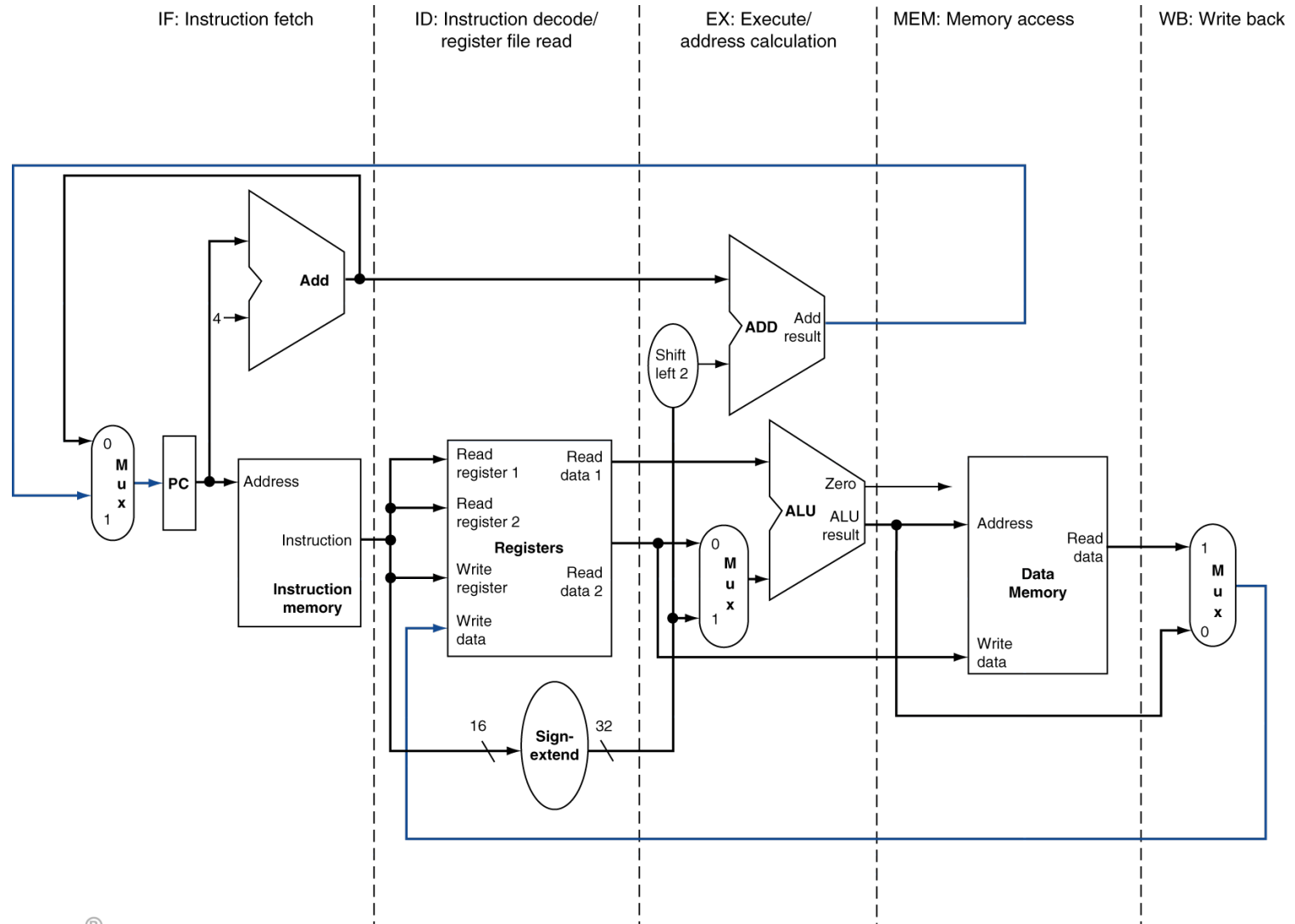
- Four loads:

- Speedup
 $= 8 / 3.5 = 2.3$

- Non-stop:

- Speedup
 $\approx 2 * n / 0.5 * n = 4$
 $= \text{number of stages}$
 - n is number of instructions

MIPS Pipelined Datapath

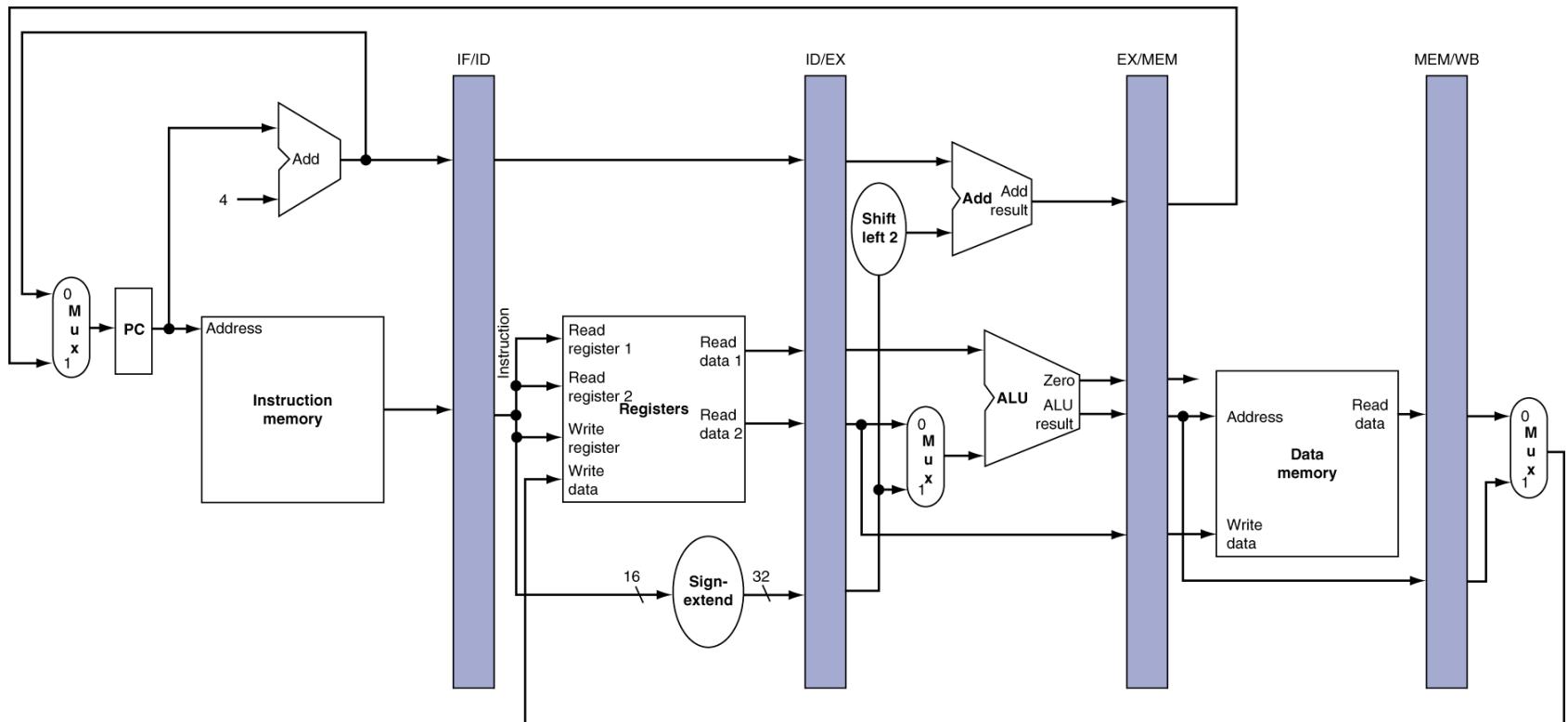


MIPS Pipeline

- Five stages, one step per stage per cycle
 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

Pipeline registers

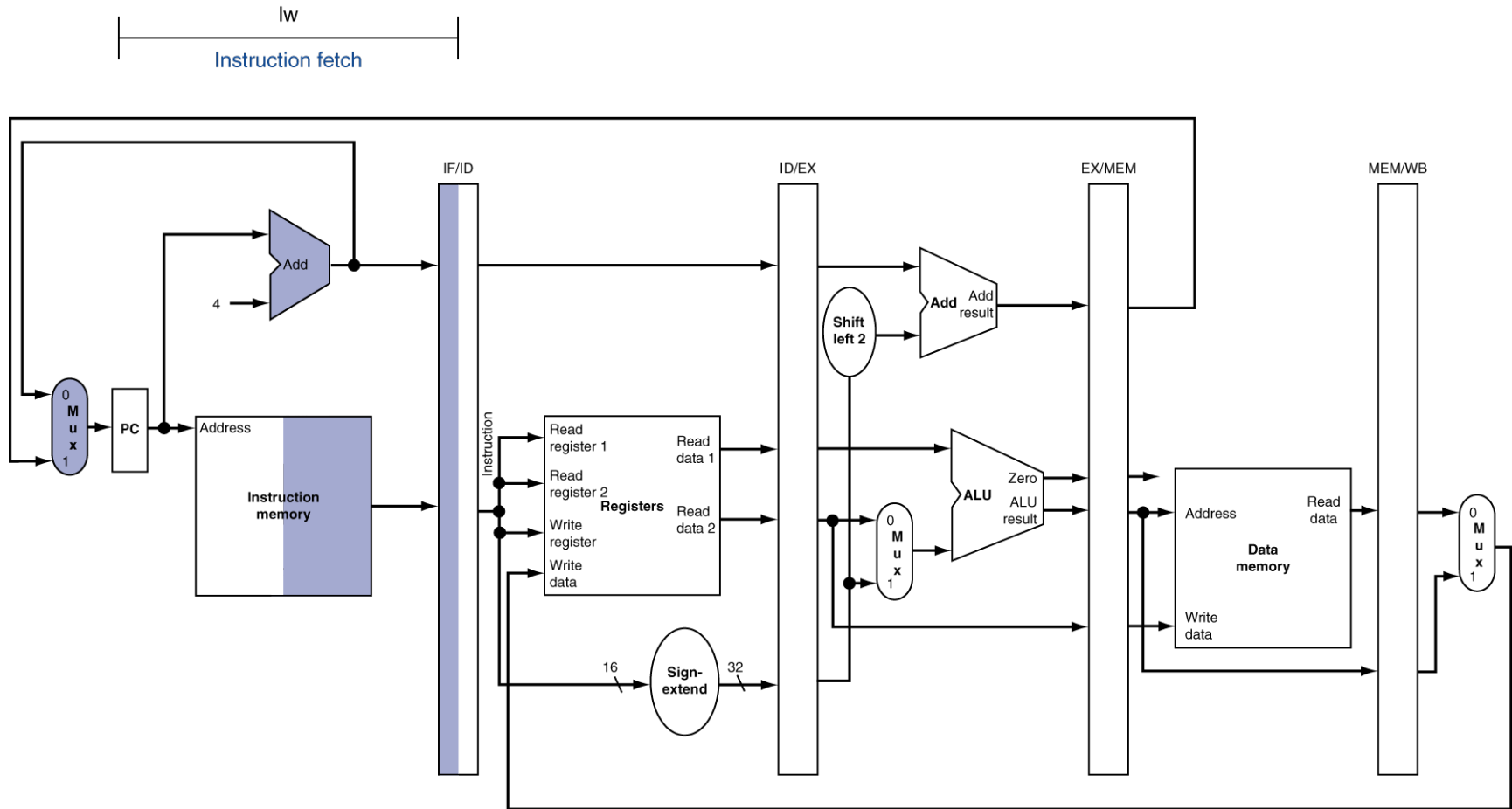
- Need registers between stages
 - To hold information produced in previous cycle



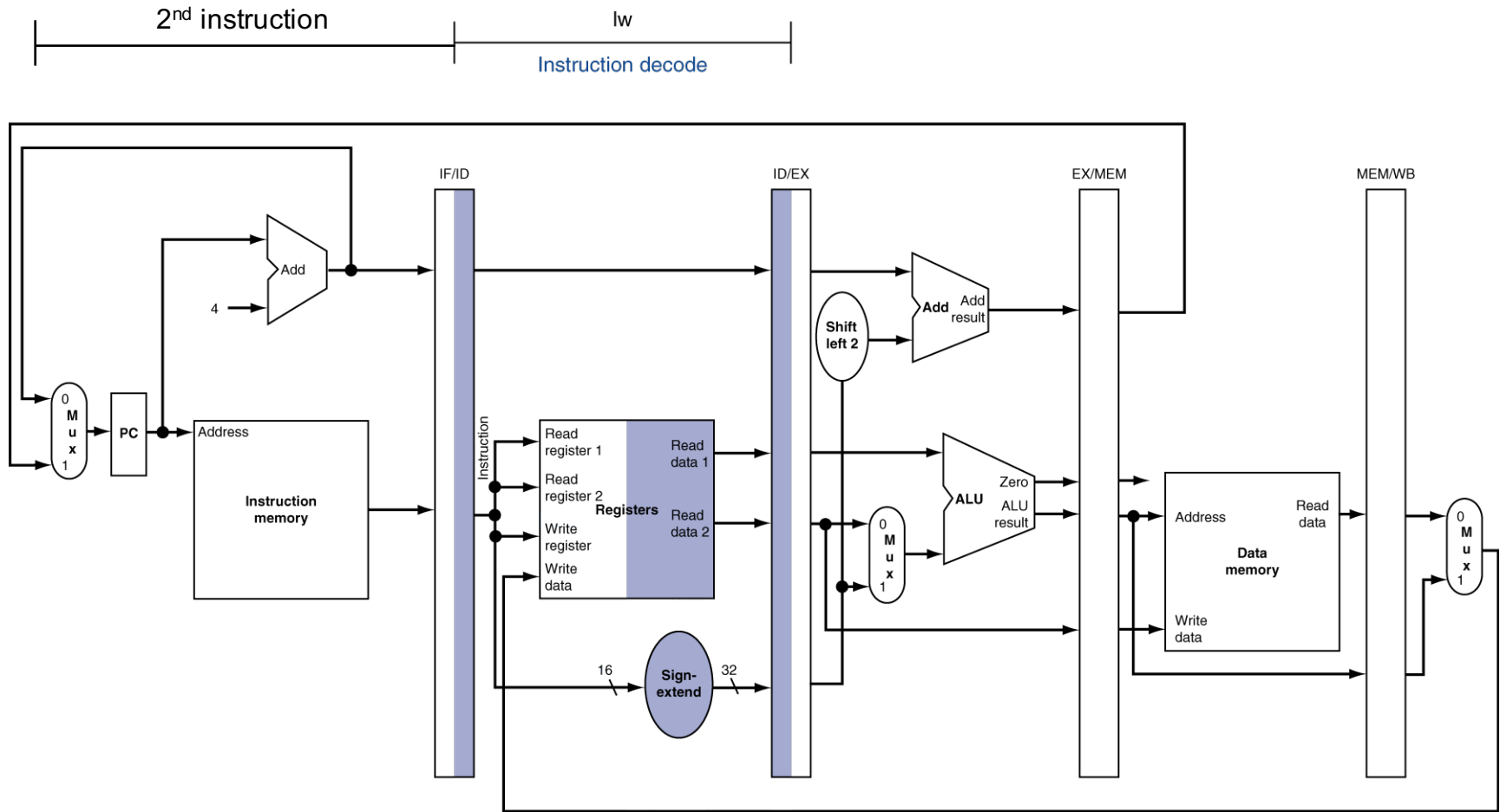
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
- Representation:
 - “Single-clock-cycle” pipeline diagram
 - Shows pipeline usage in a single cycle
 - Highlight resources used
 - “multi-clock-cycle” diagram
 - Graph of operation over time

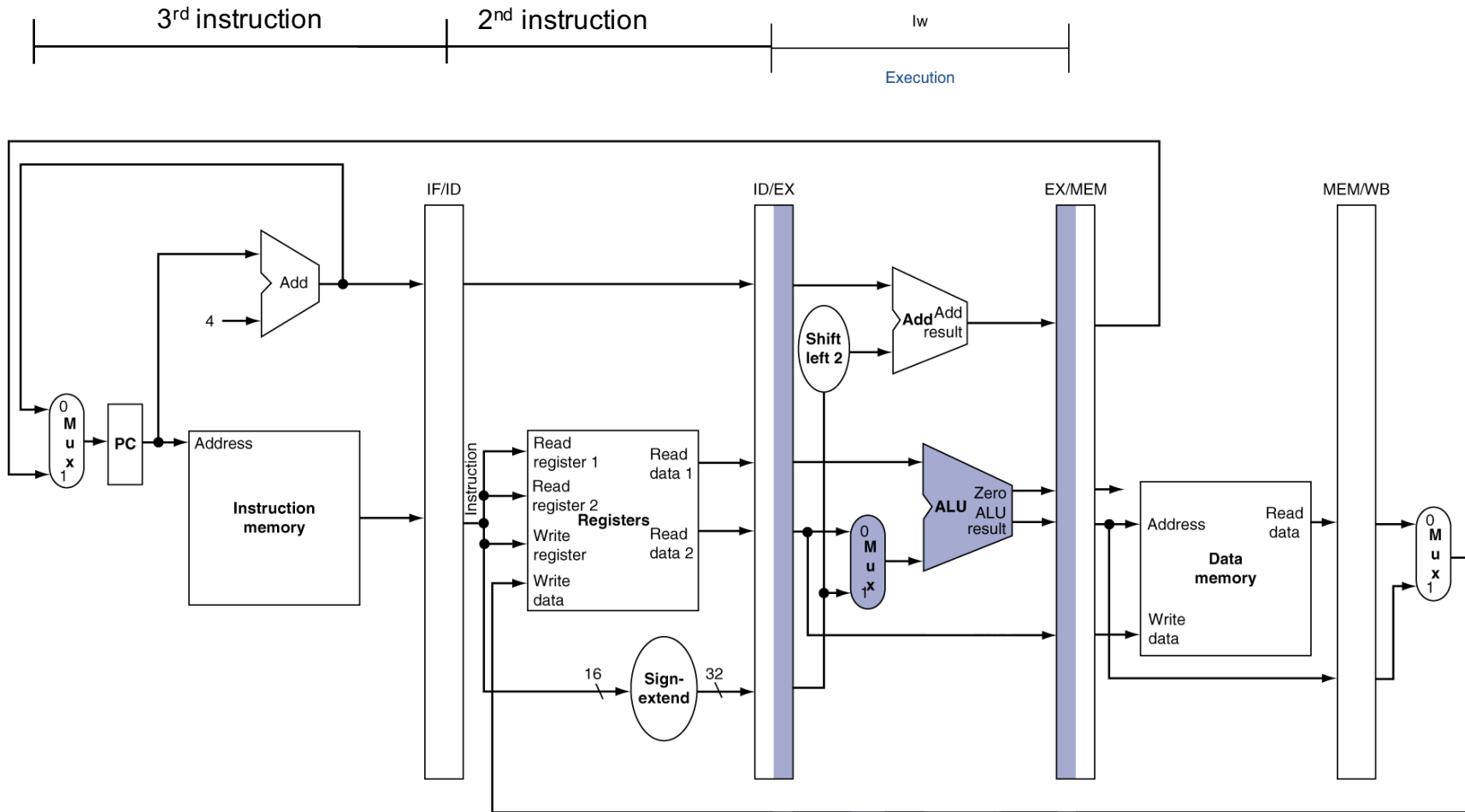
IF for Load, Store, ...



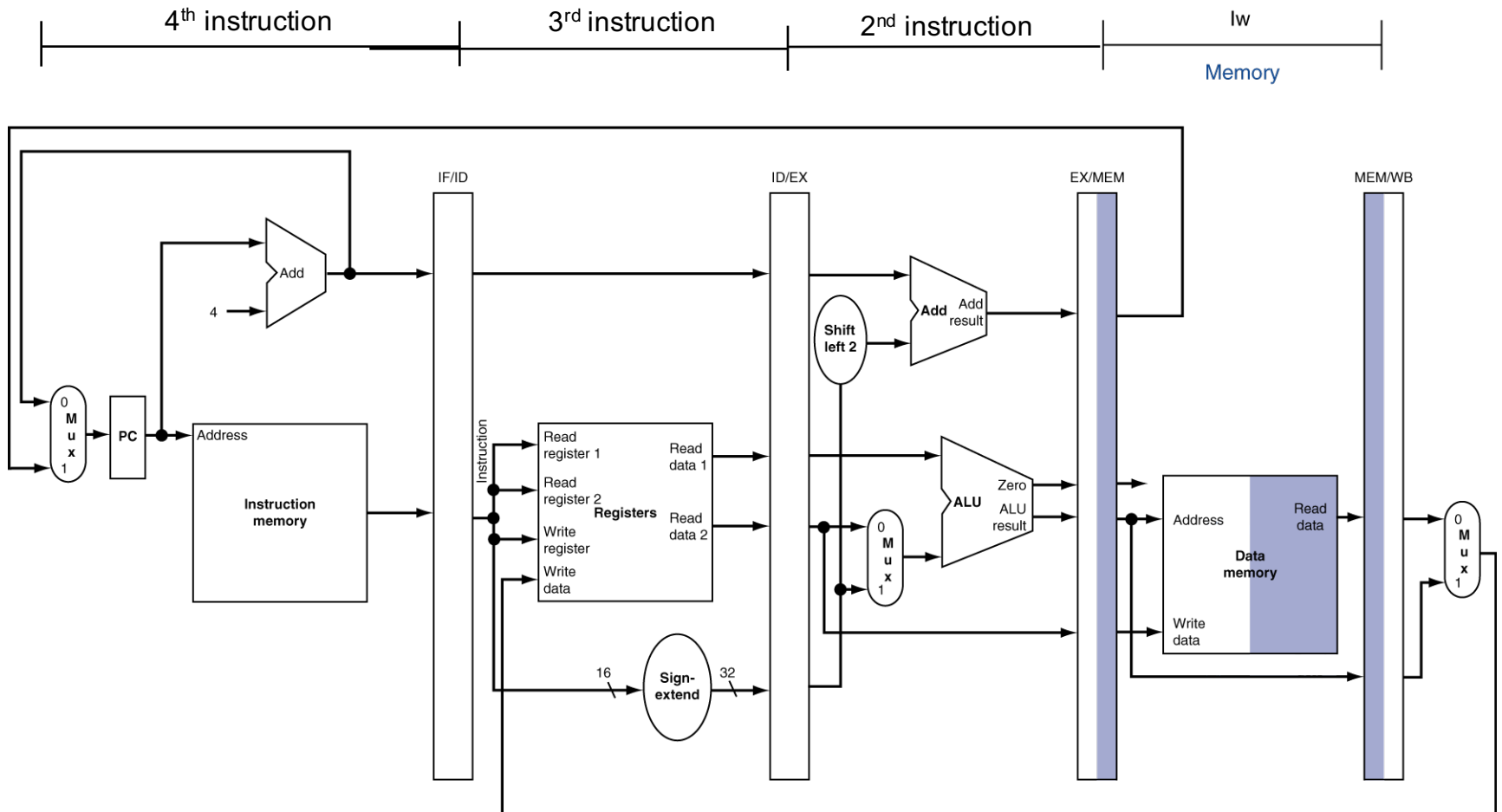
ID for Load, Store, ...



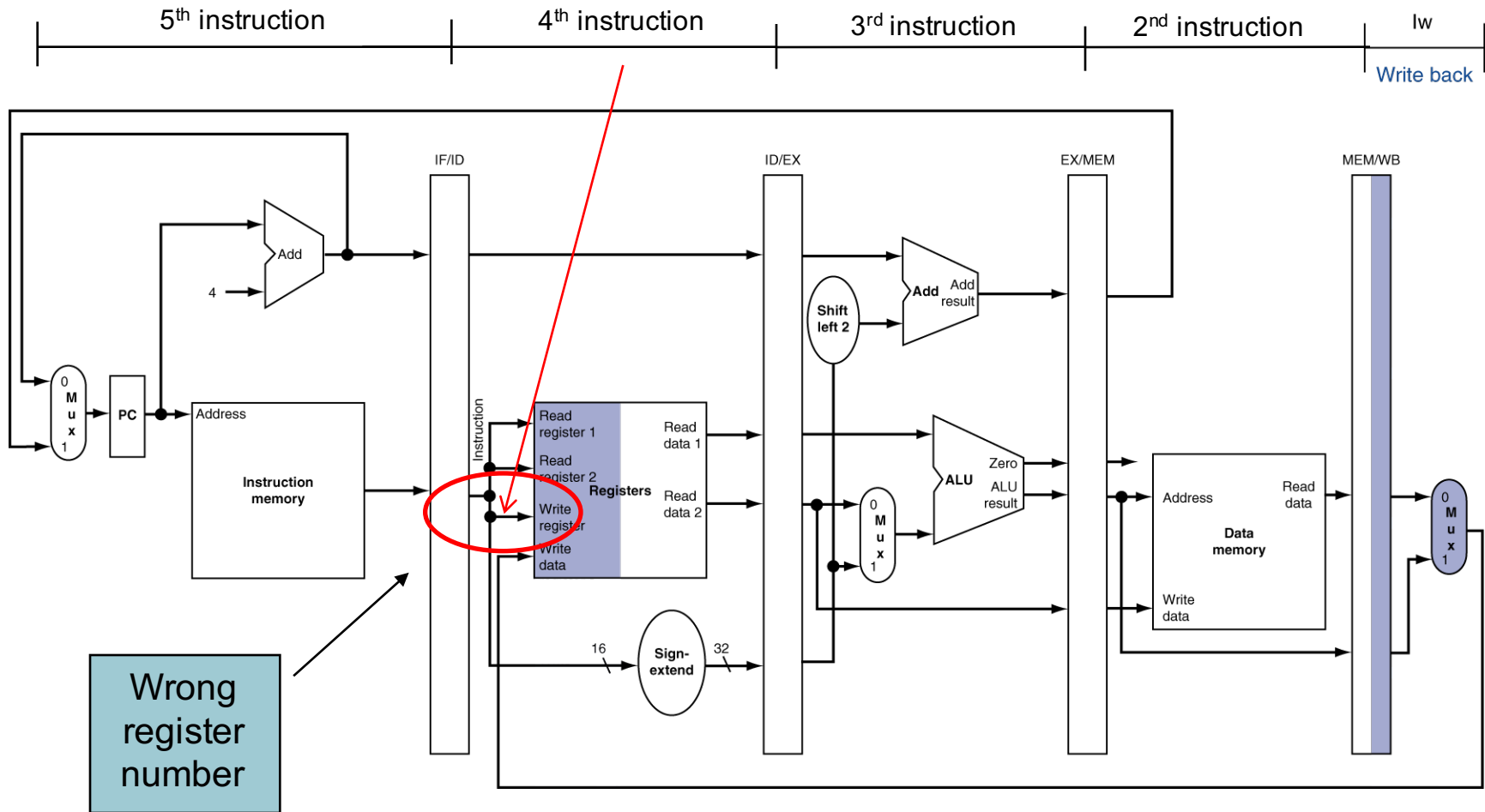
EX for Load



MEM for Load



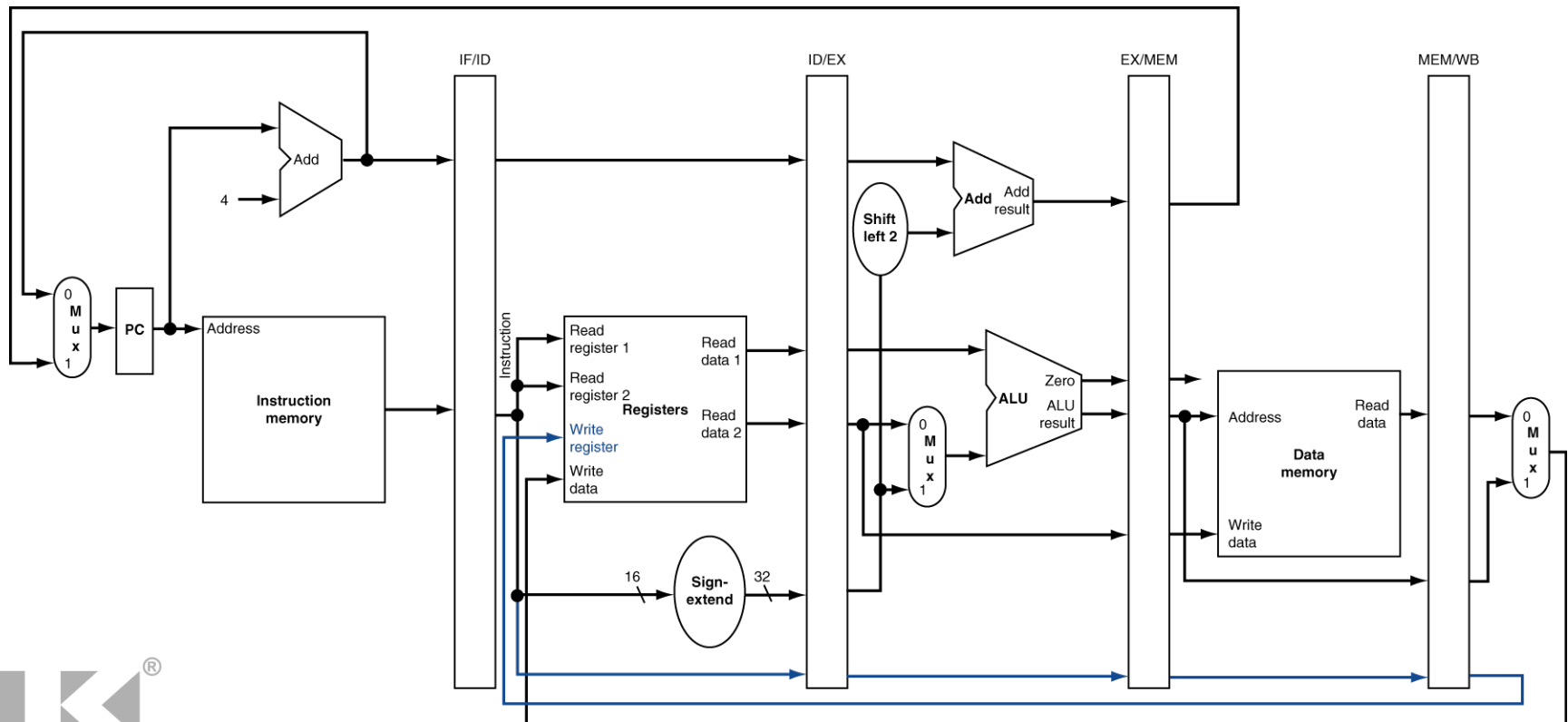
WB for Load



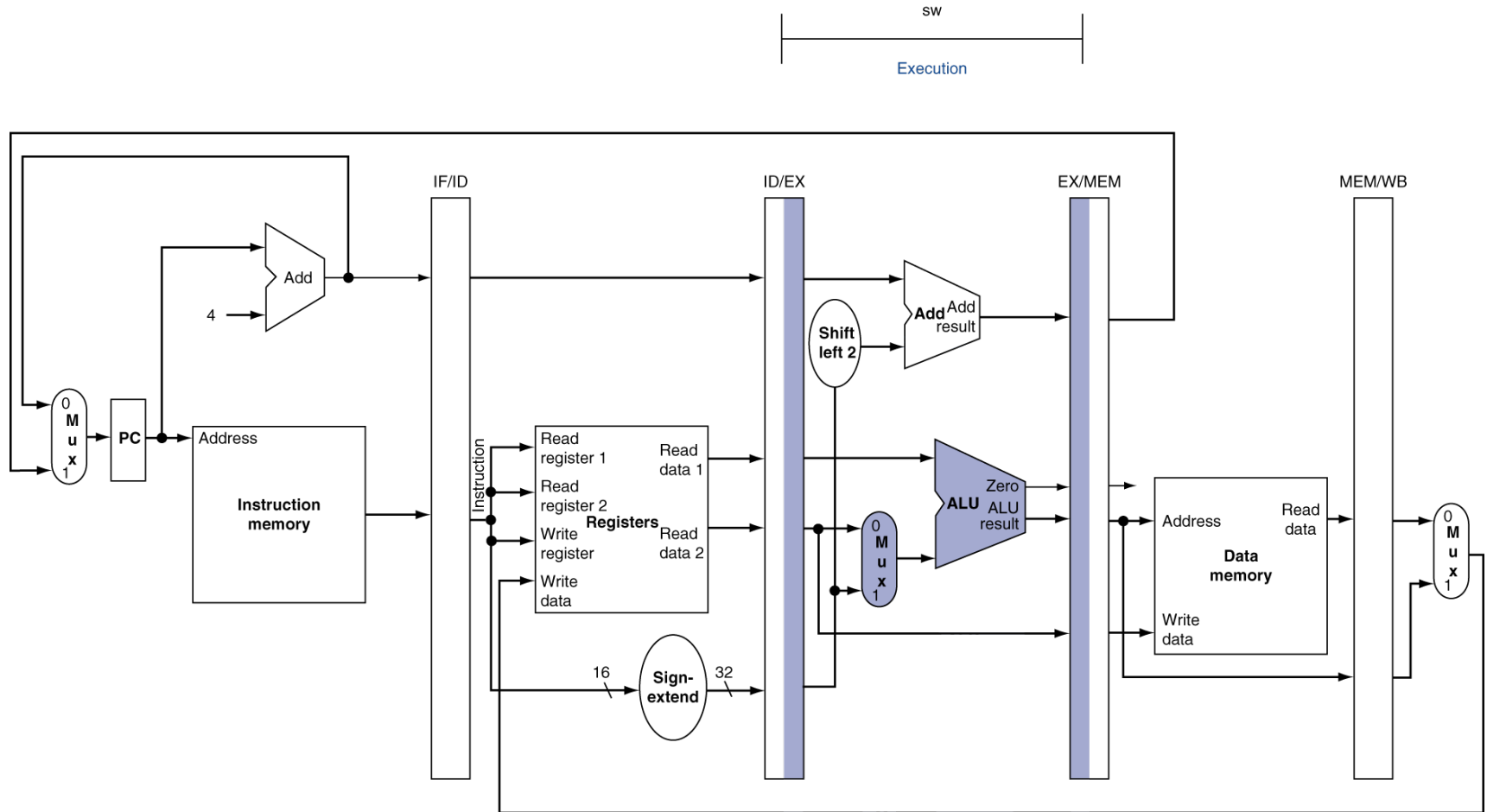
Wrong register number

Corrected Datapath for Load

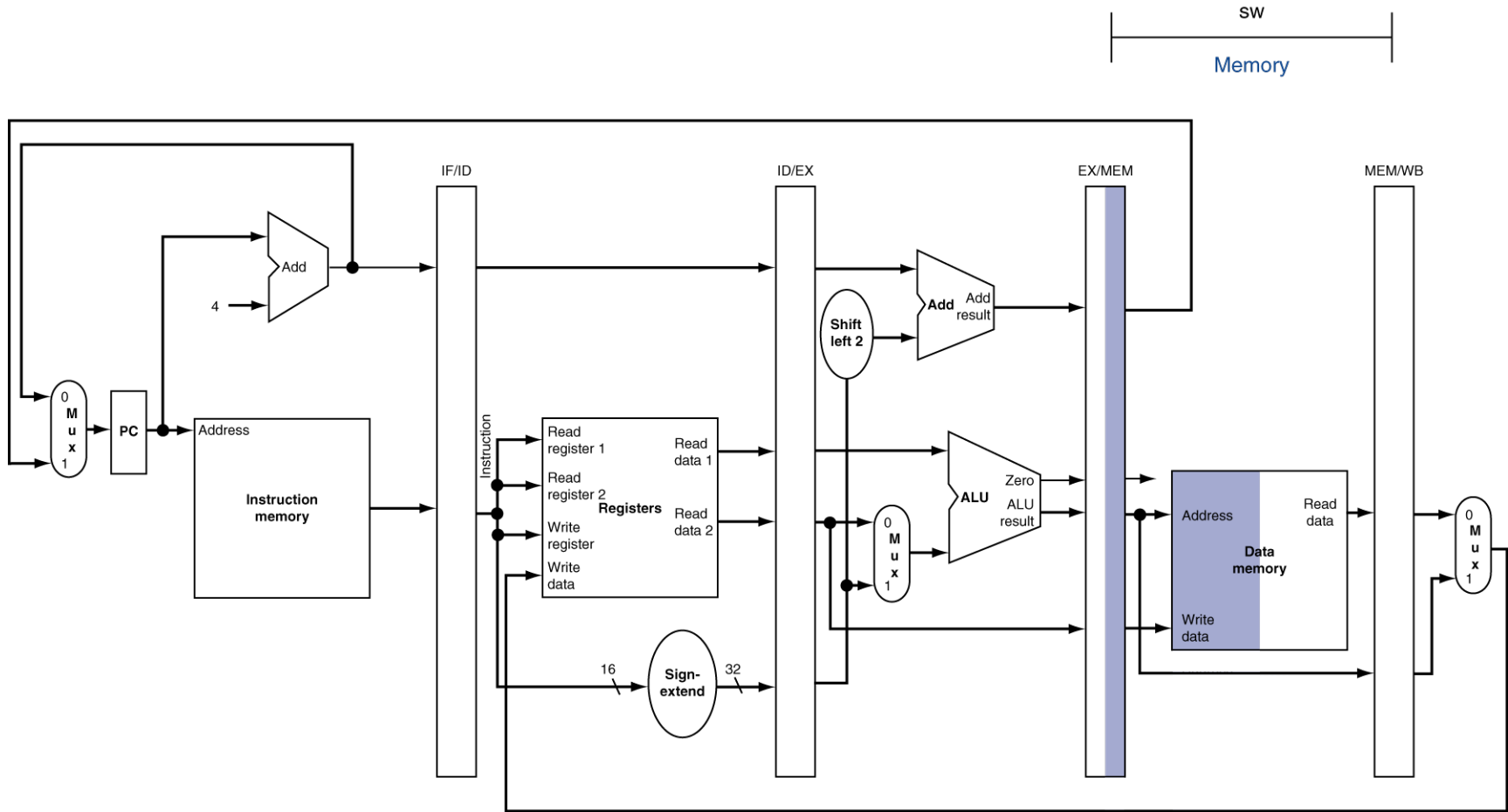
- Pass alive signals along through the pipeline
- Has to write/read register file at the same time
 - Writing reg in first half of clock
 - Reading reg in second half of clock



EX for Store

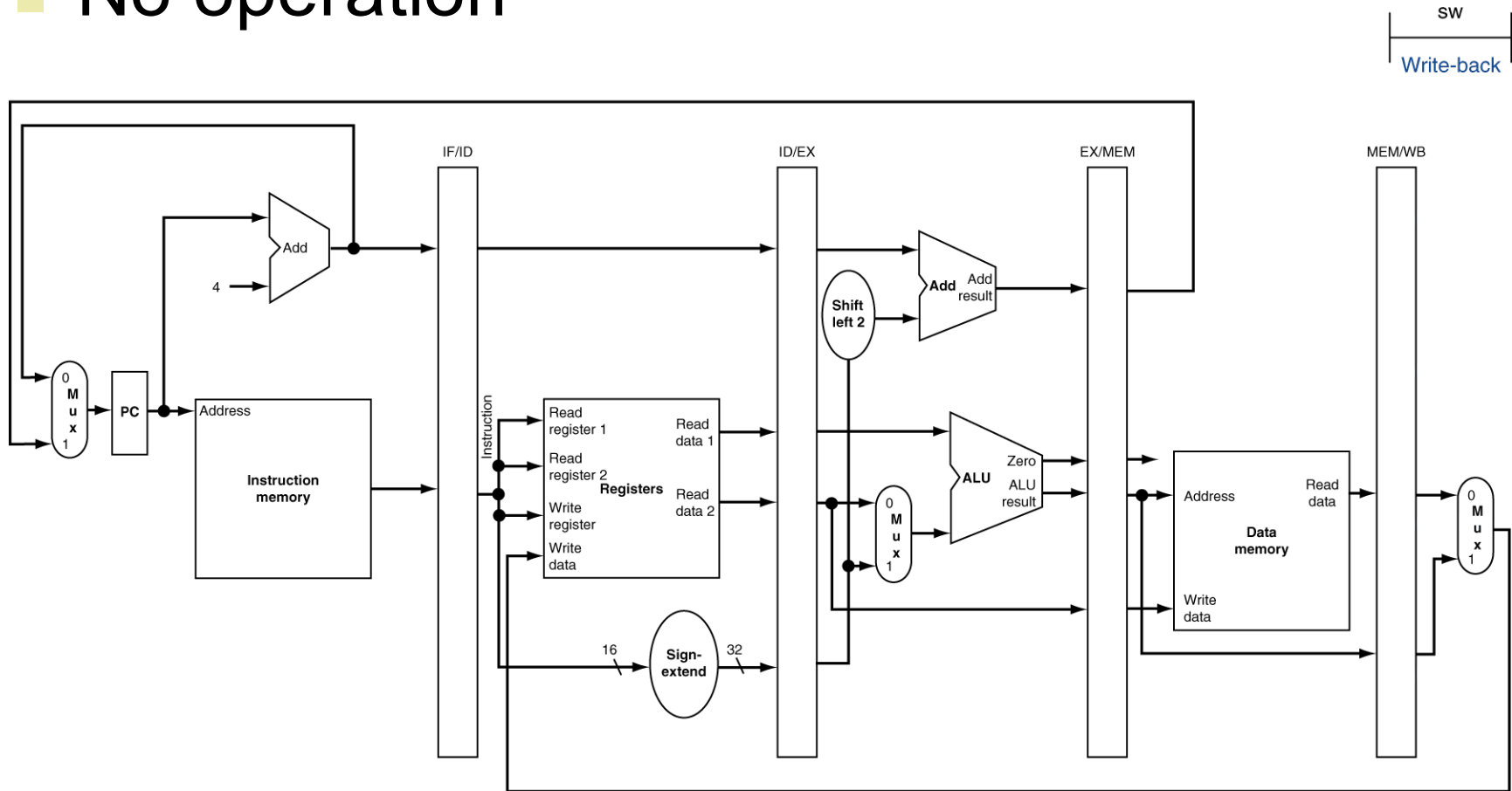


MEM for Store



WB for Store

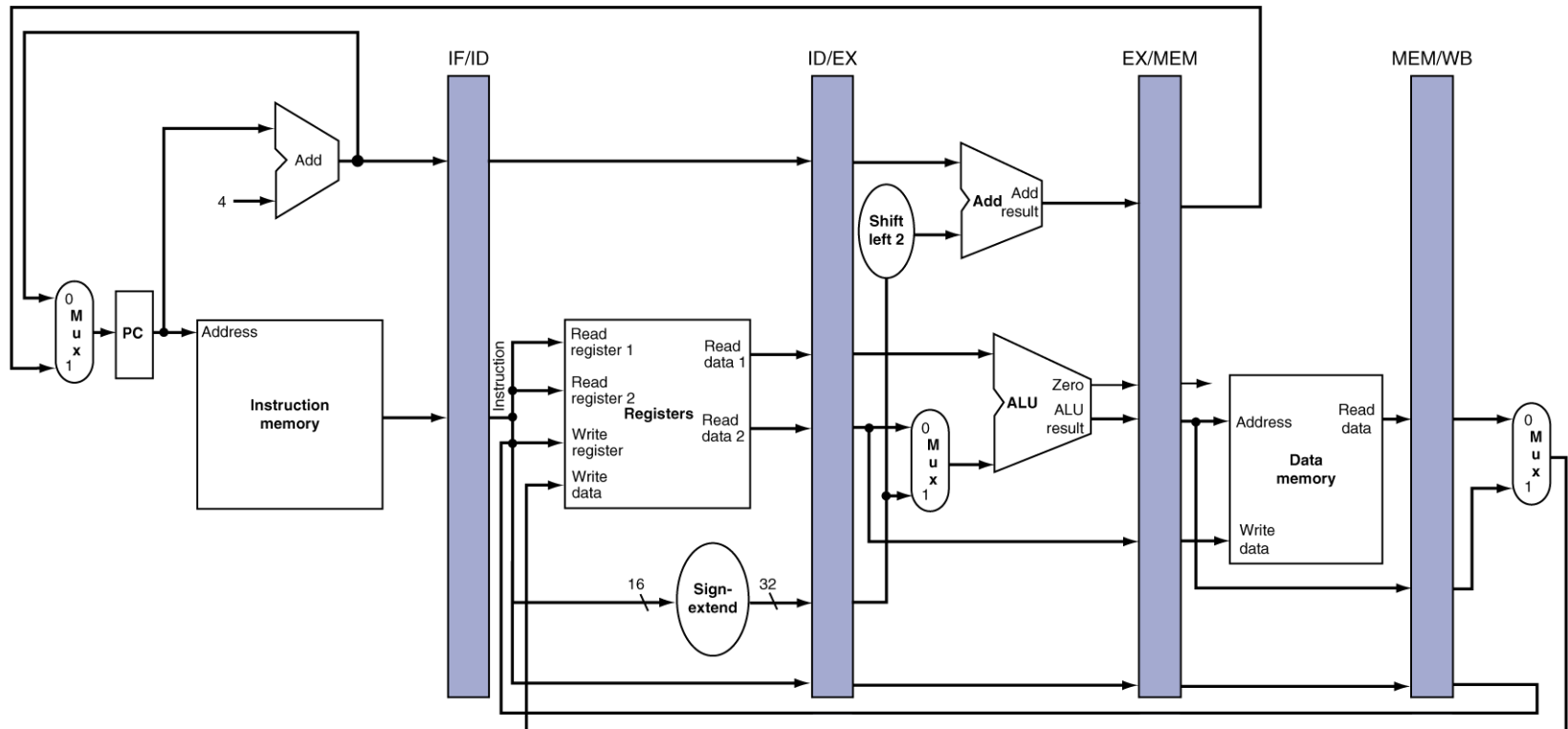
- No operation



Single-Cycle Pipeline Diagram

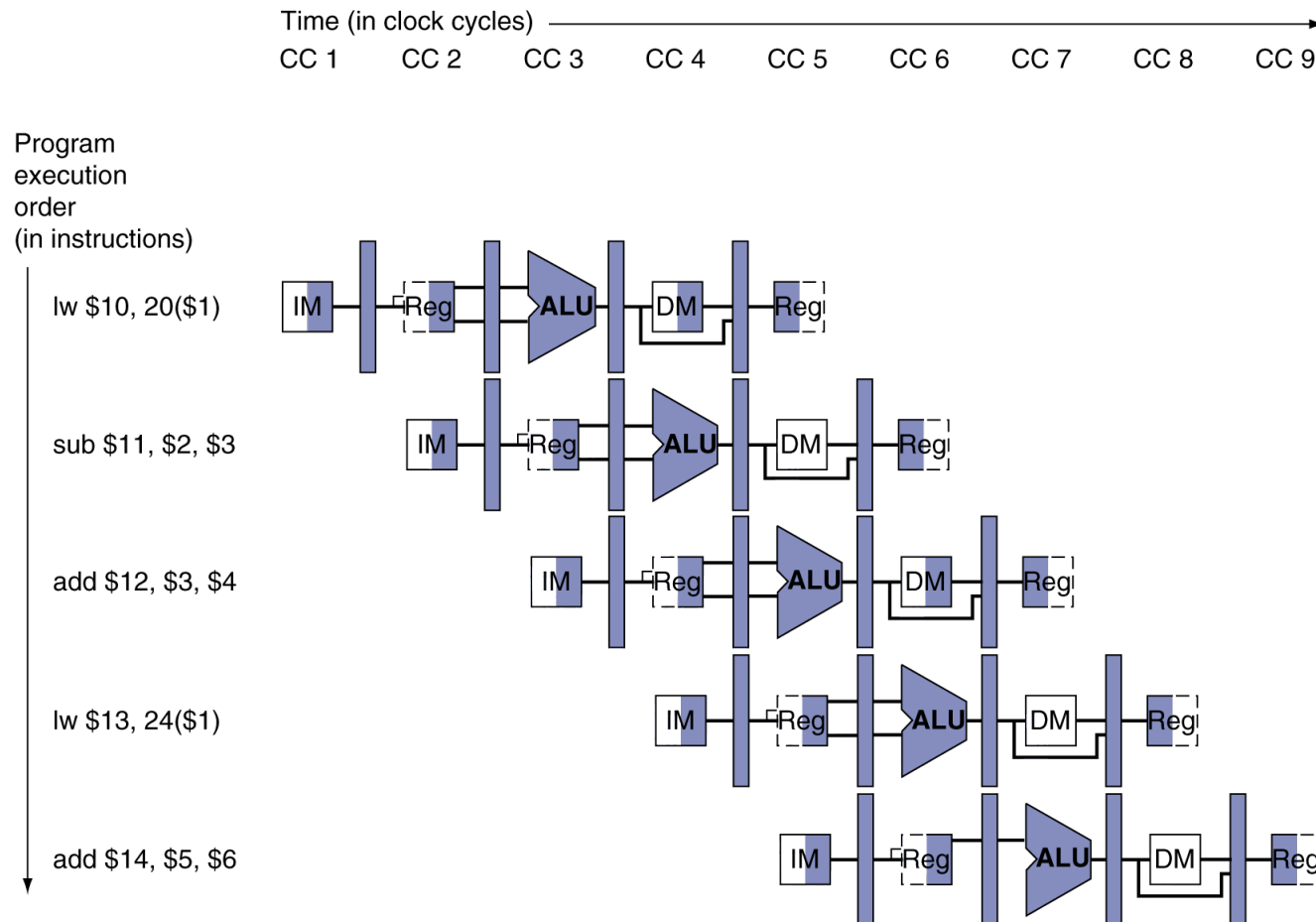
■ State of pipeline in a given cycle

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



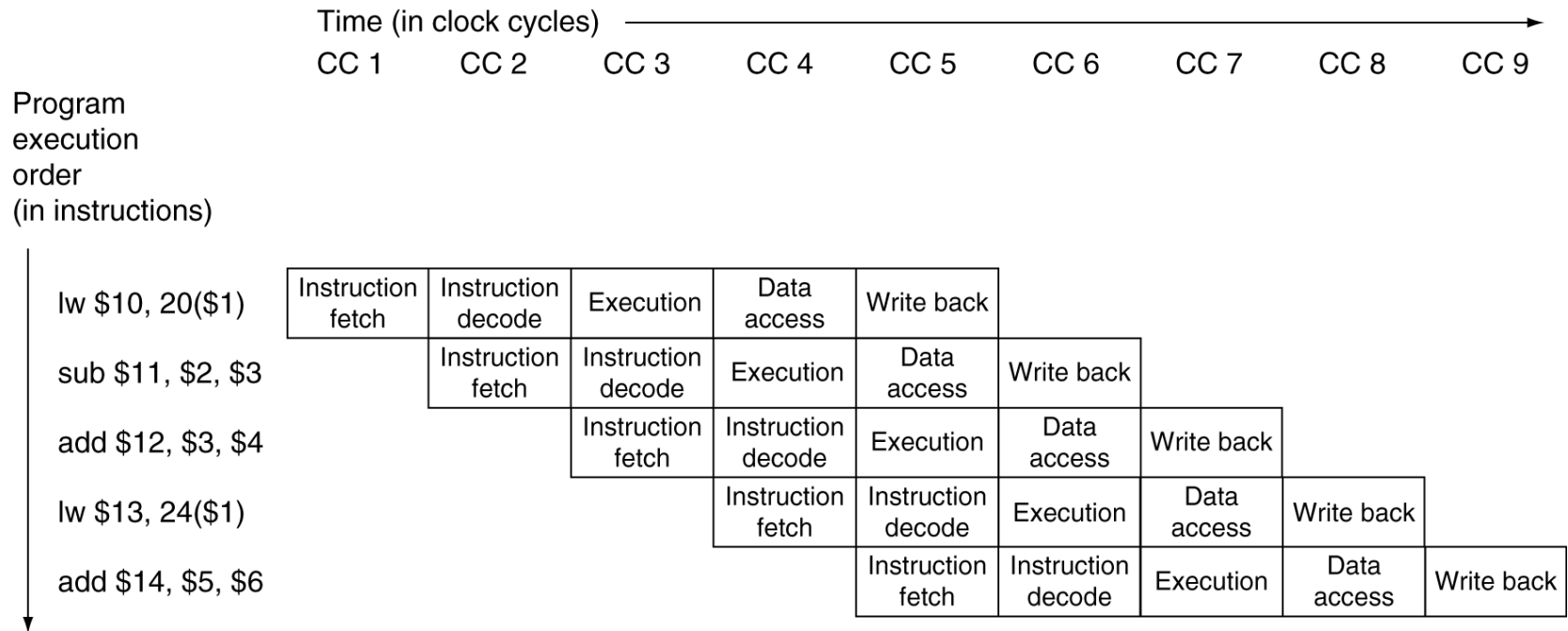
Multi-Cycle Pipeline Diagram

- Form showing resource usage

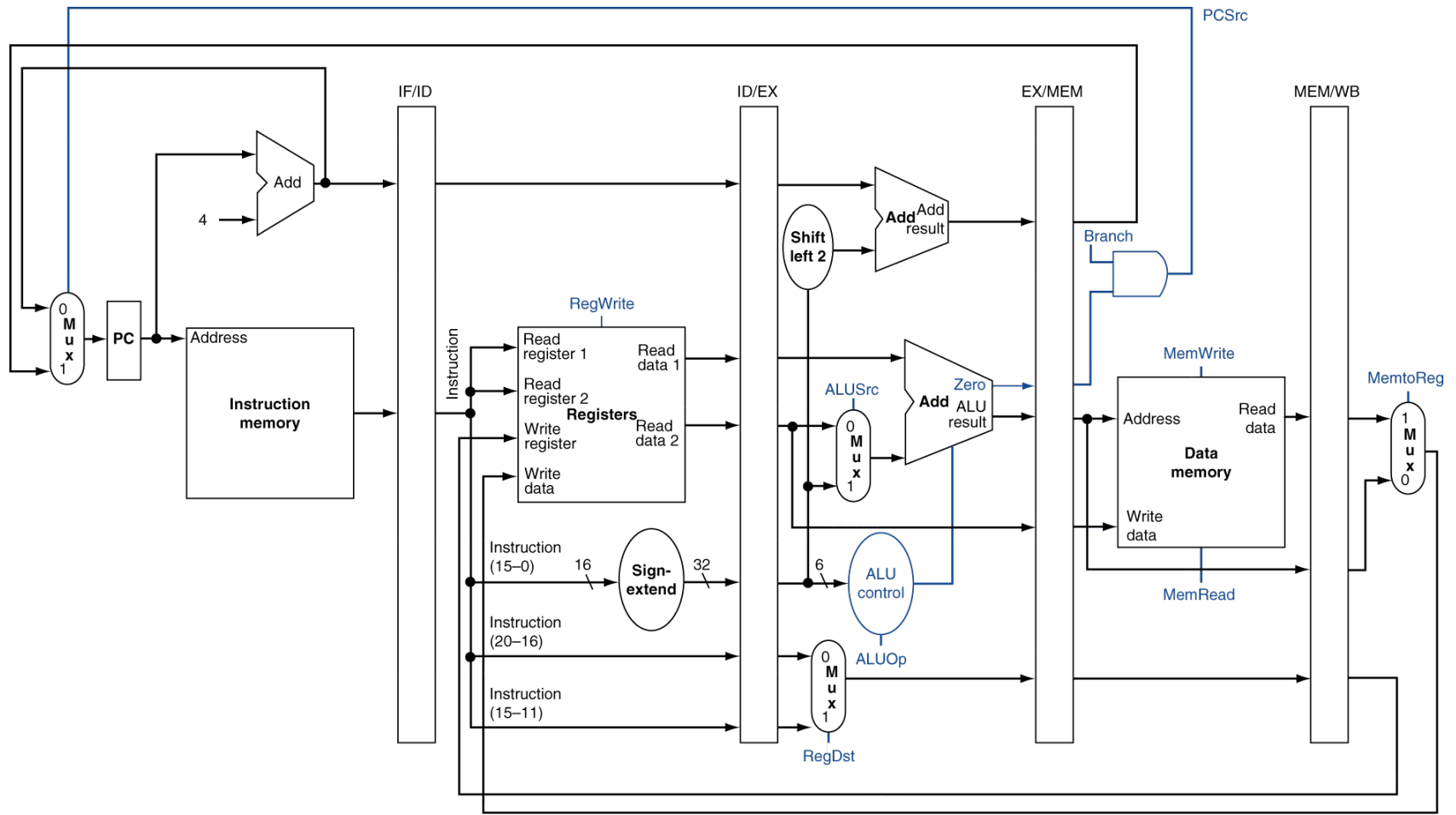


Multi-Cycle Pipeline Diagram

■ Traditional form

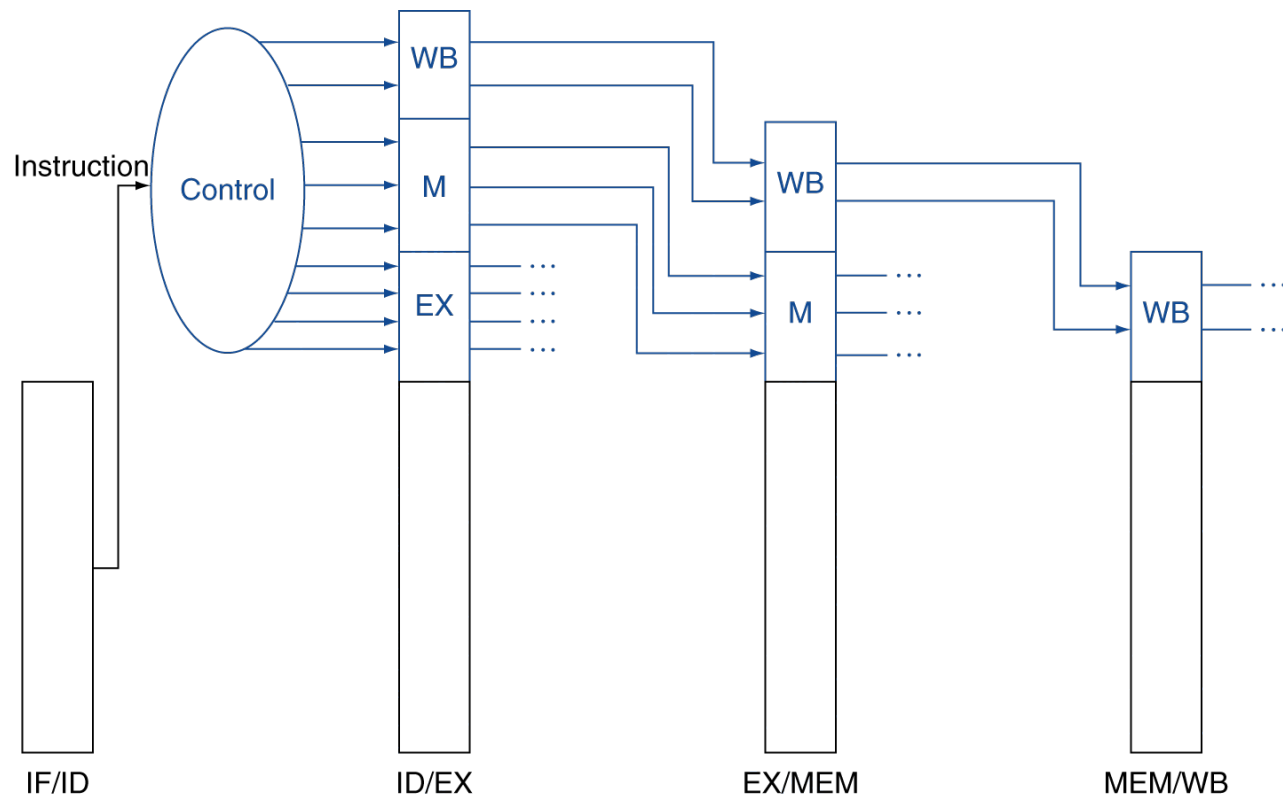


Pipelined Control (Simplified)

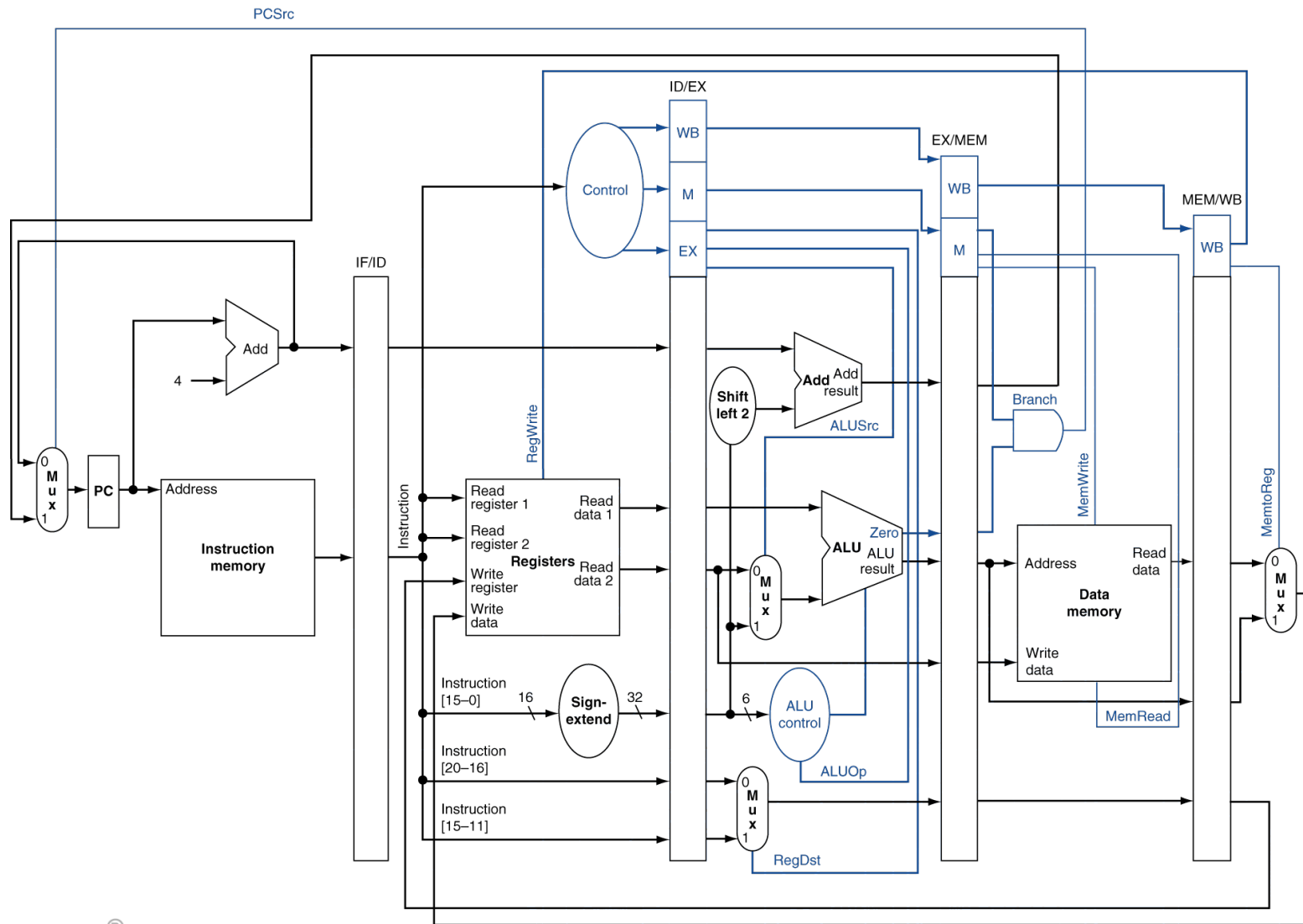


Pipelined Control

- Control signals derived from instruction
 - Passed along with corresponding instruction
 - Consumed in appropriate stages



Pipelined Control



Pipeline Summary

The BIG Picture

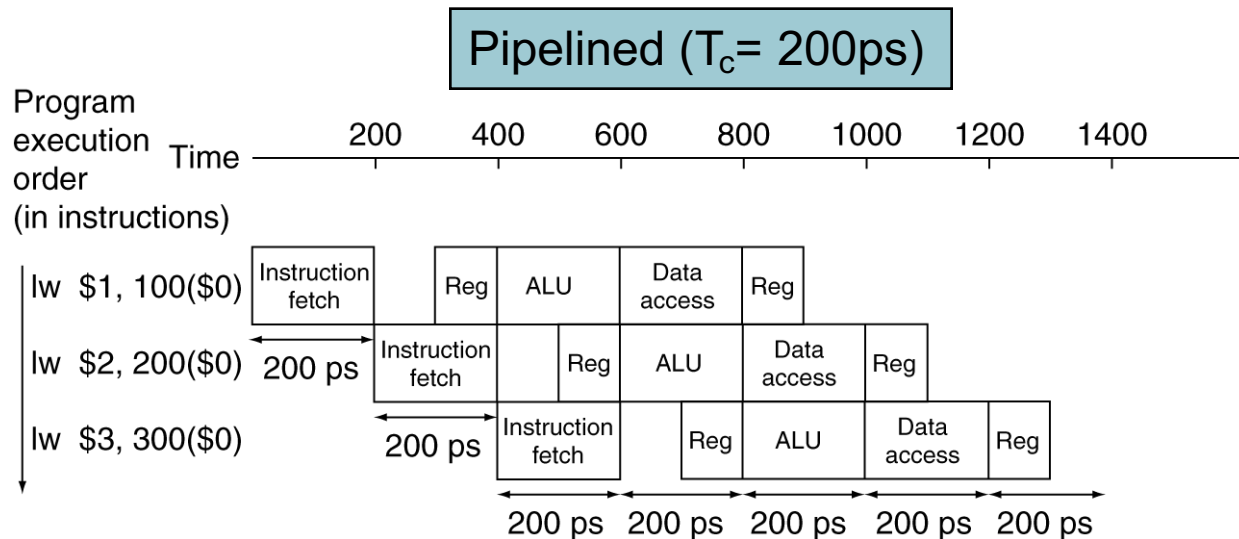
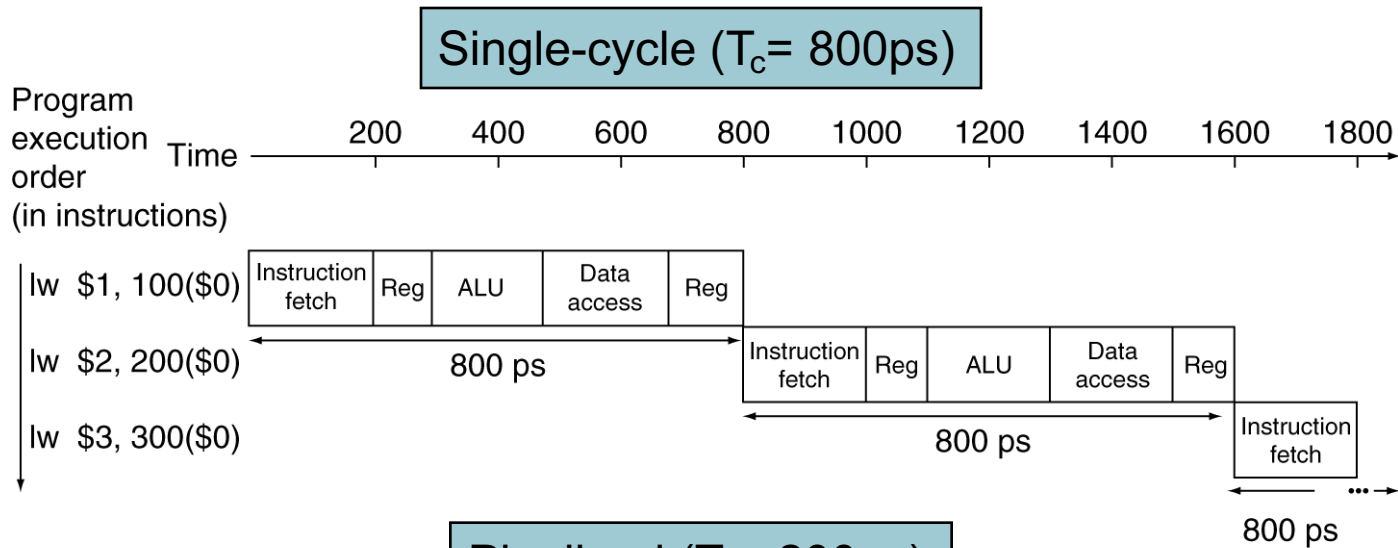
- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

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