

# CSCI 210: Computer Architecture

## Lecture 28: Pipelining

Stephen Checkoway

Slides from Cynthia Taylor

# CS History: The IBM 7030 Stretch



- First pipelined computer
  - Three stages: Fetch-Decode-Execute
- Fastest computer in the world from 1961 until 1964
- Much slower than IBM anticipated
  - Dropped price from \$13.5 million to \$7.5 million
  - PC World named it one of the biggest IT project management failures in history
- Many ideas from the Stretch, including pipelining, were used in the very successful IBM System/360

# CS History: MIPS



- “Microprocessor without Interlocked Pipelined Stages”
- Started as a class project in graduate class taught by John Hennessy at Stanford in 1981
- In 1984 Hennessy formed MIPS Computer Systems to sell it commercially
- Emphasized short clock cycle and use of pipelining

# Instruction Critical Paths

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type						
load						
store						
beq						
jump						

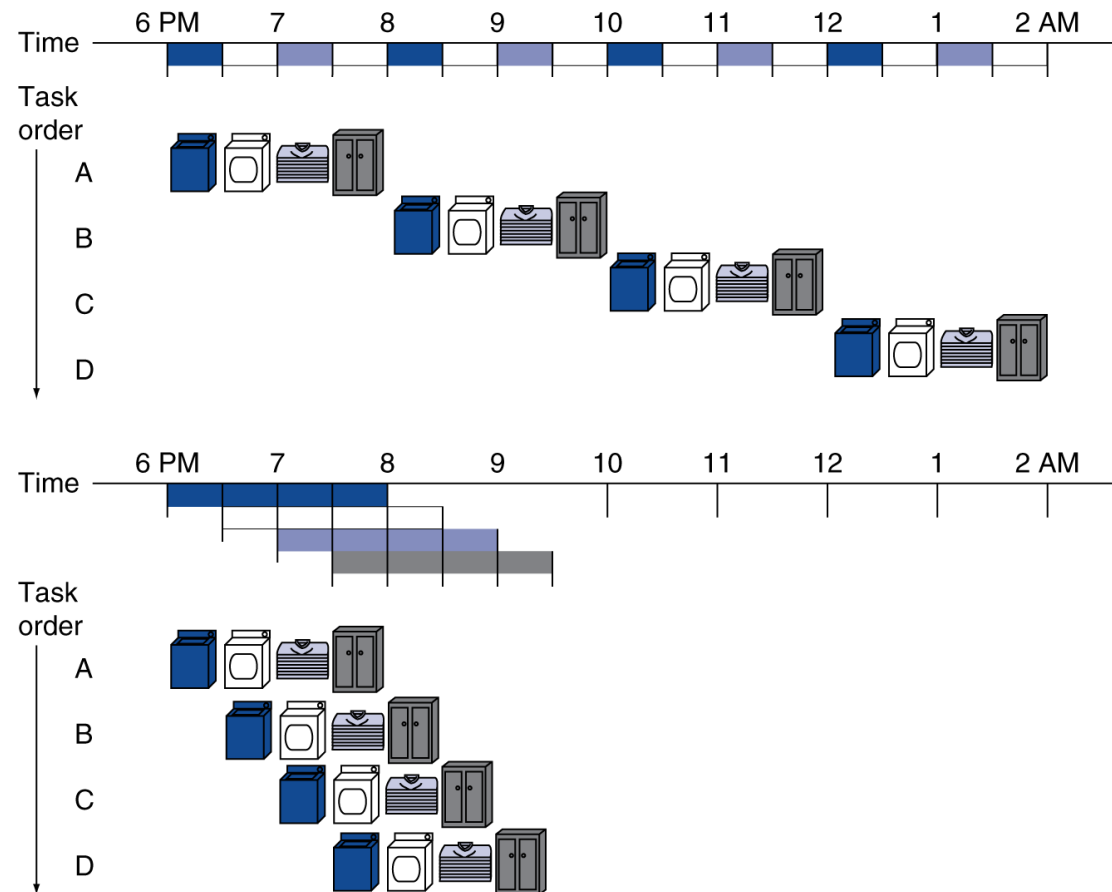
Times are just examples and not representative of real-world latencies

# 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

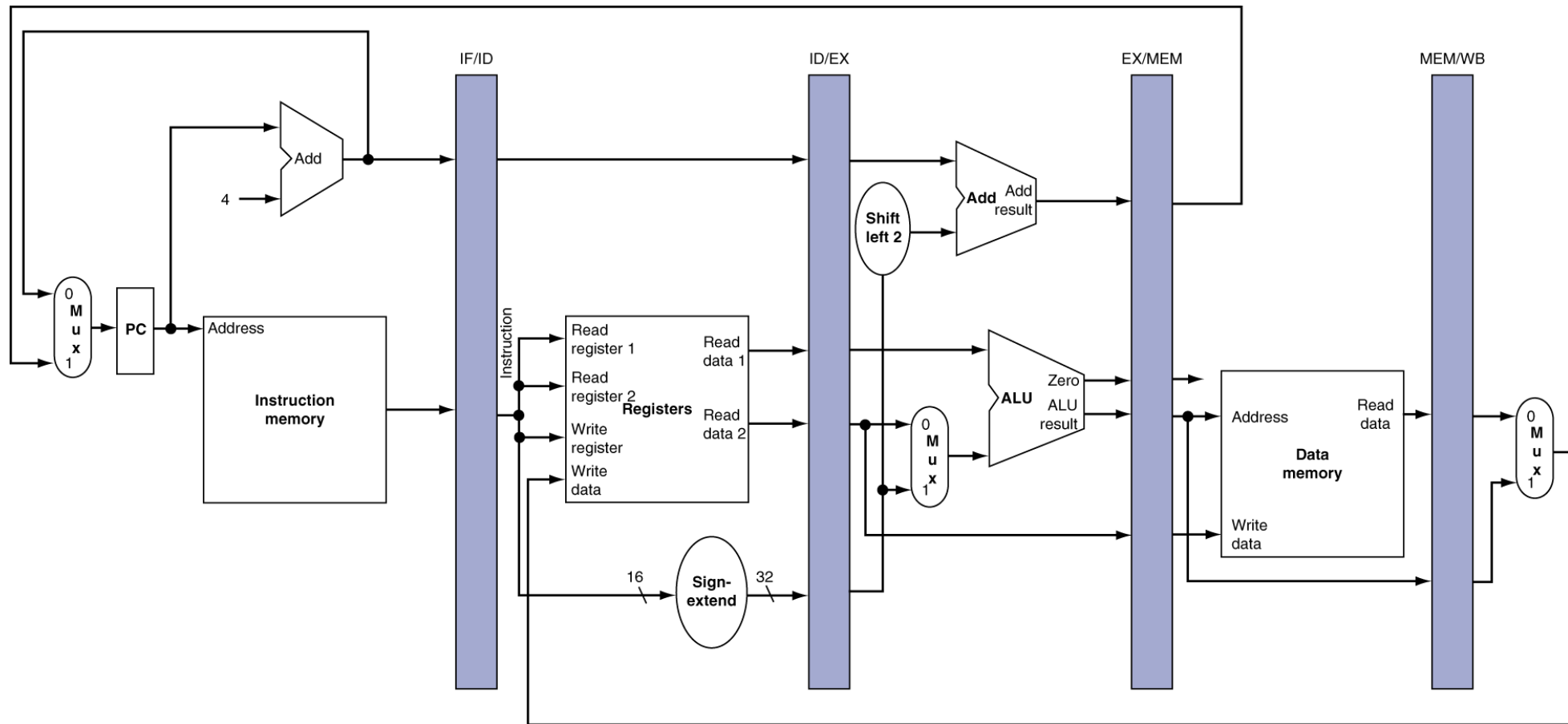
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



# 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
- Move from one instruction per clock cycle, to one stage per clock cycle (with shorter cycle period!)

# Pipelined Datapath



IF: Instruction Fetch

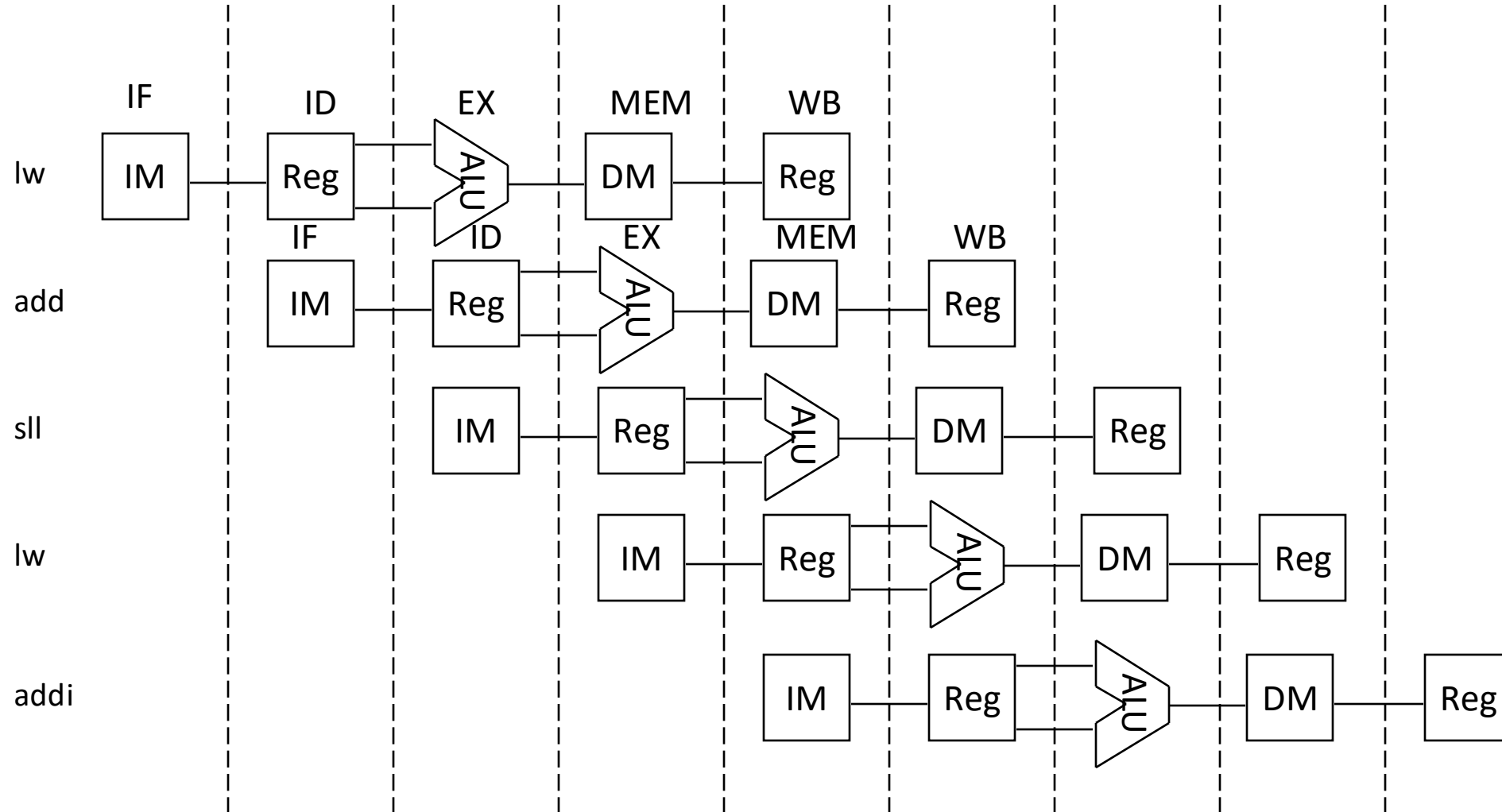
ID: Instruction Decode

EX: Execute

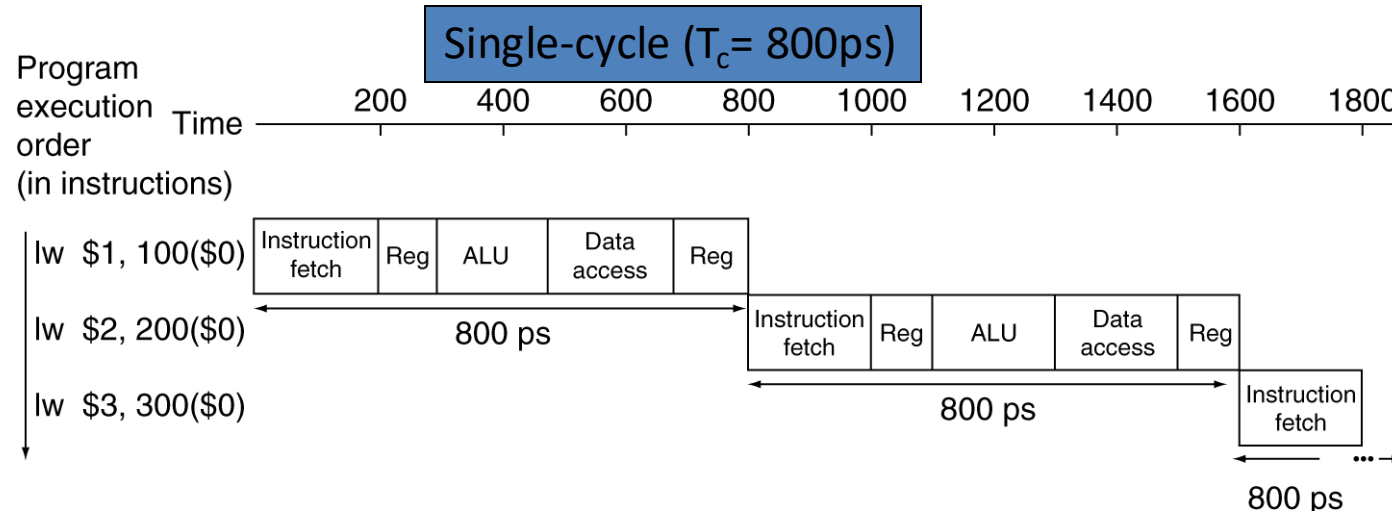
MEM: Memory

WB: Writeback

# Execution in a Pipelined Datapath



# Pipeline Performance

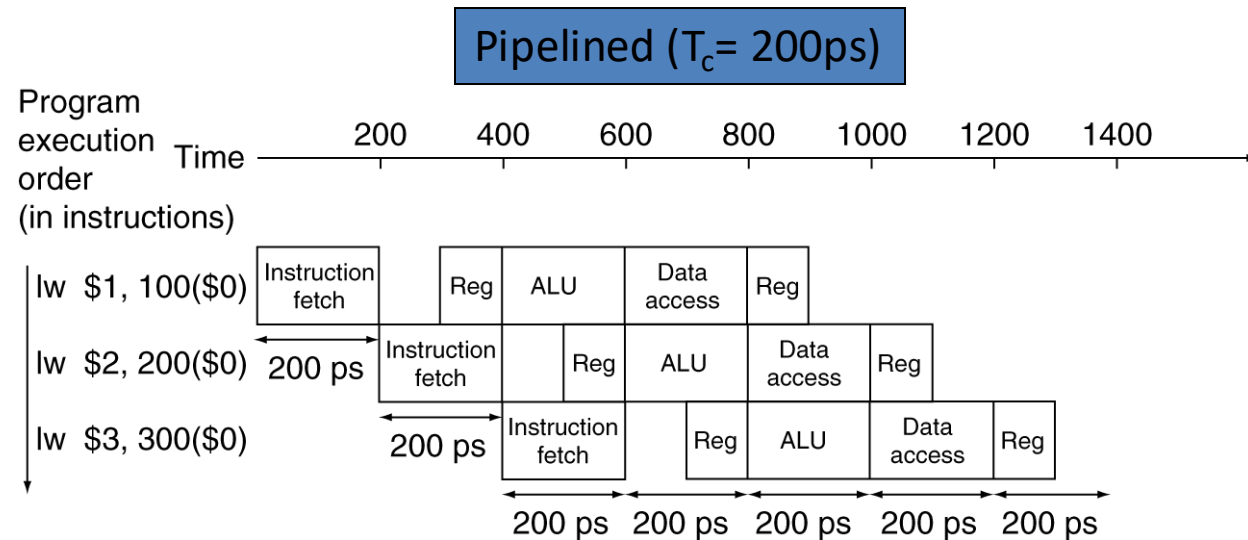
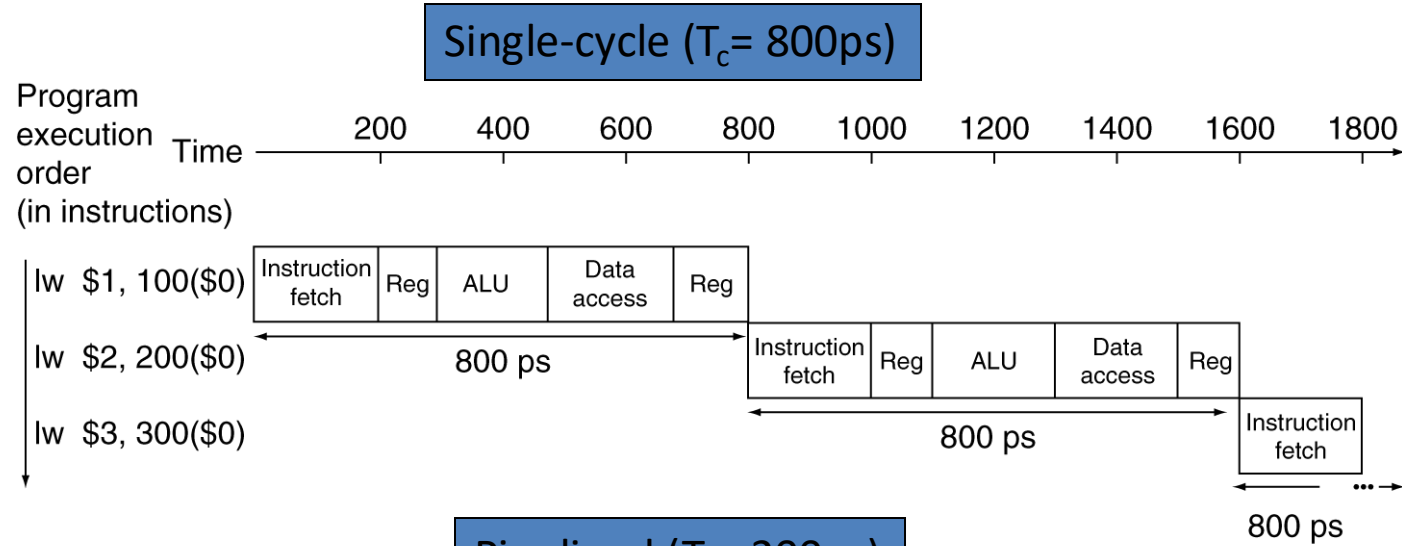


If we pipeline by running different stages at the same time (i.e., running instruction fetch for the next instruction during the Reg stage of the first instruction), running two instructions will take us

- A. 900 ps
- B. 1000 ps
- C. 1200 ps
- D. 1600 ps

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- HINT: How long will the clock cycle be in the pipelined version?

# Pipeline Performance



In our pipelined data path, latency ( $\mu\text{s}$  per instruction)  
\_\_\_\_\_, but throughput (instructions per second)  
\_\_\_\_\_

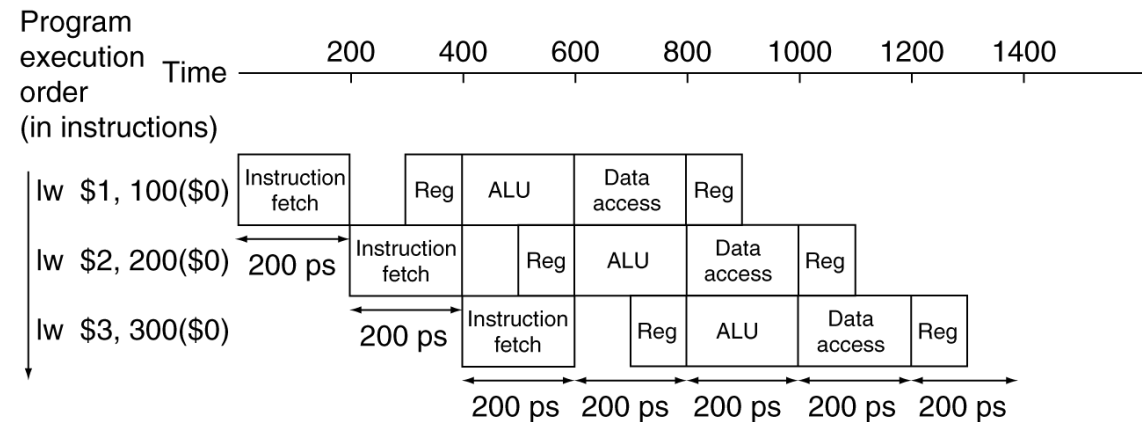
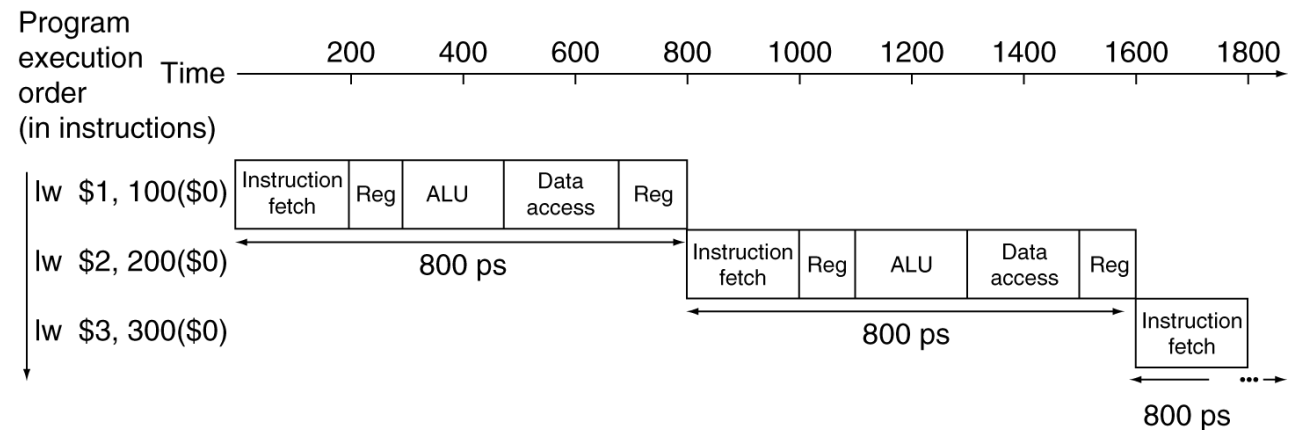
A. Improves, gets worse

B. Improves, stays the same

C. Gets worse, improves

D. Stays the same, improves

E. None of the above



# Maximum Pipeline Speedup

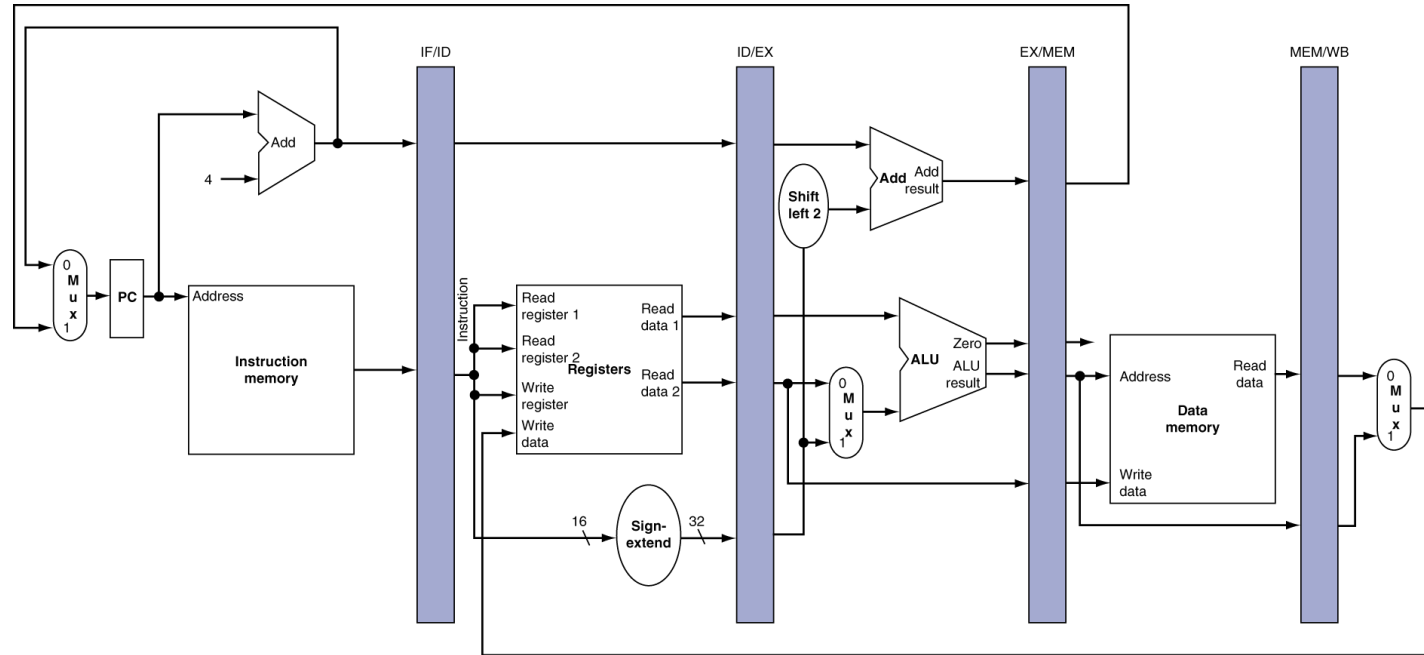
- If all stages are balanced
  - i.e., all take the same time

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- $\text{Speedup} = \text{time}_{\text{pipelined}} / \text{time}_{\text{nonpipelined}} = \text{number of stages}$
- 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 in one cycle
    - cf. x86: 1- to 15-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one cycle
  - Load/store addressing
    - Can calculate address in 3<sup>rd</sup> stage (EX), access memory in 4<sup>th</sup> stage (MEM)



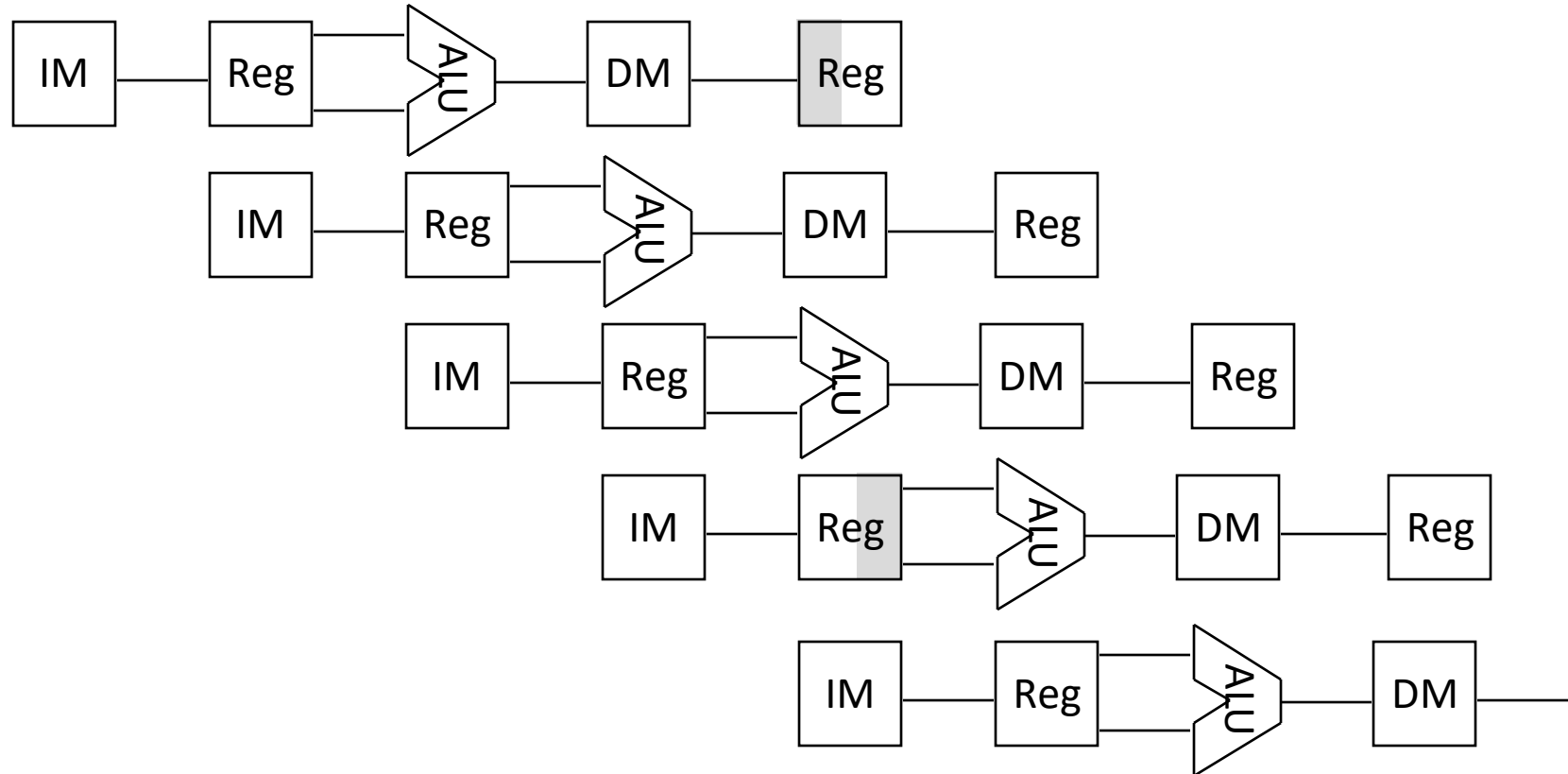
sub **\$2**, \$1, \$3

and \$12, **\$2**, \$5

or \$13, \$6, **\$2**

add \$14, **\$2**, **\$2**

sw \$15, 100(**\$2**)



What just happened here which is problematic (BEST ANSWER)?

- A. The register file is trying to read and write the same register
- B. The ALU and data memory are both active in the same cycle
- C. A value is used before it is produced
- D. Both A and B
- E. Both A and C

# Hazards

Situations that (if not handled) prevent starting the next instruction in the next cycle

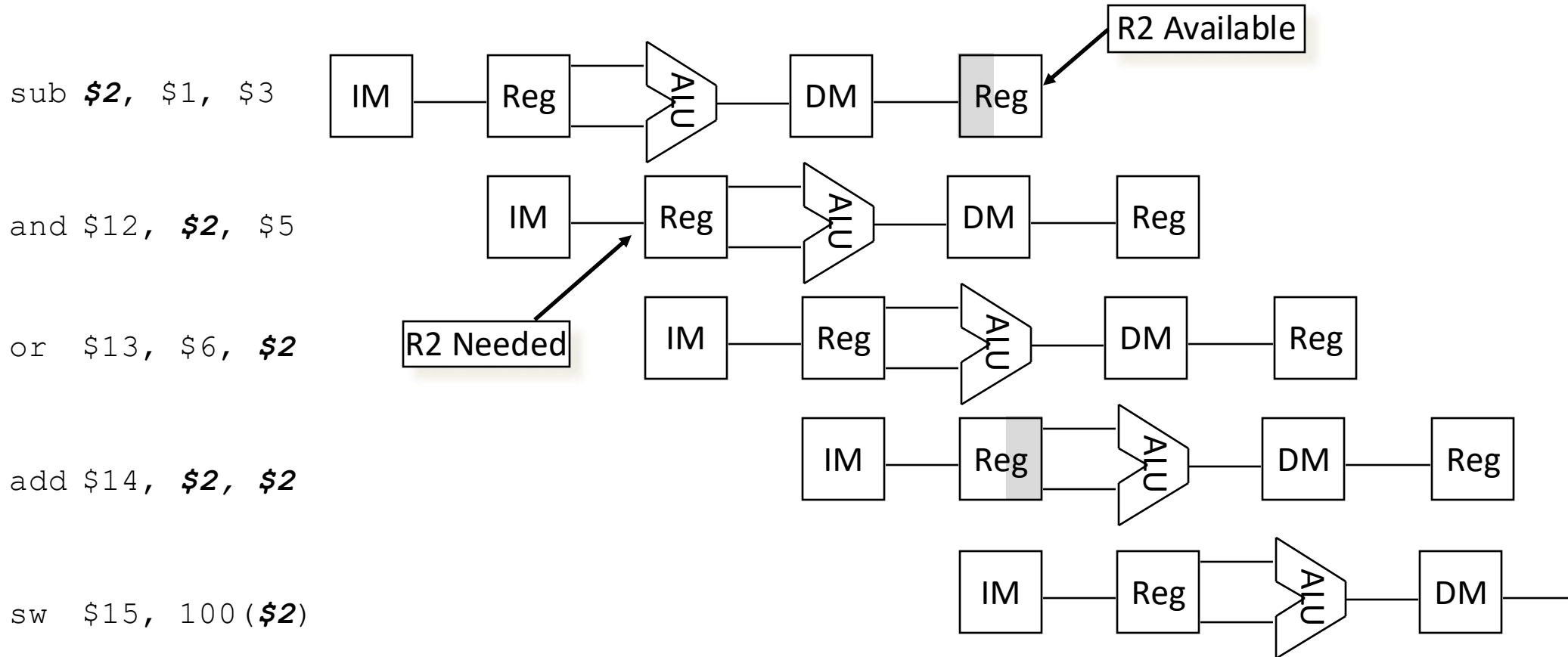
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

# 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 data paths require separate instruction/data memories (or “caches” which we’ll talk about later in great detail)

# Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

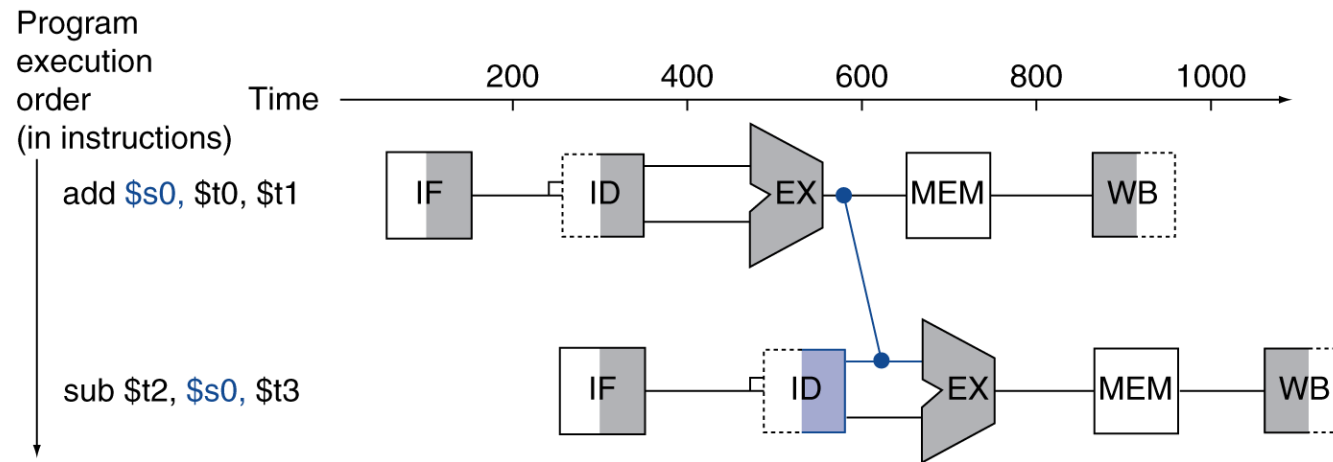


# We could solve data hazards by

- A. Reordering instructions
- B. Not running the second instruction until the data is ready
- C. Sending the calculated value straight from the ALU to the next instruction, skipping the registers
- D. More than one of the above

# Forwarding (a.k.a. Bypassing)

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



# Would forwarding work if our instructions were

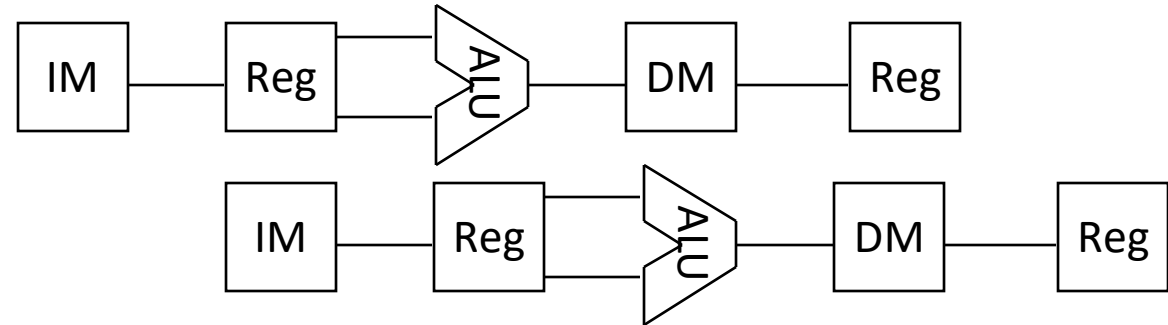
```
lw    $s0, 20($1)
```

```
sub   $t2, $s0, $t3
```

A. Yes

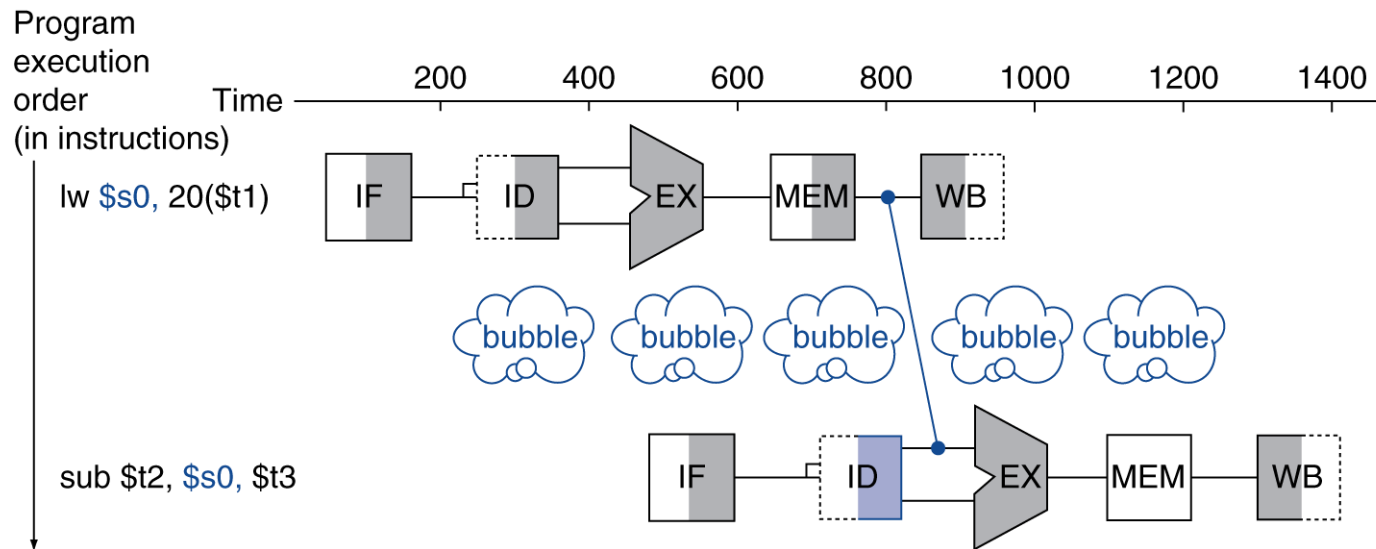
B. No

C. Depends on the value loaded



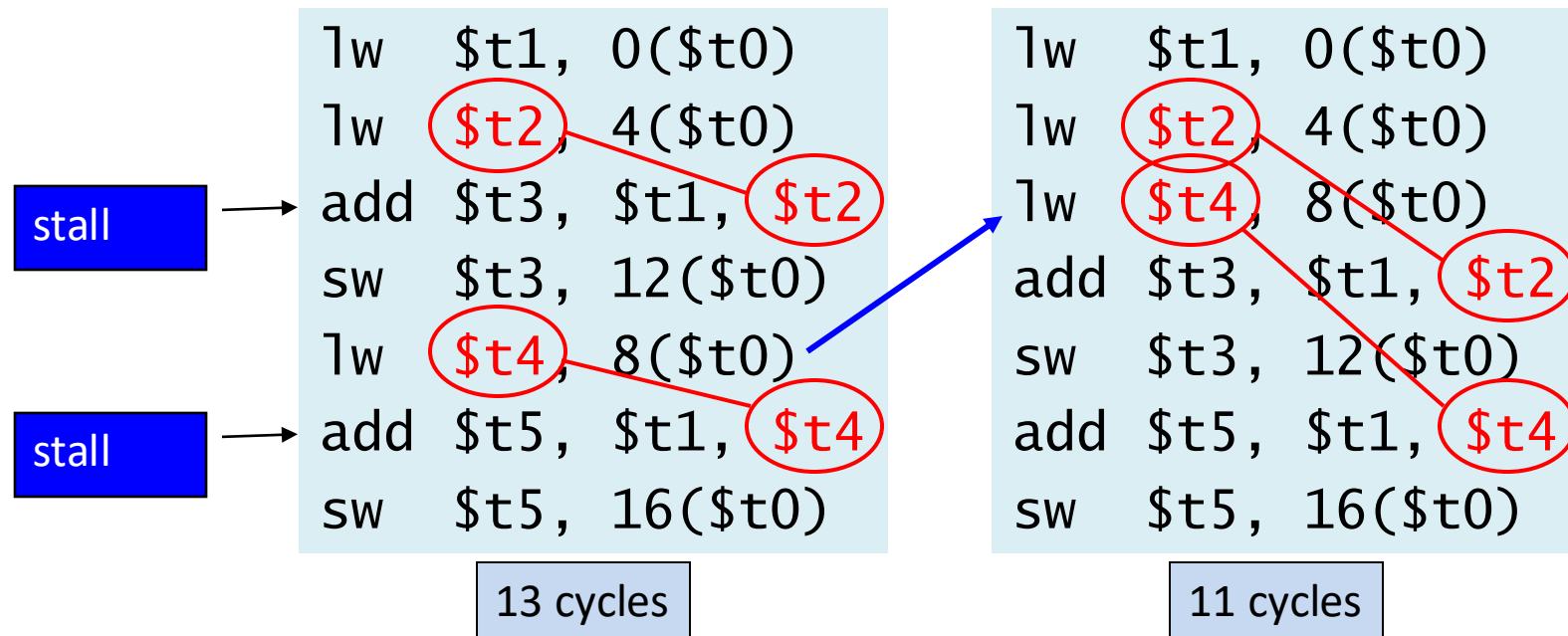
# Load-Use Data Hazard

- 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
- Assembly code for  $A = B + E$ ;  $C = B + F$ ;



# Questions on Data Hazards?

- Basic problem: Need a value before it's written to a register
- Solutions: Forwarding, stalling, reordering instructions (code scheduling)

# 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

# Control Hazards in MIPS

- In MIPS 5 stage pipeline
  - Branch outcome and branch target aren't determined until the EX stage
  - By this point, the following two instructions are in the pipeline (the instruction after the branch is in the ID stage, the instruction after that is in the IF stage)
- Partial solution
  - Compare registers and compute target earlier in the pipeline
  - Add circuitry to do it in ID stage (what do we need to compare register values and compute the branch target?)

We moved the branch instruction implementation  
as early as we possibly can

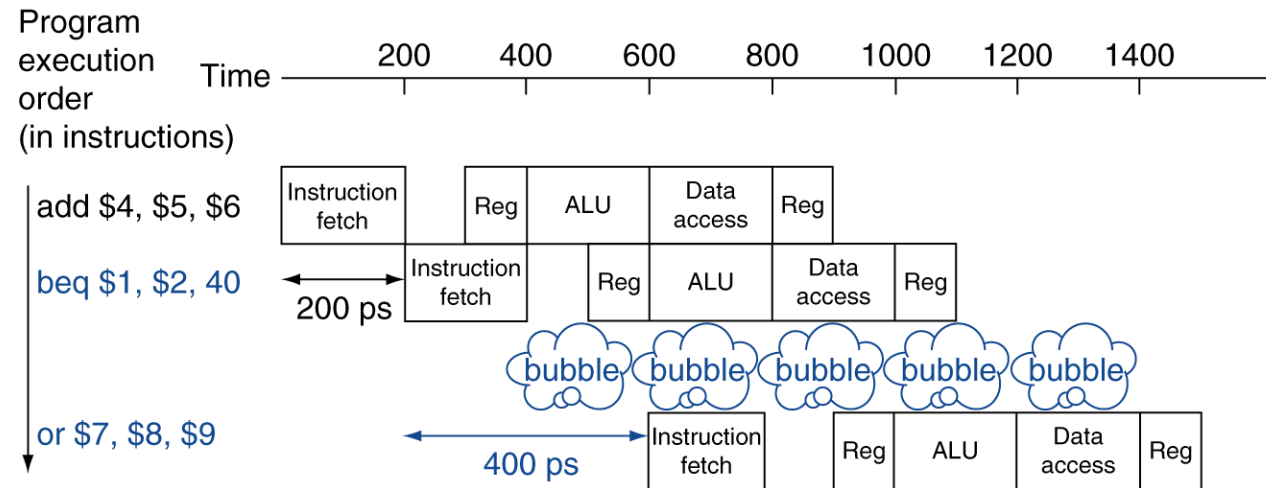
- We can't compute the branch outcome earlier than the ID stage because we need to read register values
- We still have a control hazard because we don't know what the next instruction to execute is

# To solve control hazards, we could

- A. Rearrange instructions.
- B. Guess what the branch will do and if we guess wrong, discard the incorrect instruction and fetch the correct one
- C. Just stall until we know the branch outcome.
- D. More than one of the above.

# Stall on Branch

Wait until branch outcome determined before fetching next instruction



# Problem

- Branches are about 15% of instructions
- If all instructions except branches have a CPI of 1, and branches have a CPI of 2, we have an average CPI of 1.15
- Even worse if we can't resolve the branch until later

# Branch Prediction

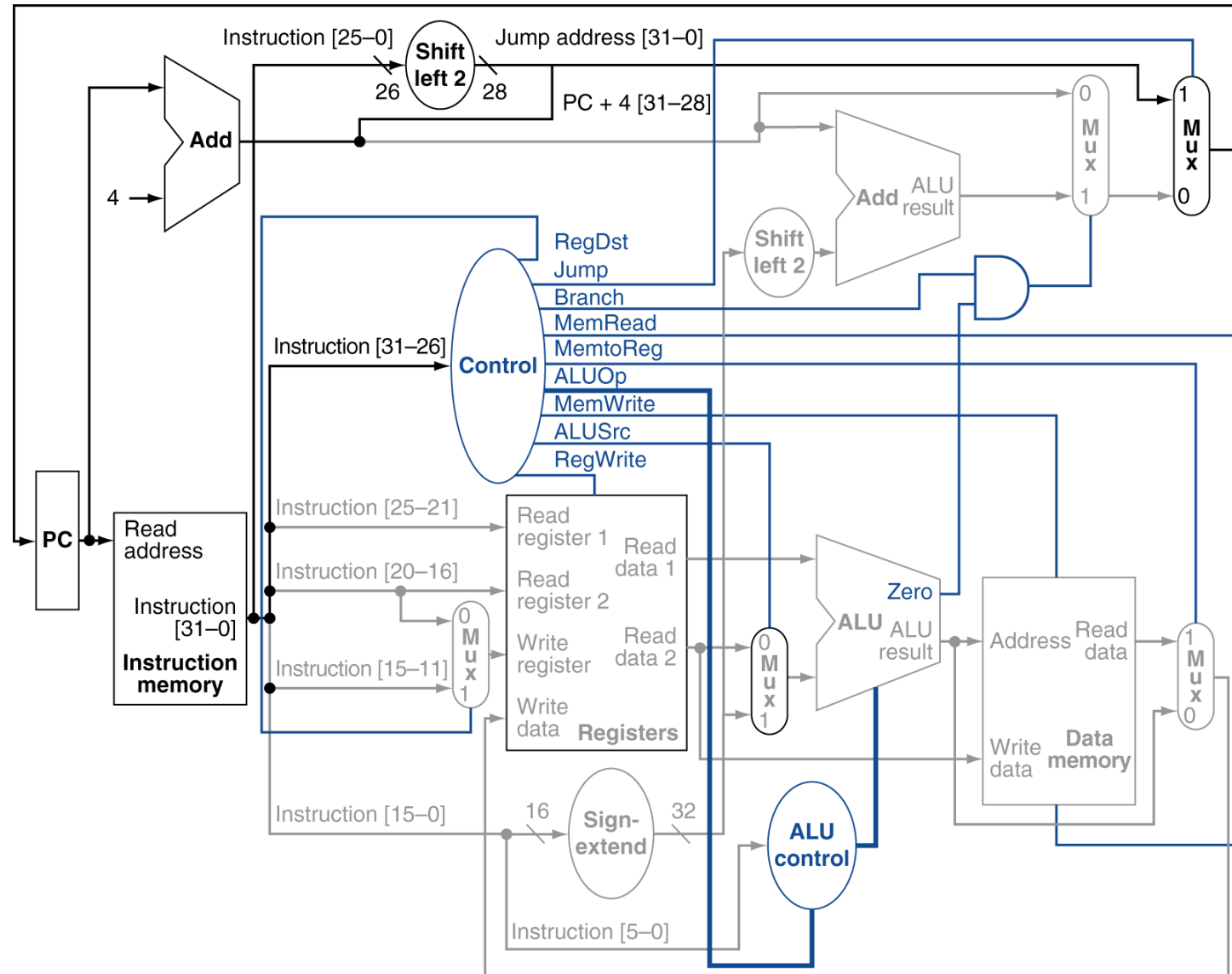
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - If prediction is wrong, then “flush” the pipeline
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# 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

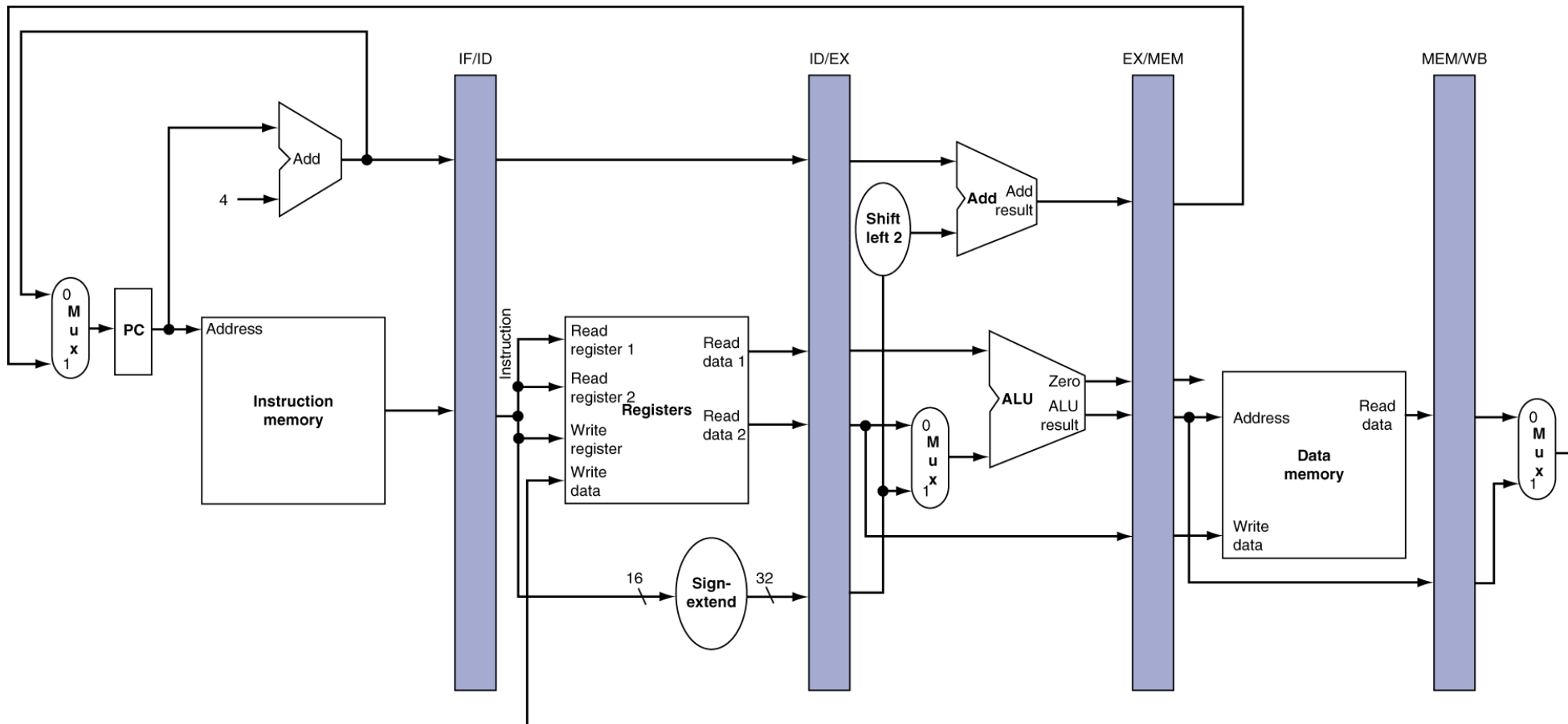
# Questions on Control Hazards?

# Single Cycle Data Path

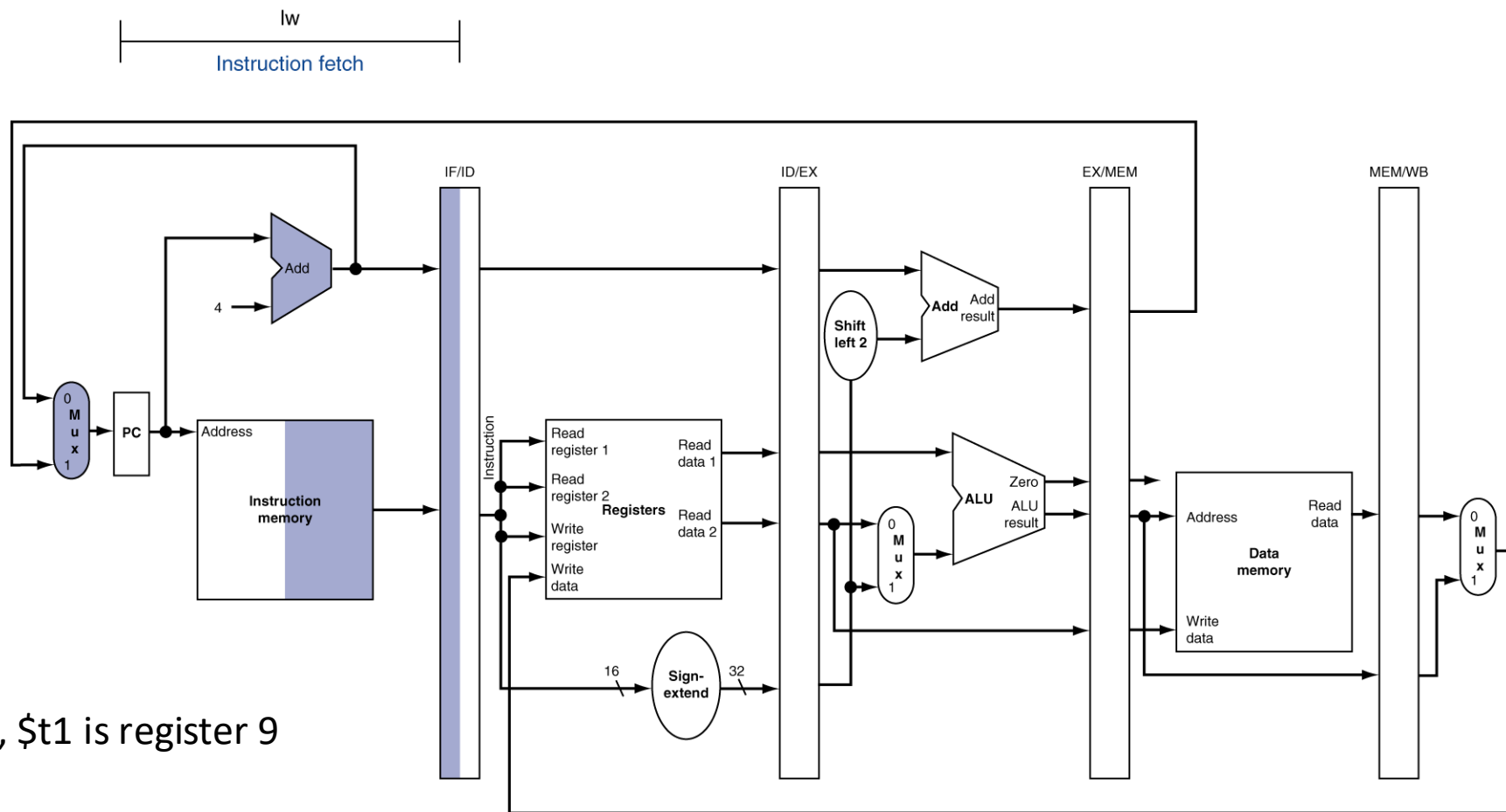


# Pipeline registers

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



# IF for lw \$t0, 4(\$t1)



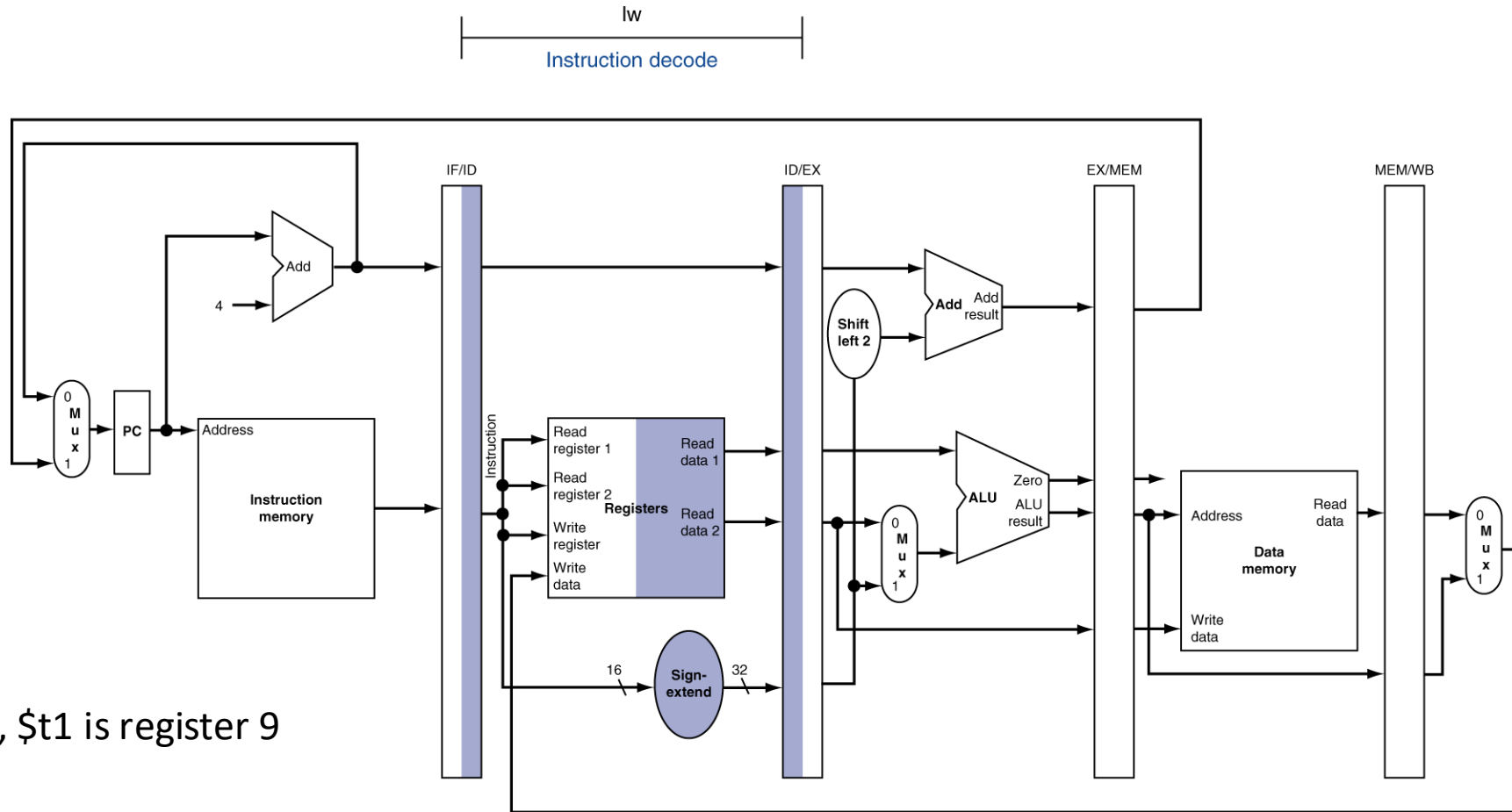
\$t0 is register 8, \$t1 is register 9

\$t0 holds 5

\$t1 holds 0x4810CAB0

0x4810CAB0 holds 12

# ID for lw \$t0, 4(\$t1)



\$t0 is register 8, \$t1 is register 9

\$t0 holds 5

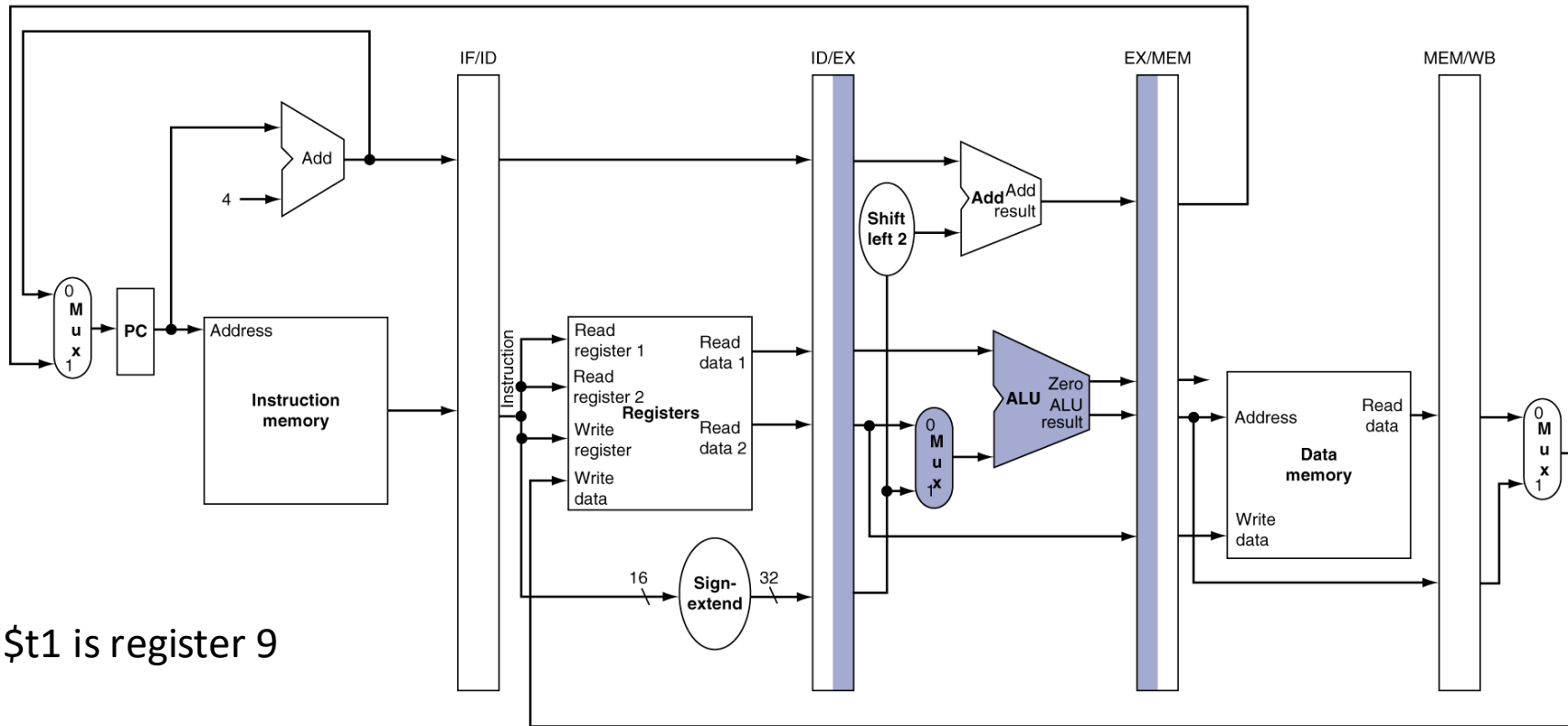
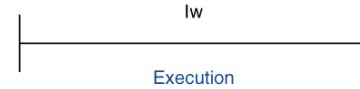
\$t1 holds 0x4810CAB0

0x4810CAB0 holds 12

The register file will output data from both read registers, but load will only use one of them. We should

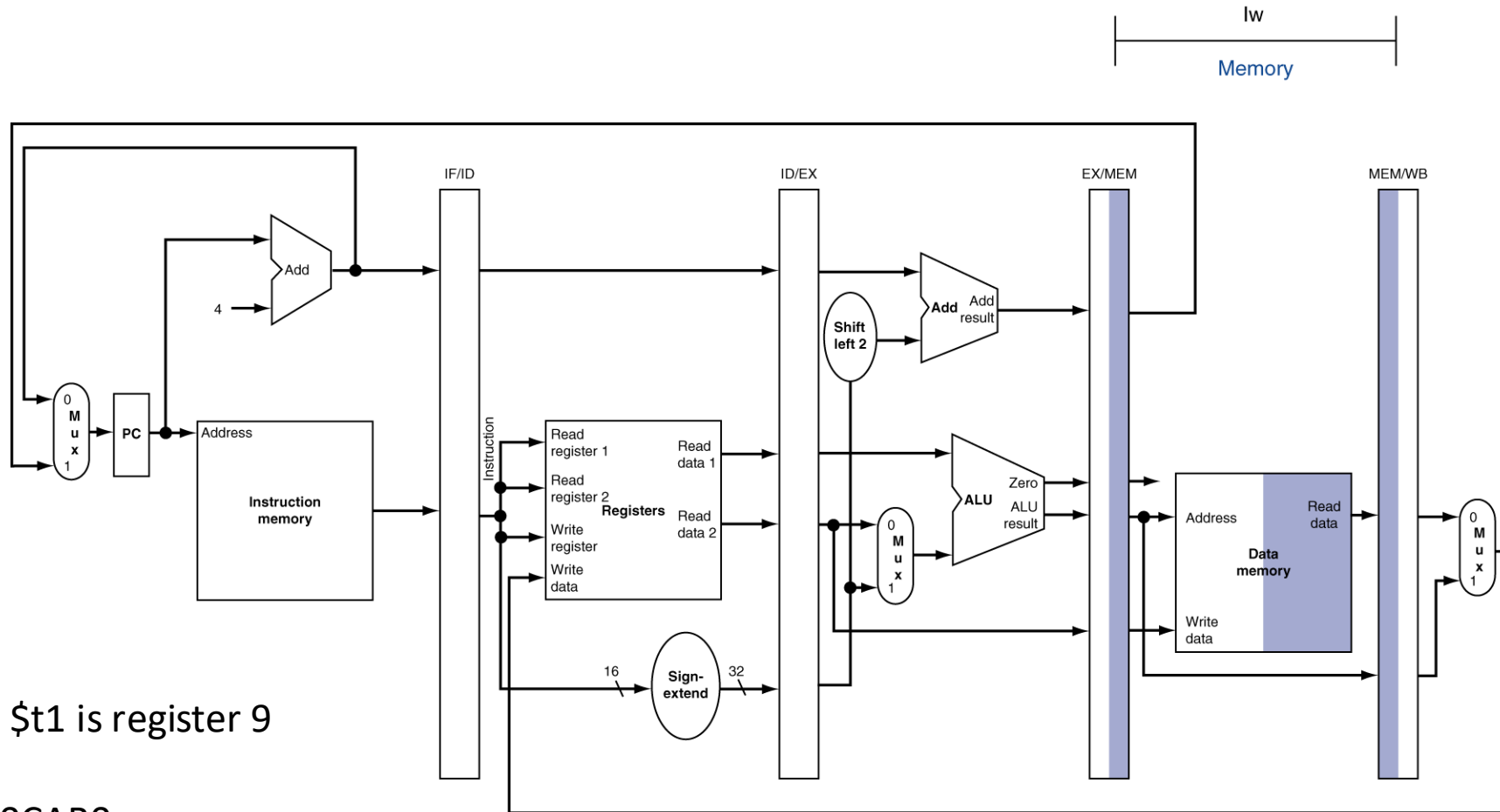
- A. Save both of them in ID/EX
- B. Only save the one we will use in ID/EX
- C. Do something else

# EX for lw \$t0, 4(\$t1)



\$t0 is register 8, \$t1 is register 9  
\$t0 holds 5  
\$t1 holds 0x4810CAB0  
0x4810CAB0 holds 12

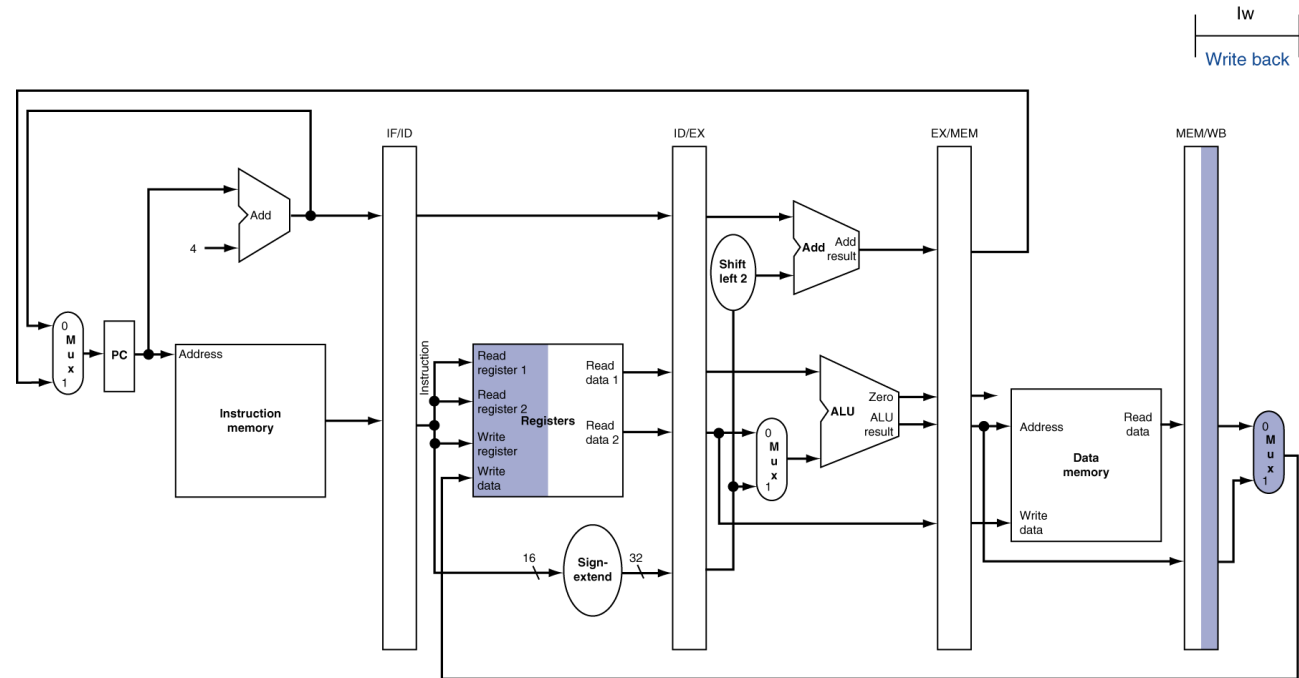
# MEM for lw \$t0, 4(\$t1)



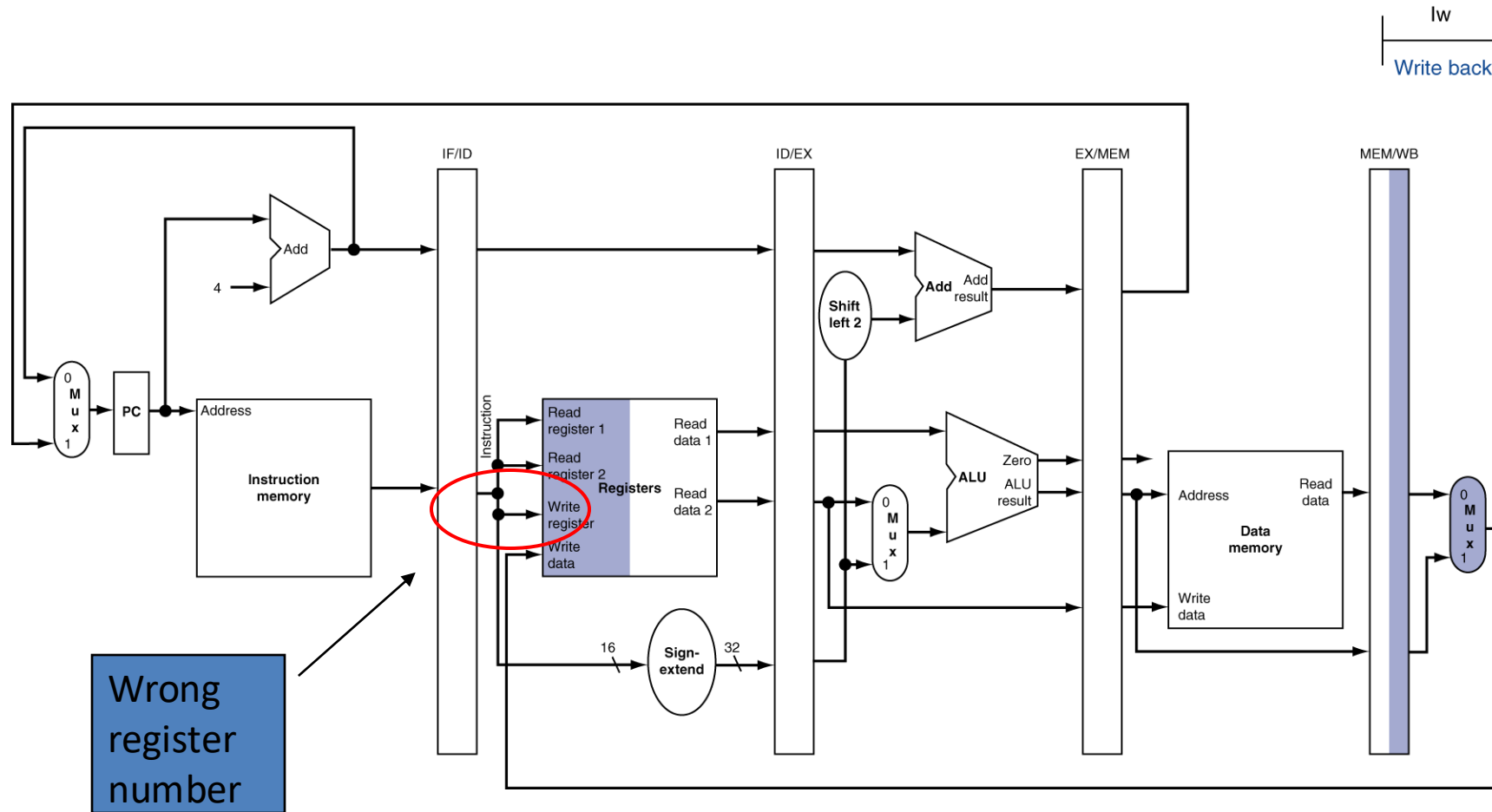
\$t0 is register 8, \$t1 is register 9  
\$t0 holds 5  
\$t1 holds 0x4810CAB0  
0x4810CAB0 holds 12

# When we do WB for load

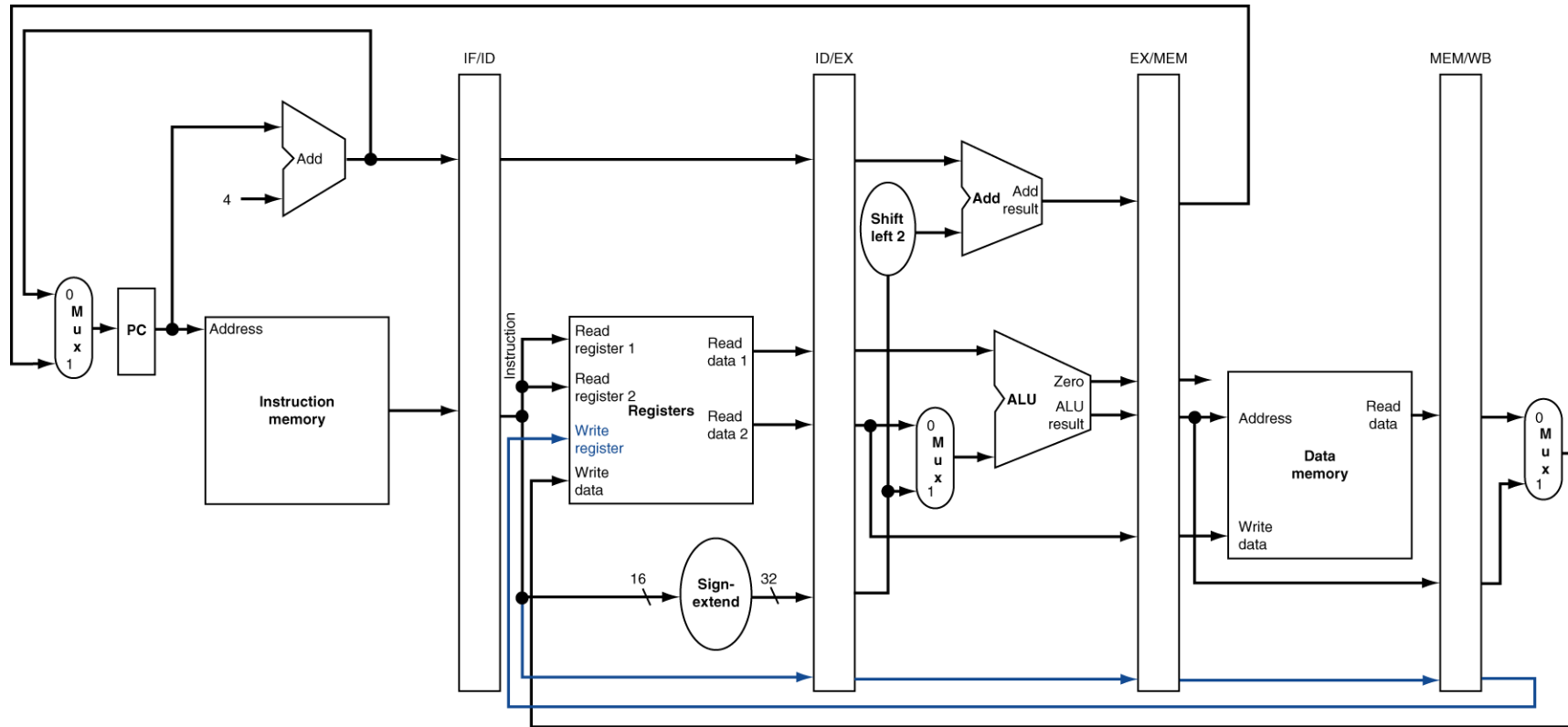
- A. Everything will be fine
- B. The data to write to the register will be wrong
- C. The register number to write to will be wrong
- D. Both the data and the register number will be wrong



# WB for Load



# Corrected Datapath for Load



# Reading

- Next lecture: Pipelined Datapath
  - Section 5.7–5.7.10