



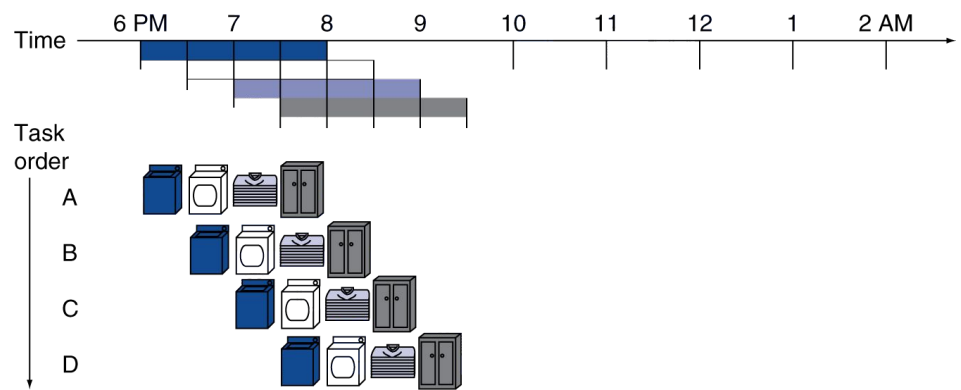
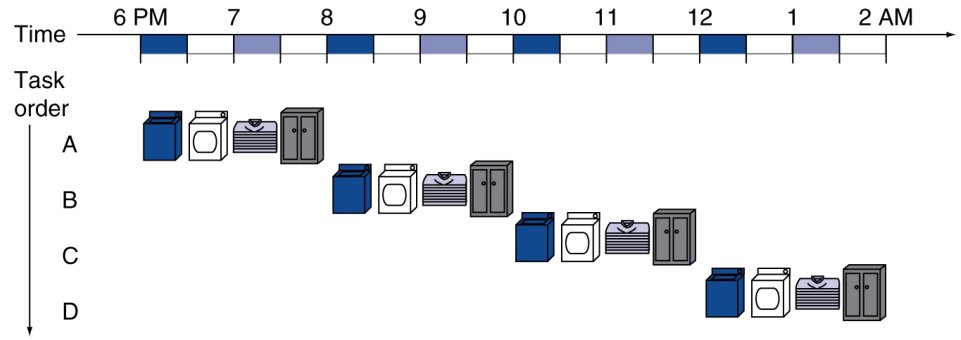
# Pipelining

# 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

# Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

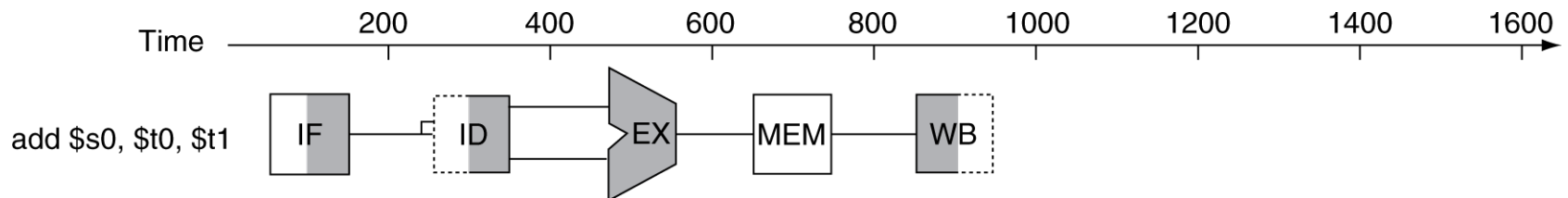


- Four loads:
  - Speedup =  $16/7 = 2.3$
- Non-stop:
  - Speedup =  $4n/n + 3 \approx 4$   
= number of stages

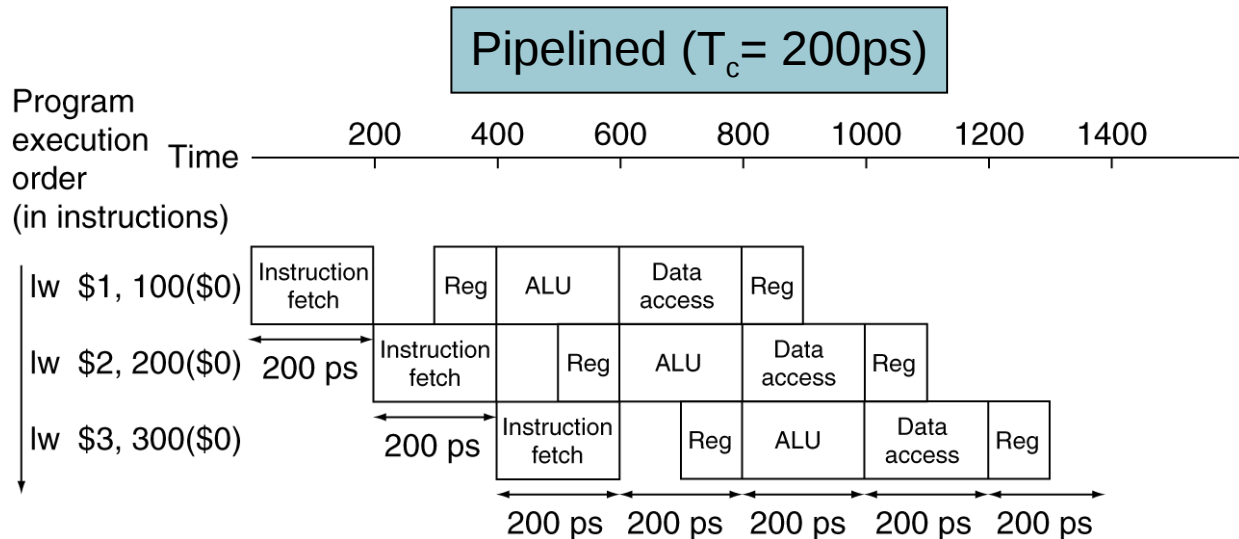
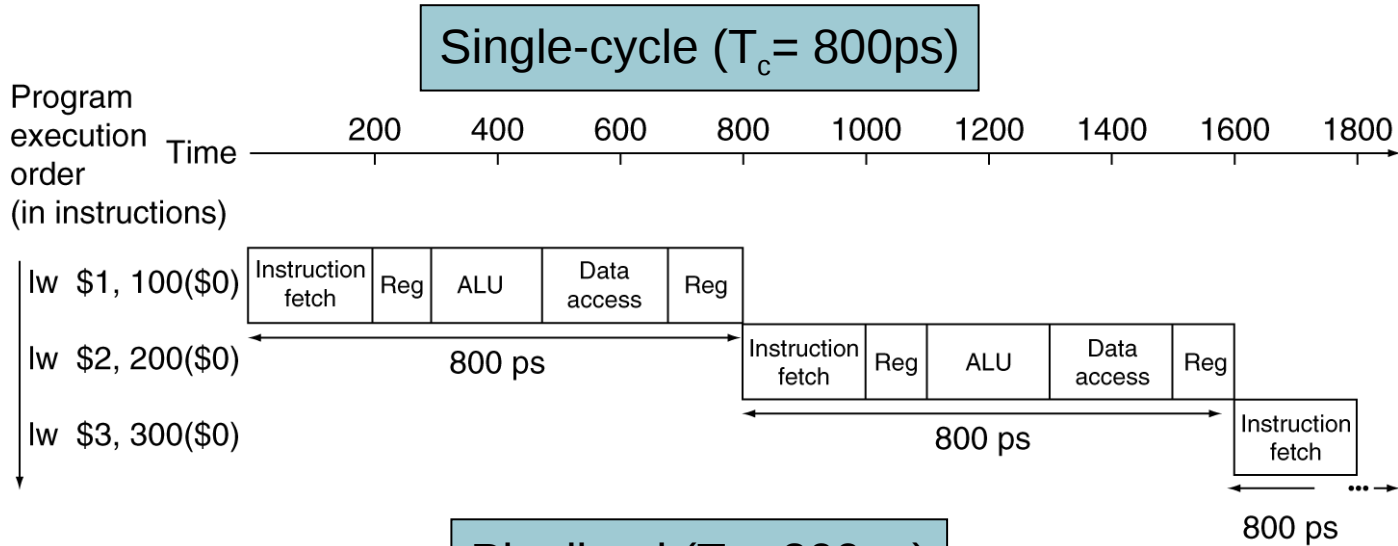


# MIPS Pipeline

- 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



# Pipeline Performance



# Pipeline Speedup

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

# 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

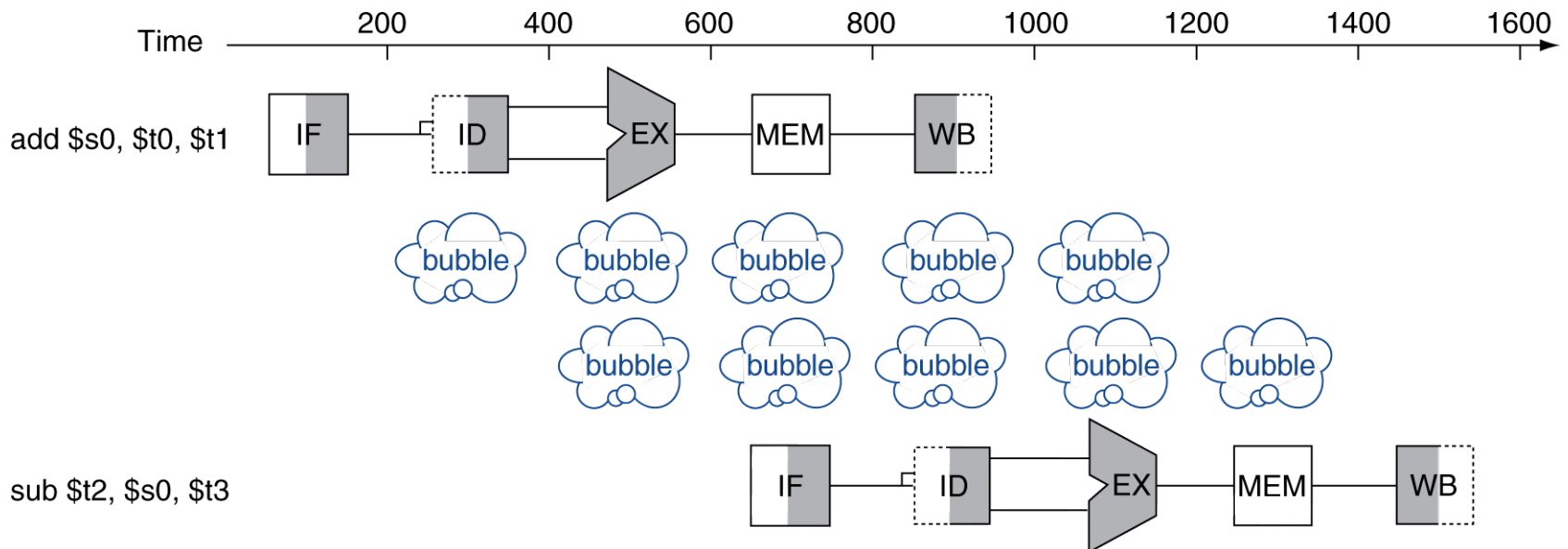
# Structure Hazards

- 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



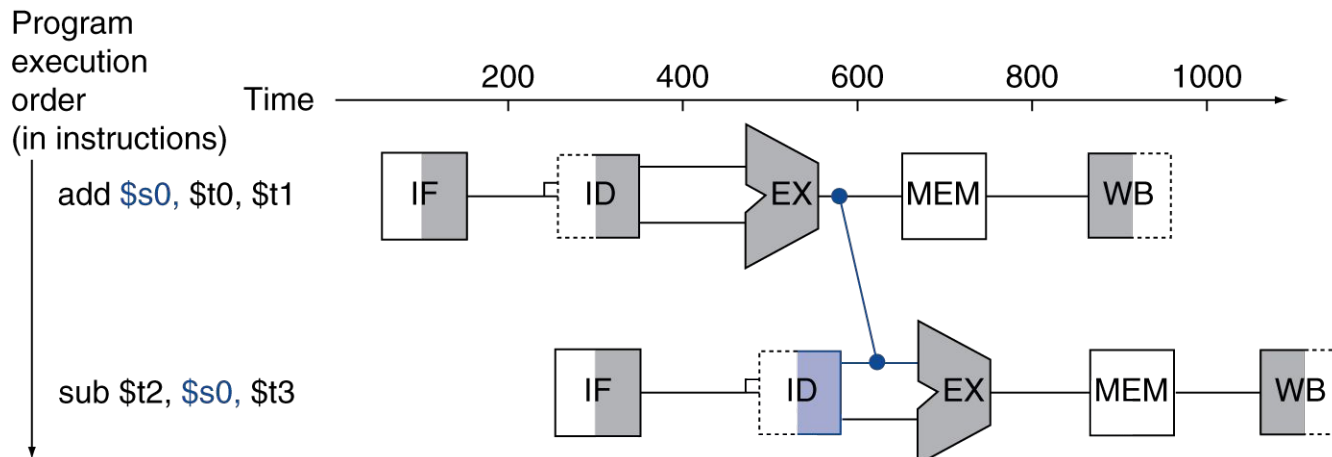
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add **\$s0**, \$t0, \$t1
  - sub \$t2, **\$s0**, \$t3



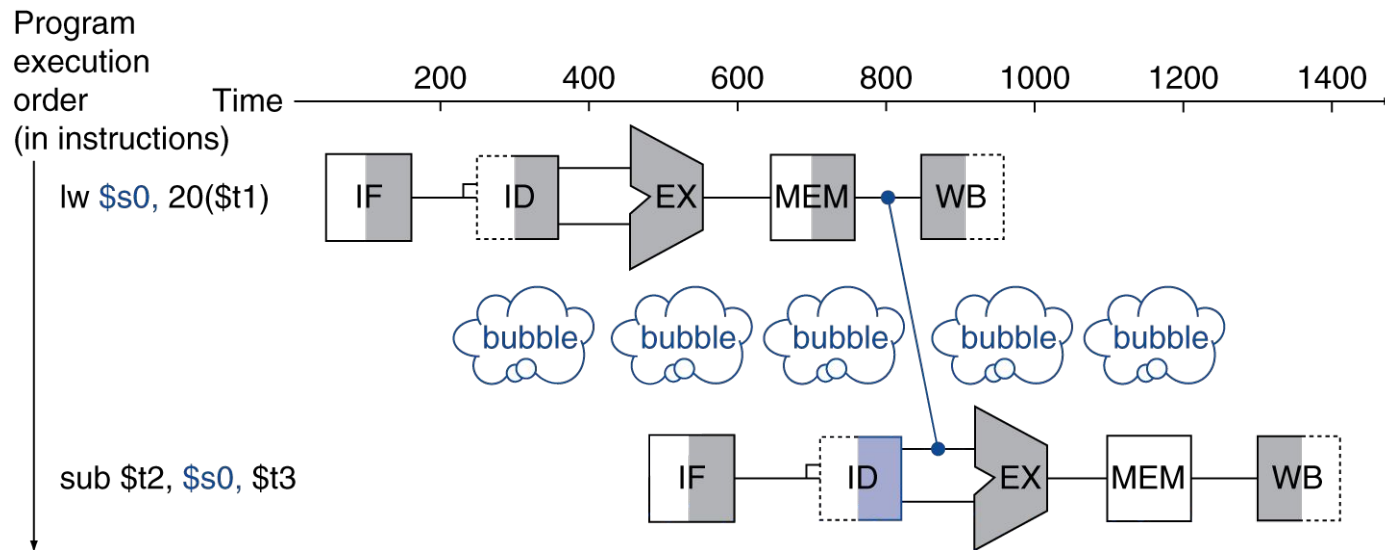
# Forwarding (aka Bypassing)

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



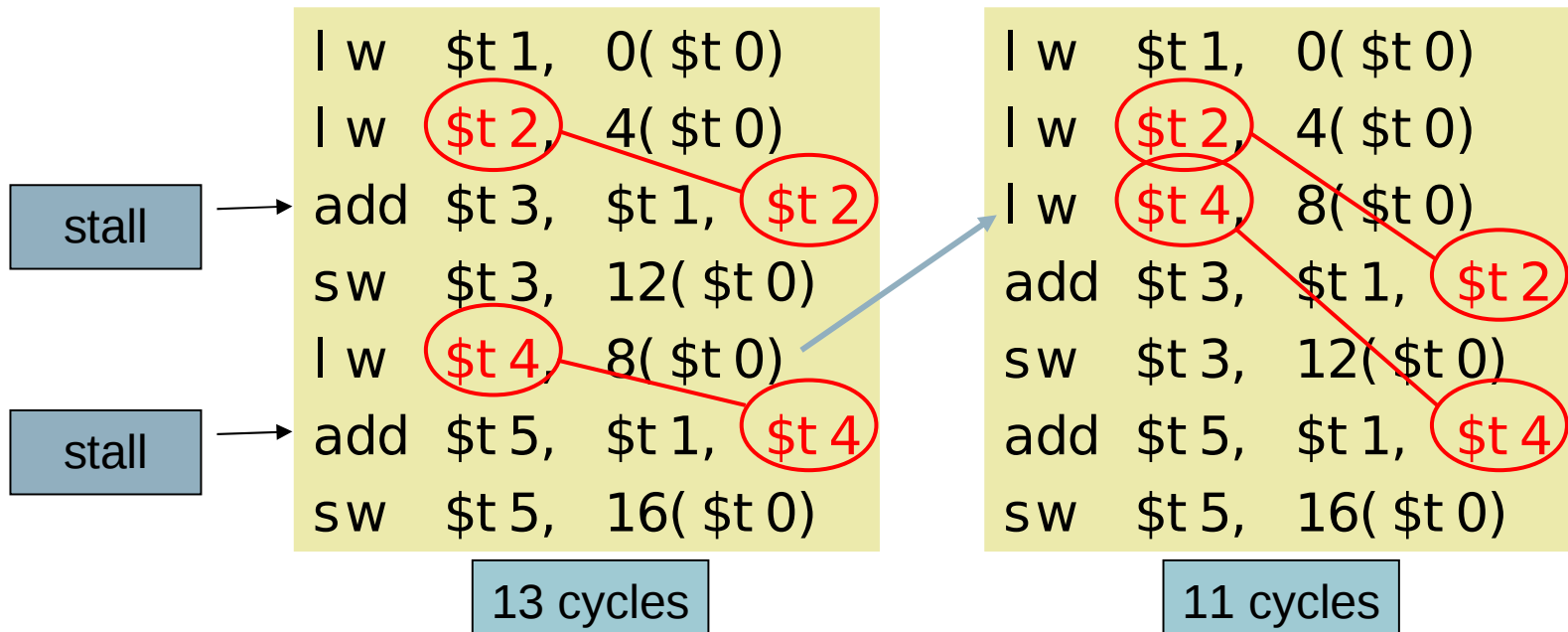
# Load-Use Data Hazard

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

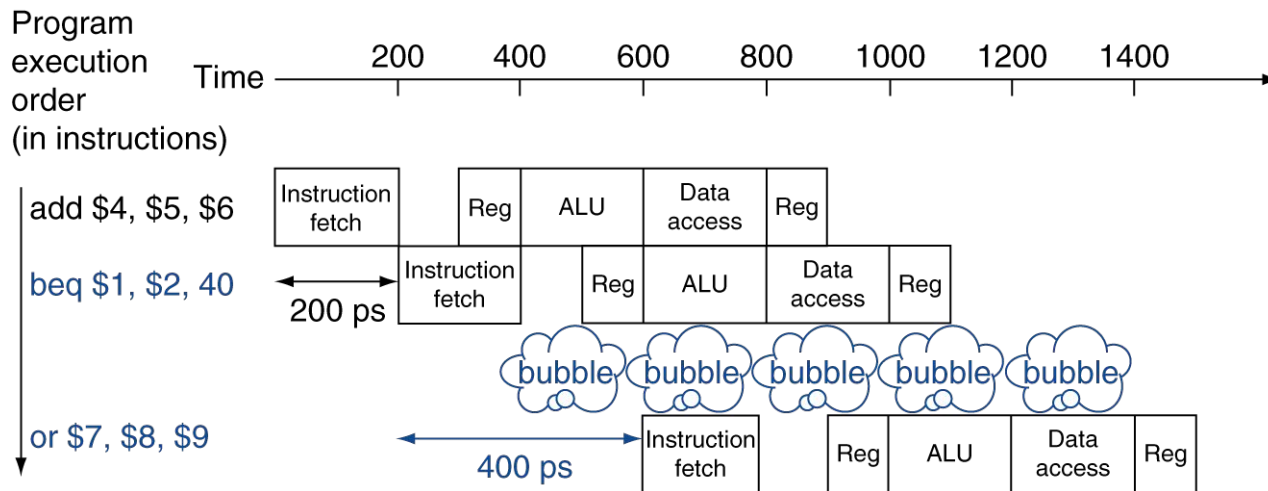


# Control Hazards

- 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
- 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

- Wait until branch outcome determined before fetching next instruction

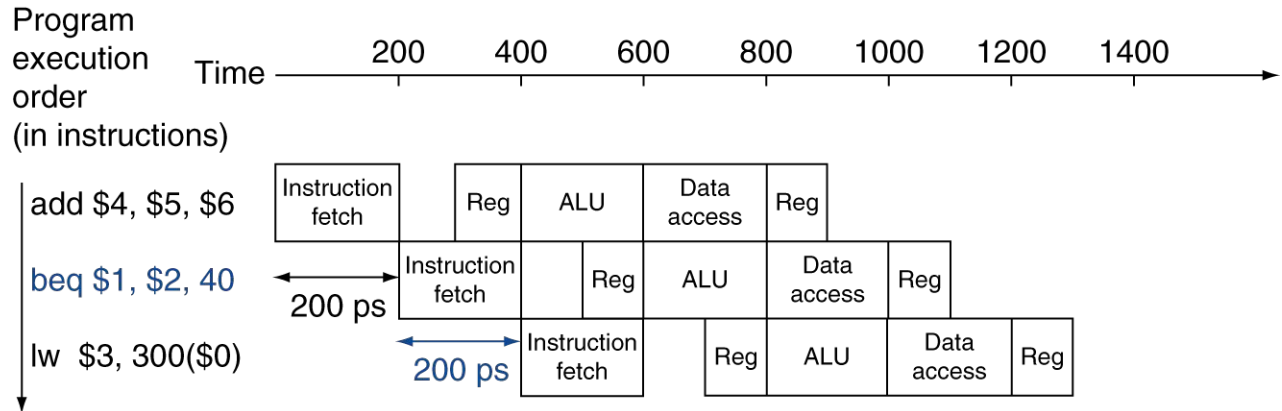


# Branch Prediction

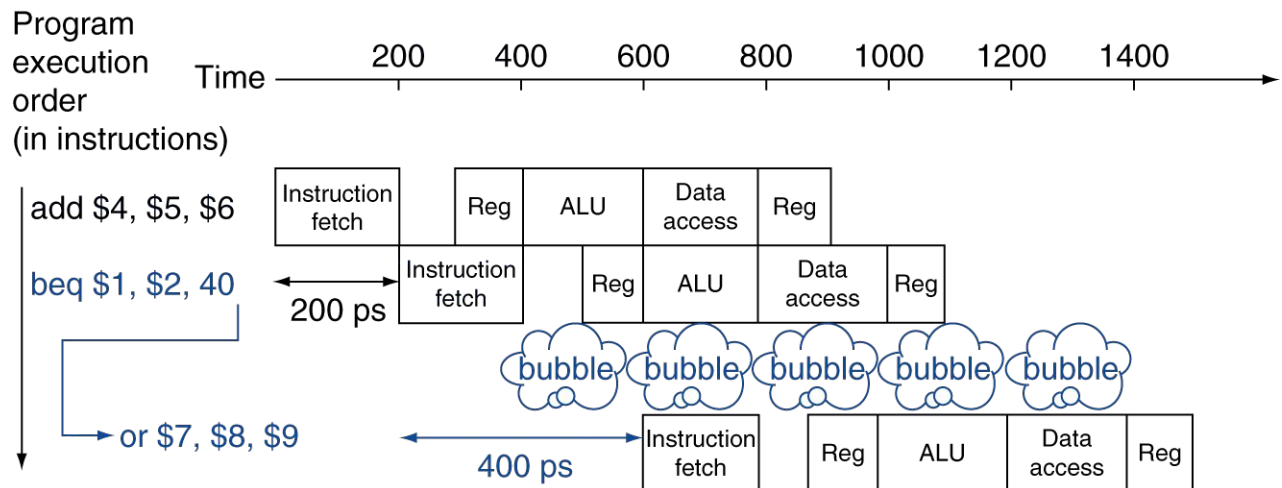
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# 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

# Pipelining and ISA Design

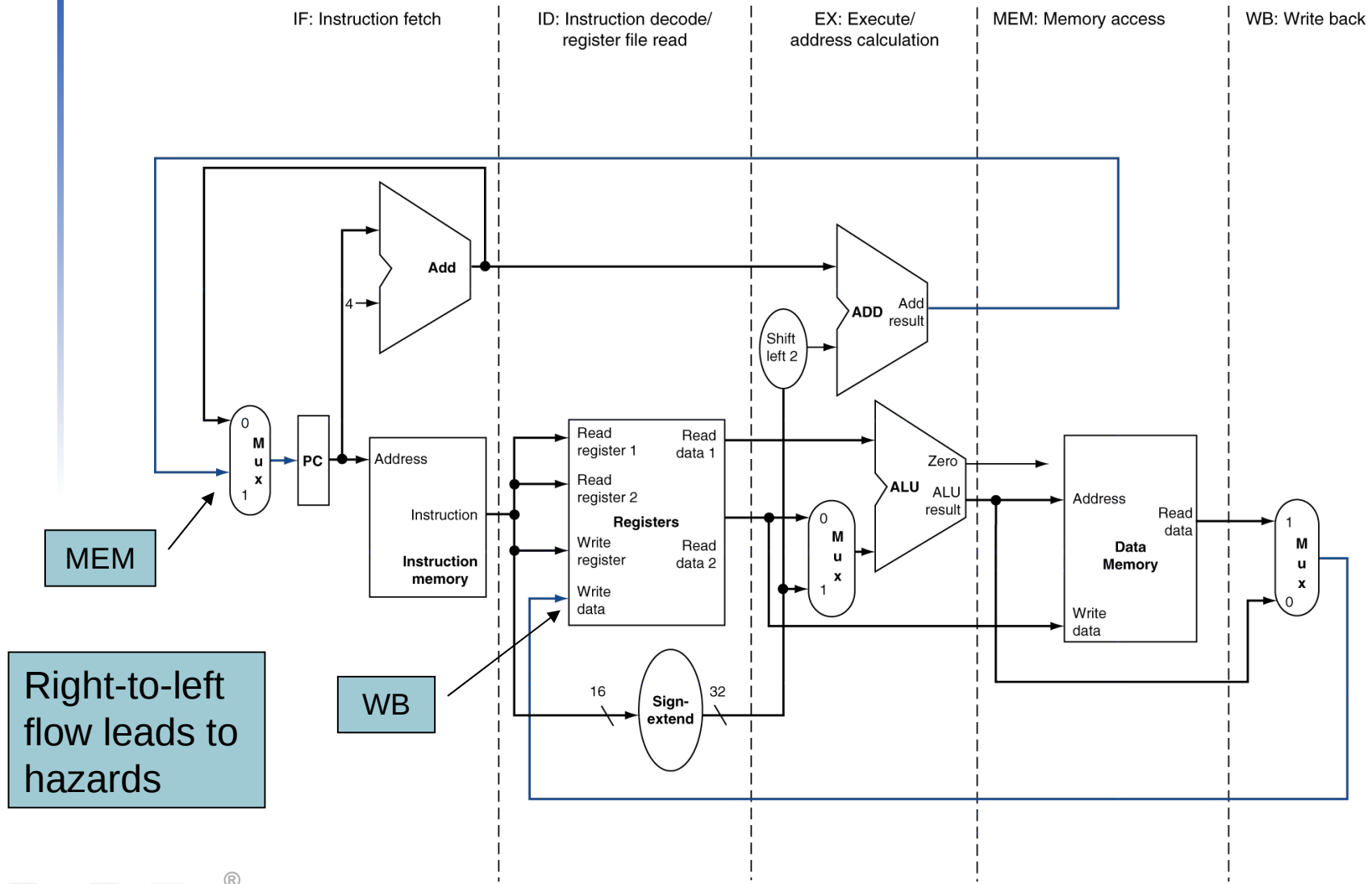
- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3<sup>rd</sup> stage, access memory in 4<sup>th</sup> stage
  - Alignment of memory operands
    - Memory access takes only one cycle

# 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

# MIPS Pipelined Datapath

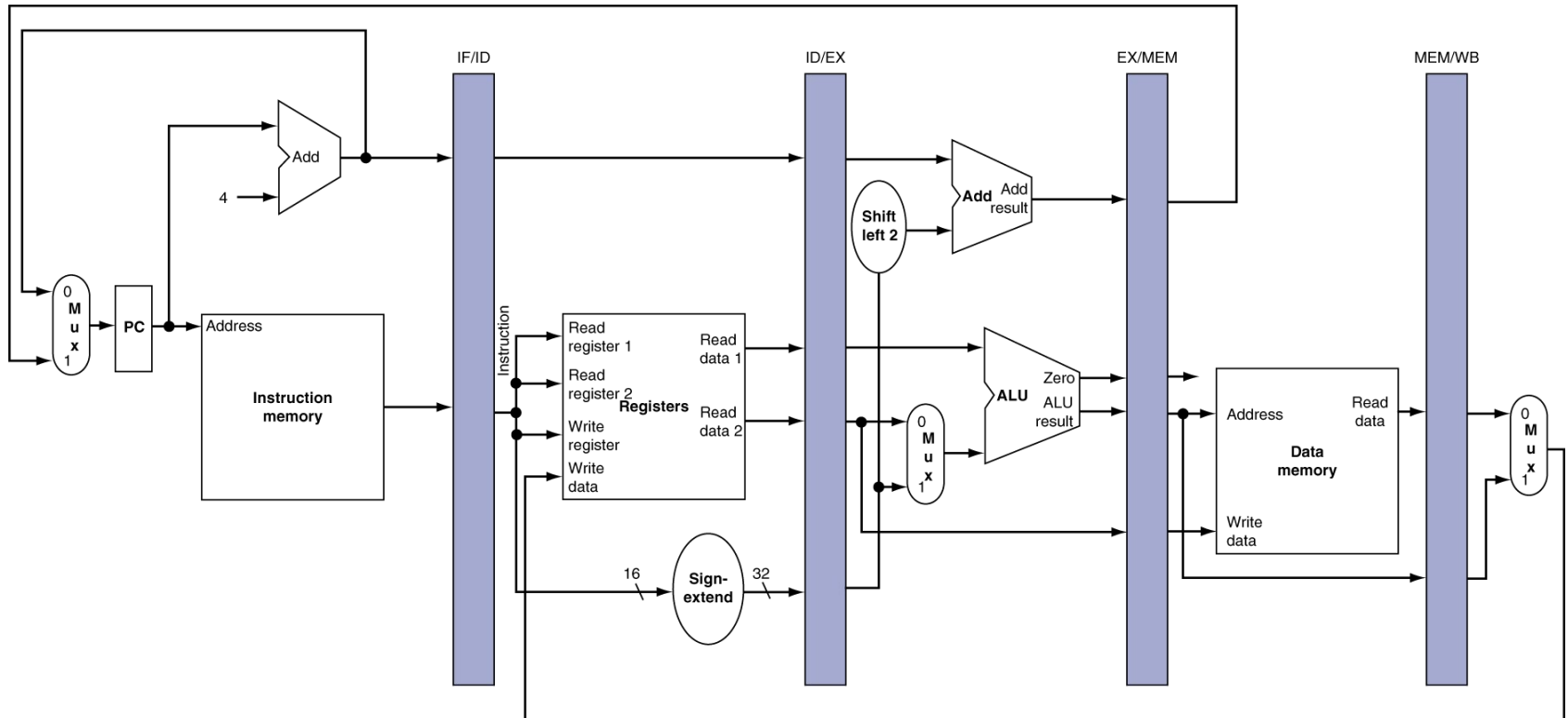


Right-to-left flow leads to hazards

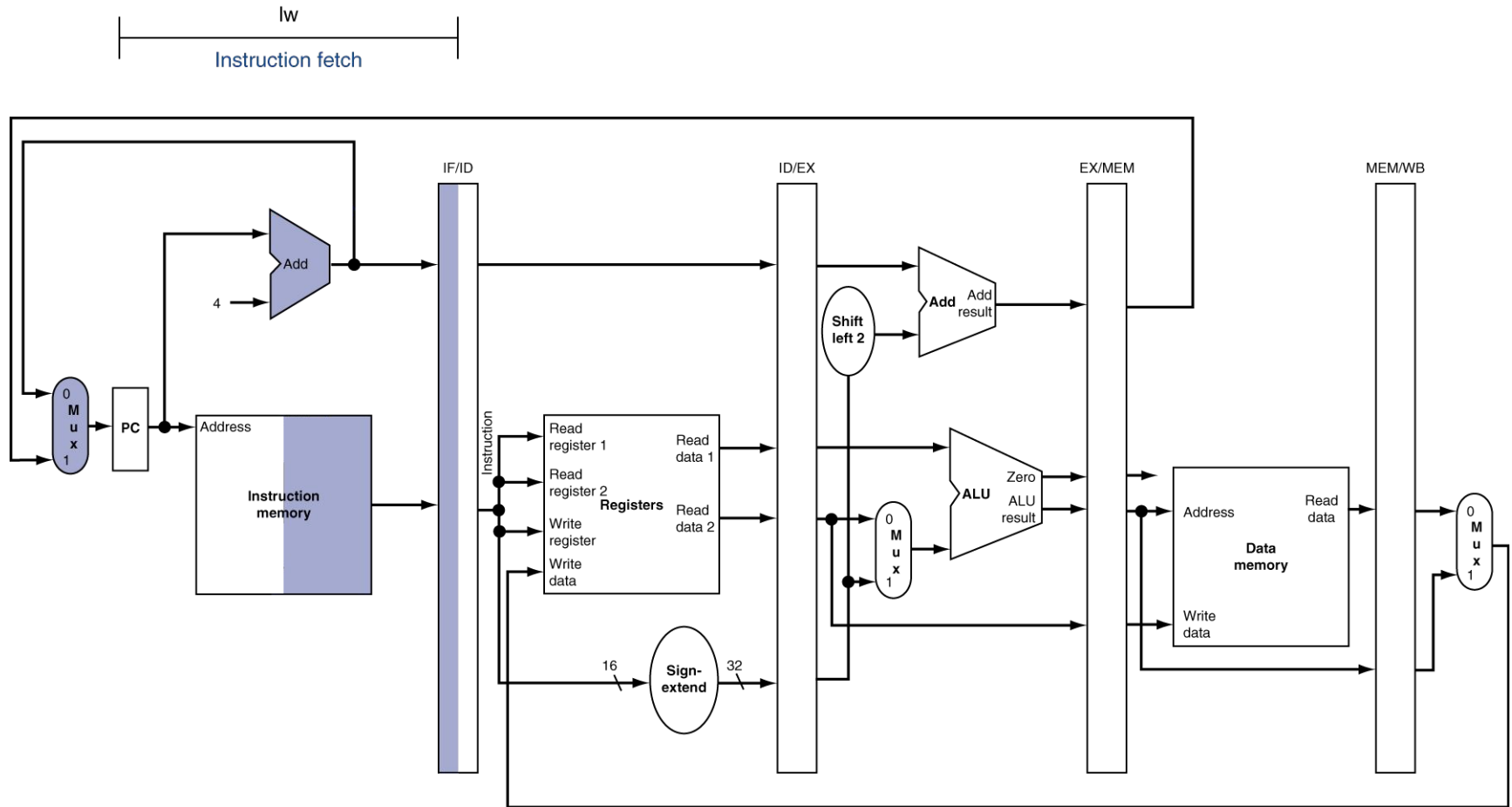


# Pipeline registers

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

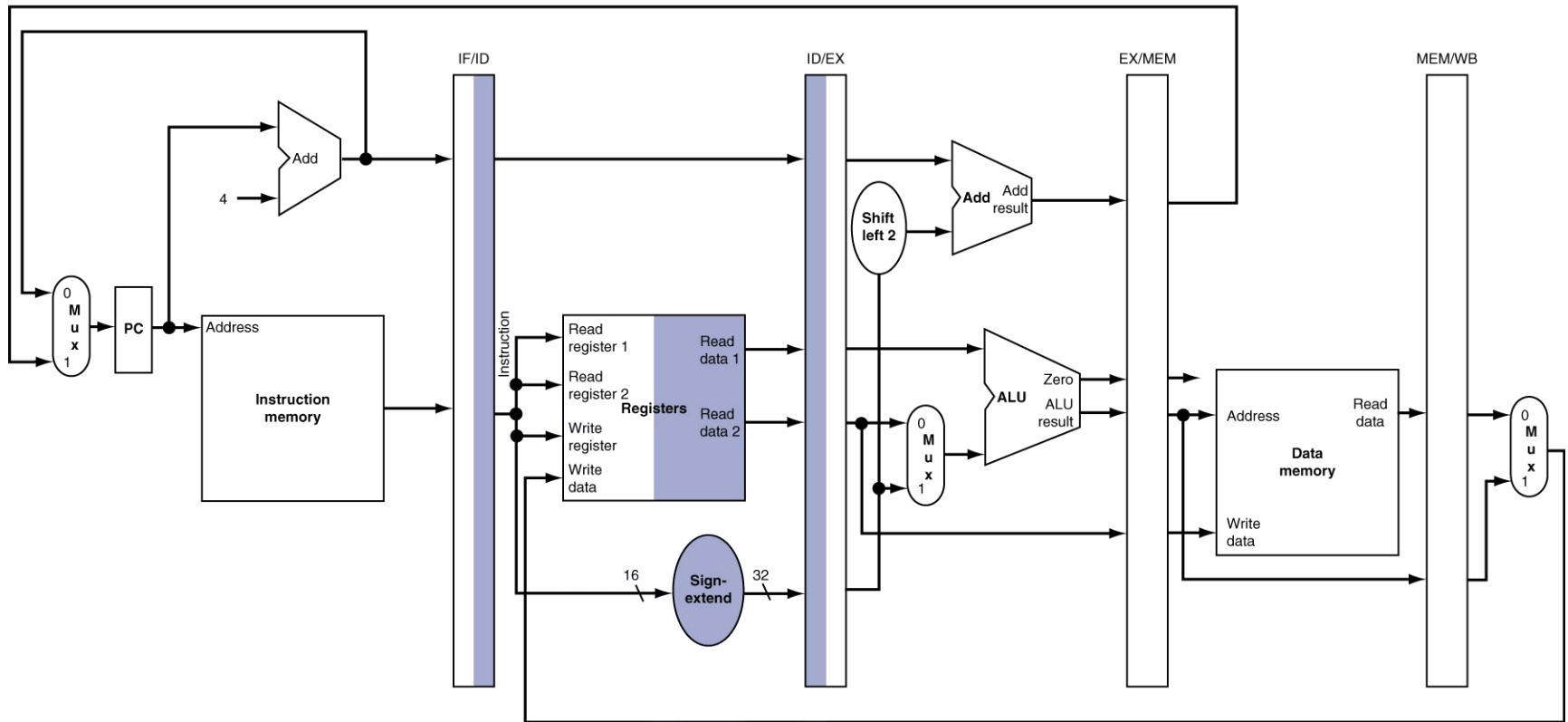


# IF for Load, Store, ...

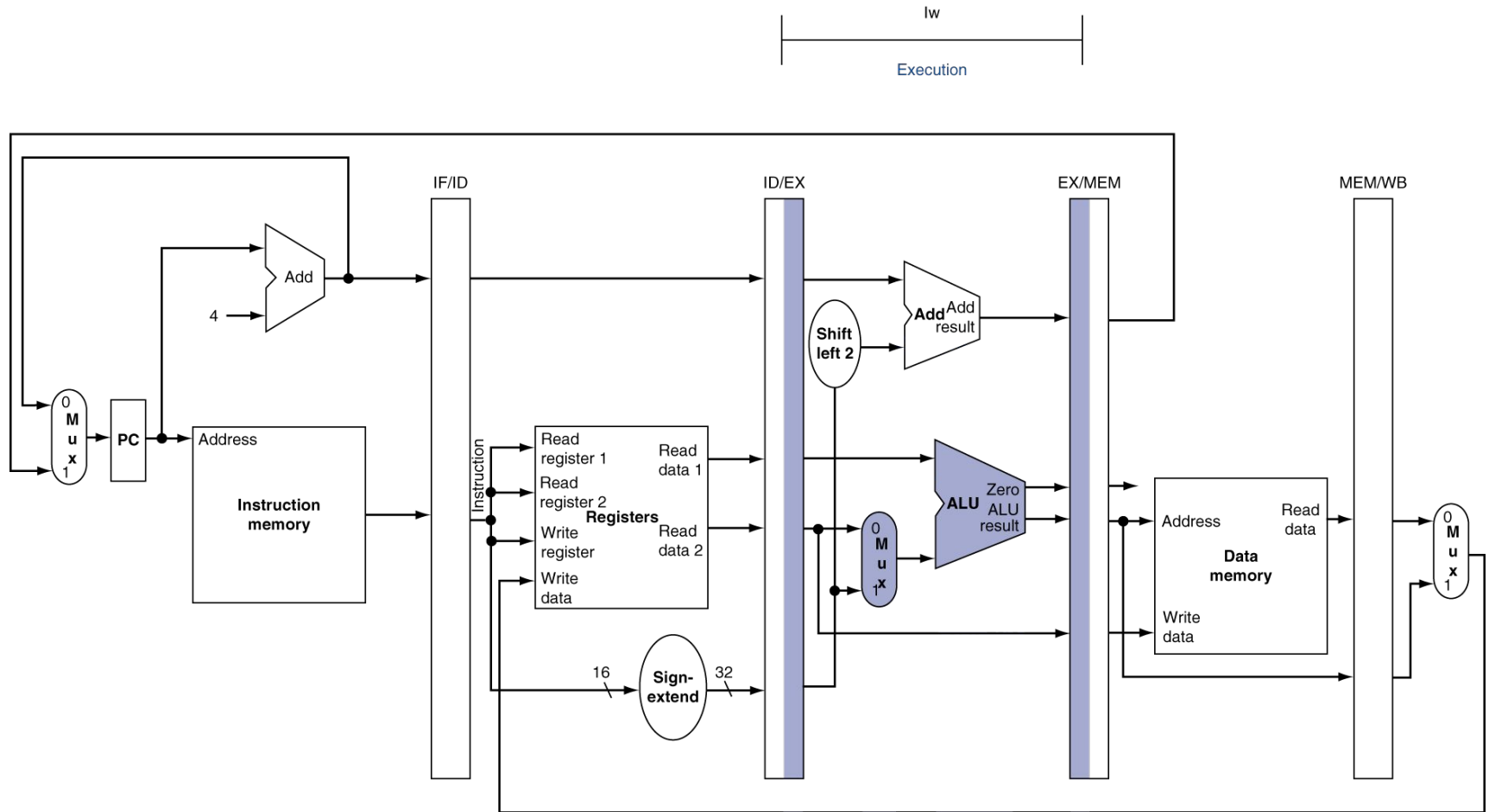


# ID for Load, Store, ...

lw  
Instruction decode

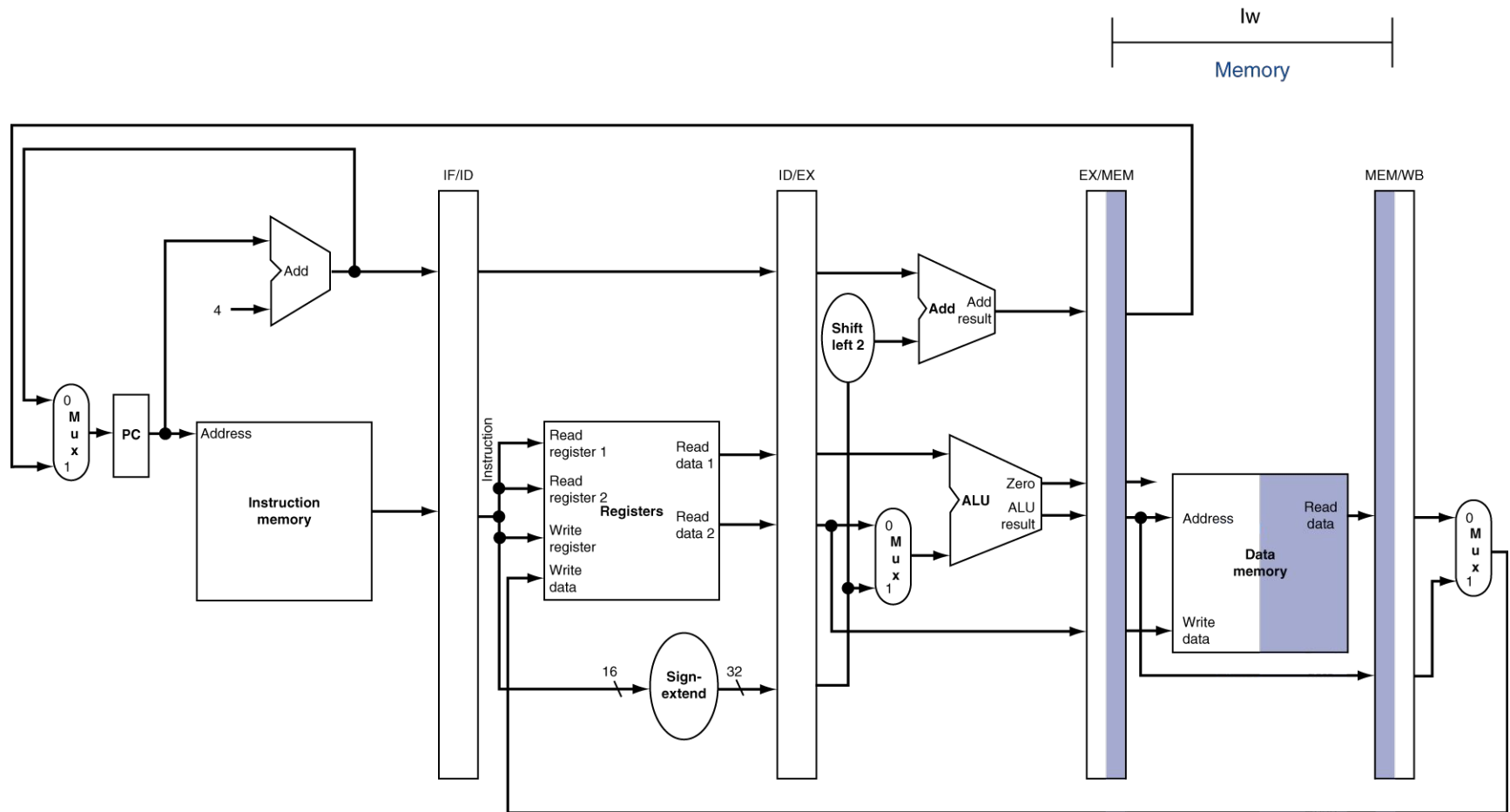


# EX for Load

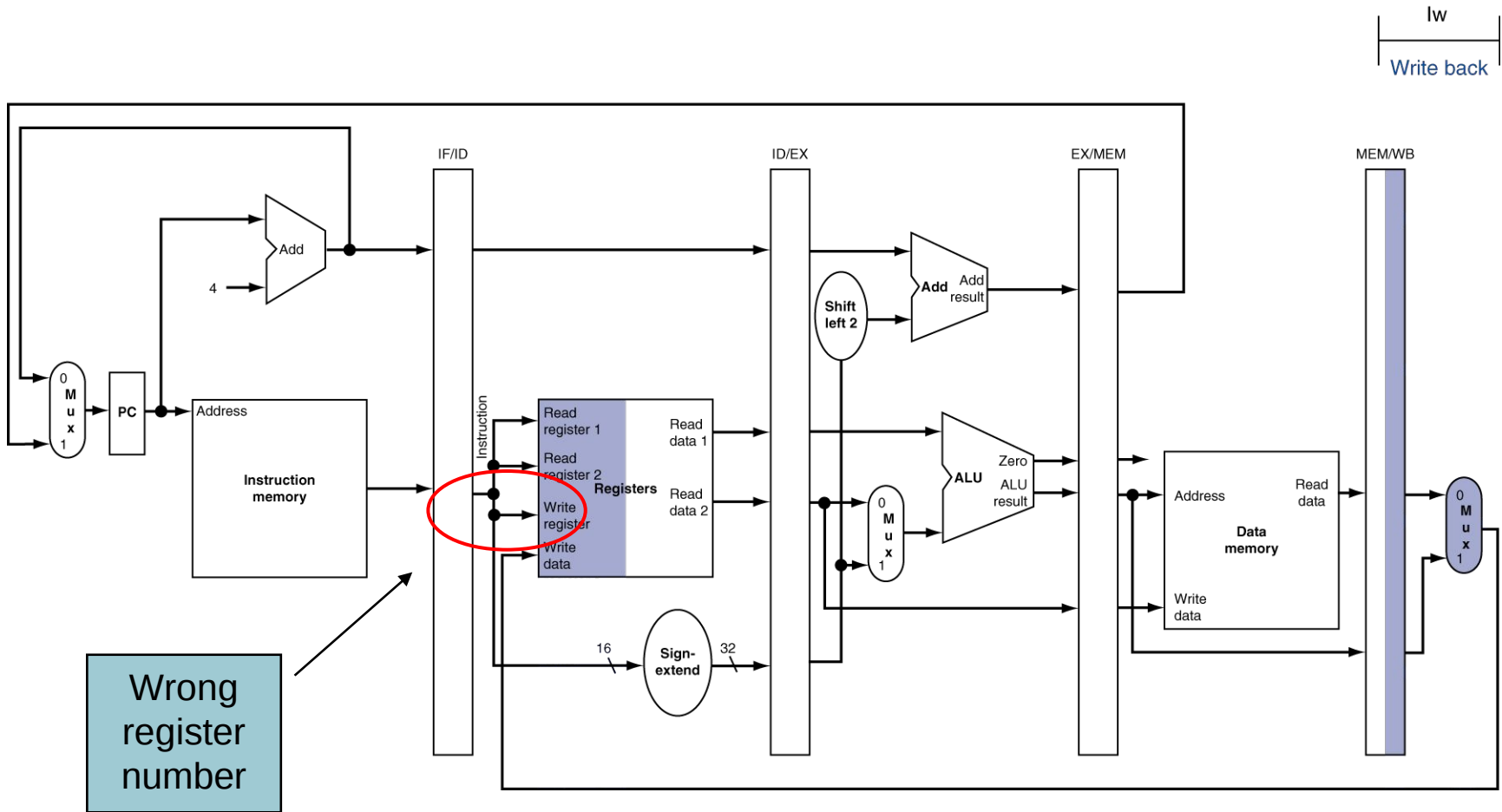




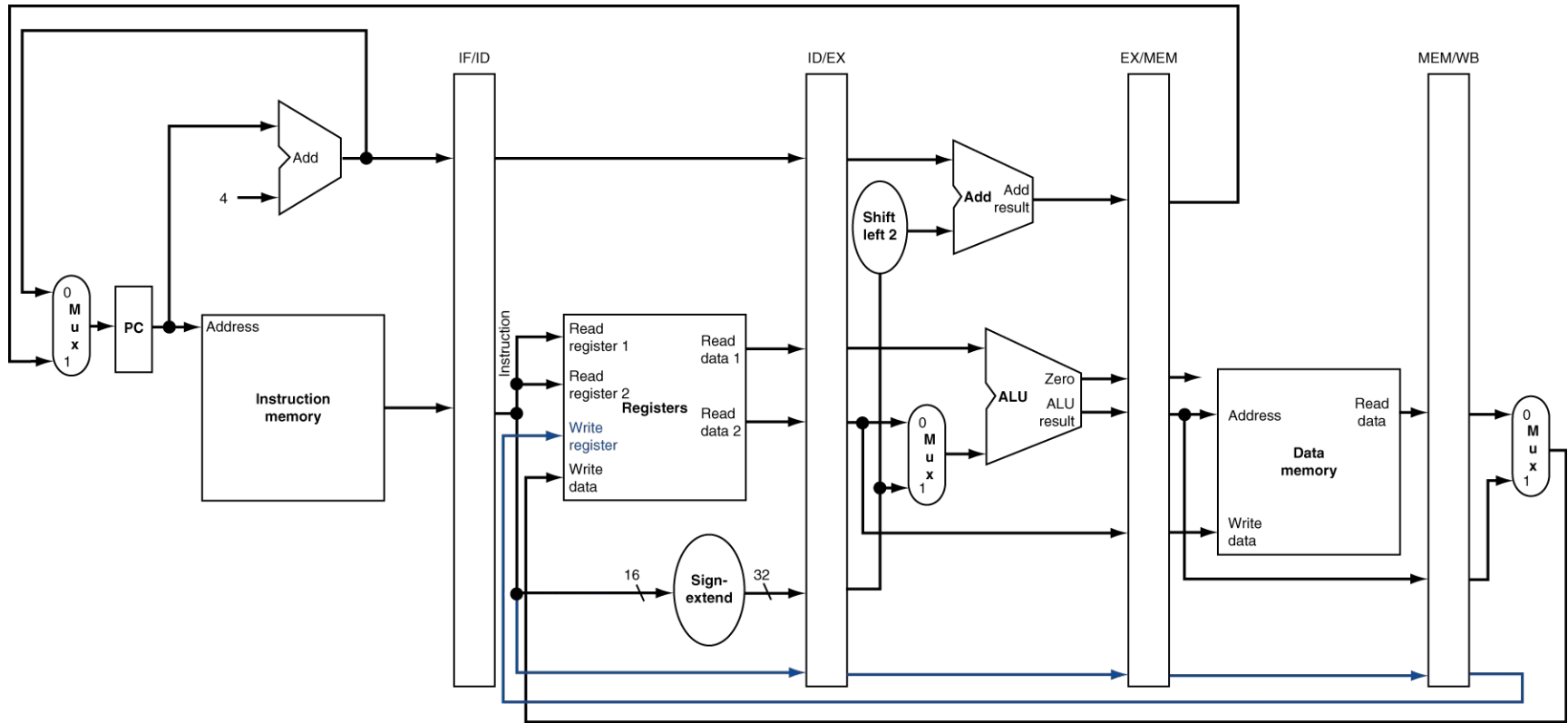
# MEM for Load



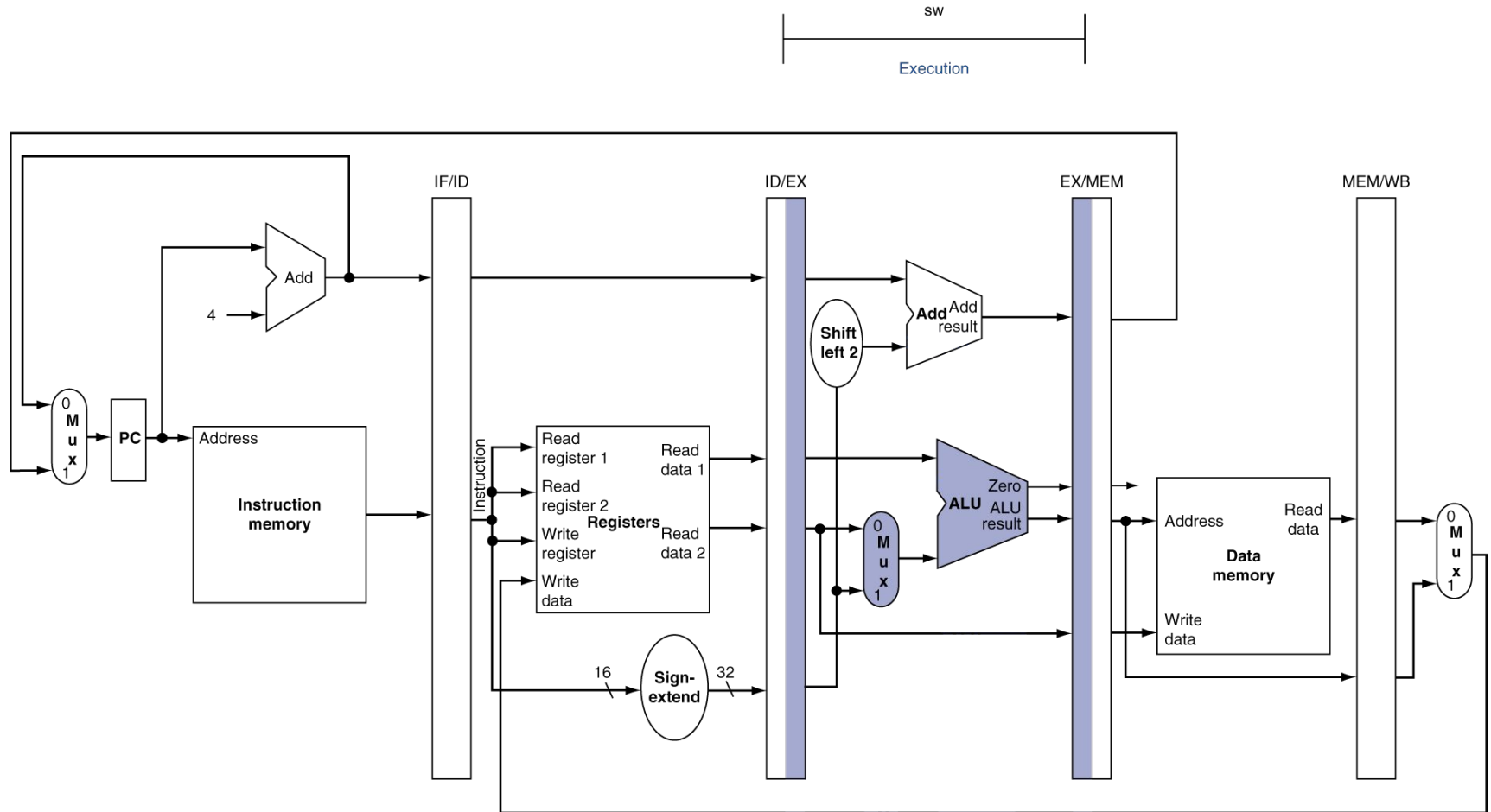
# WB for Load



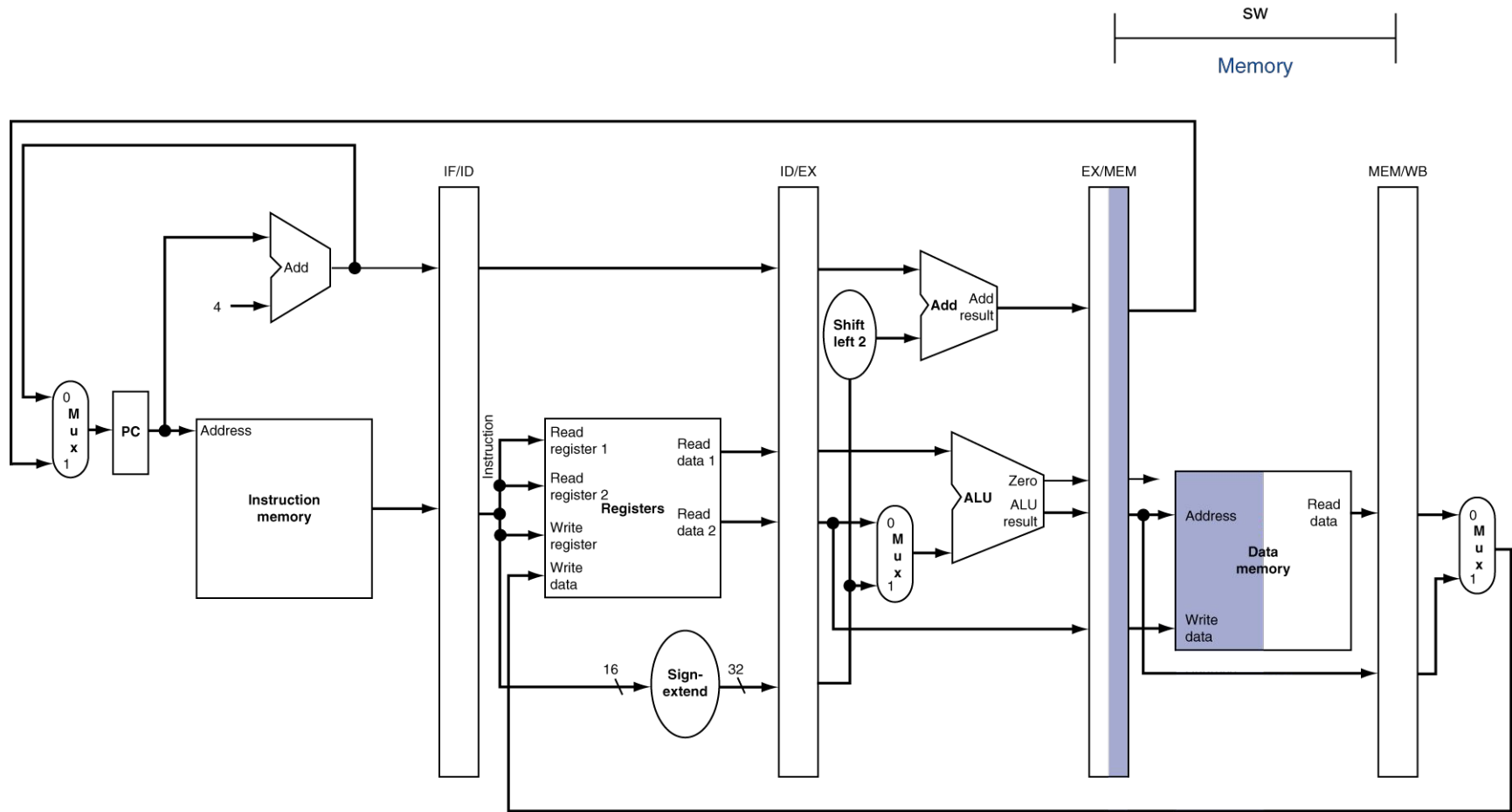
# Corrected Datapath for Load



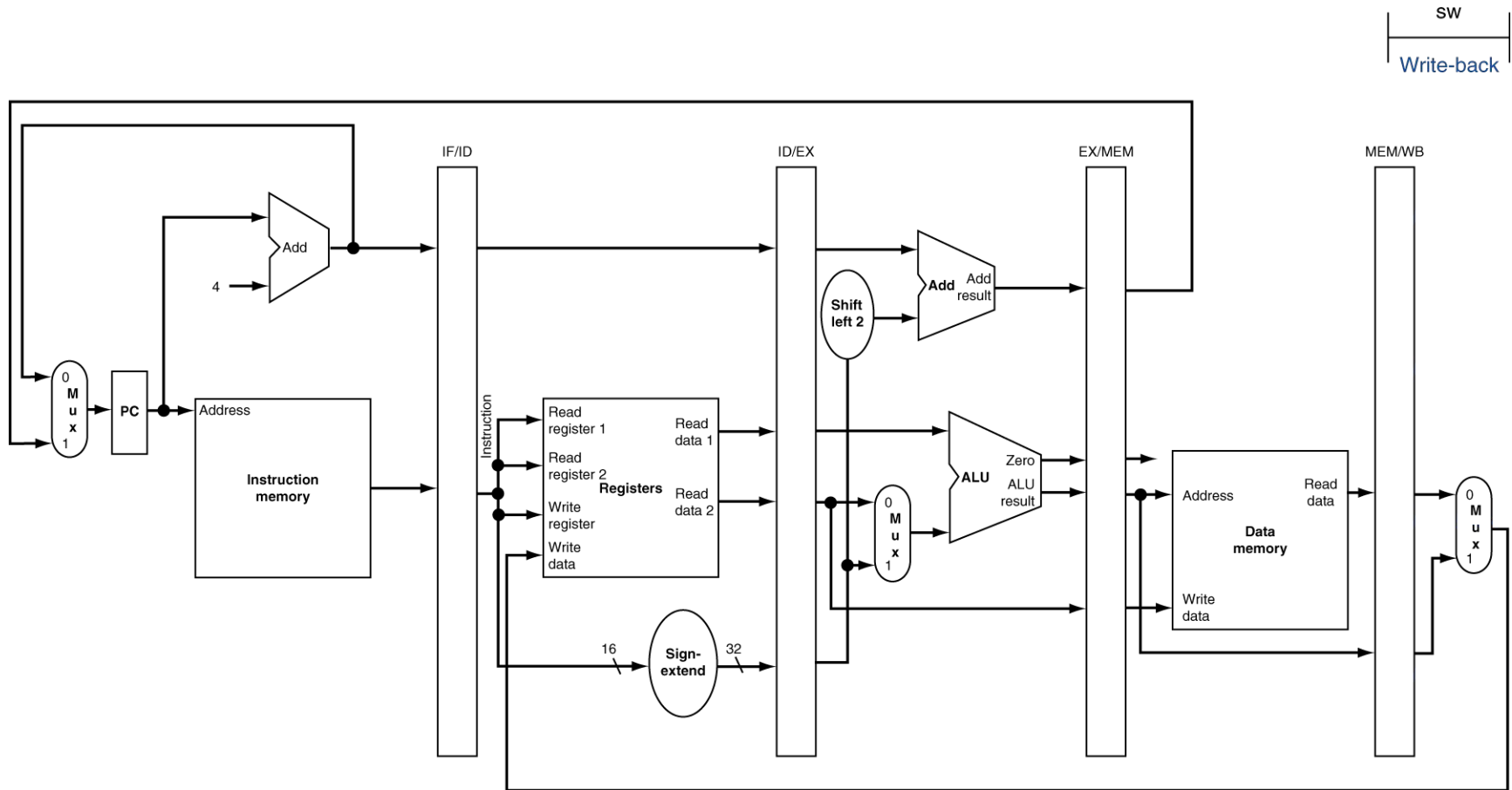
# EX for Store



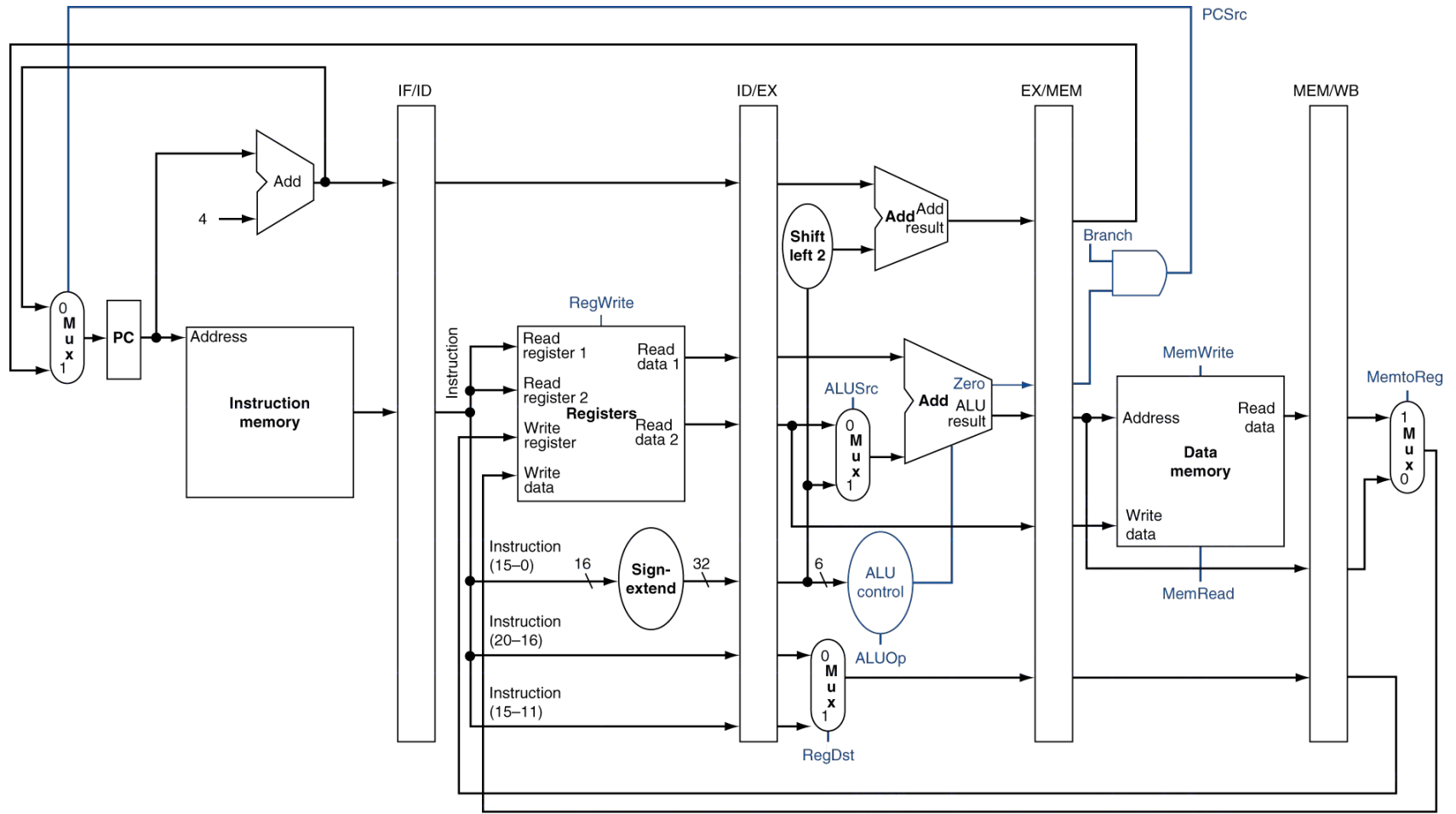
# MEM for Store



# WB for Store



# Pipelined Control (Simplified)



# Pipelined Control

