

# CSCI 210: Computer Architecture

## Lecture 28: Pipelining

Stephen Checkoway

Oberlin College

Dec. 13, 2021

Slides from Cynthia Taylor

# Announcements

- Problem Set 9 due Friday
- Lab 8 due Sunday, January 2, 2022
- Office hours Tuesday 13:30–14:30

# 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)

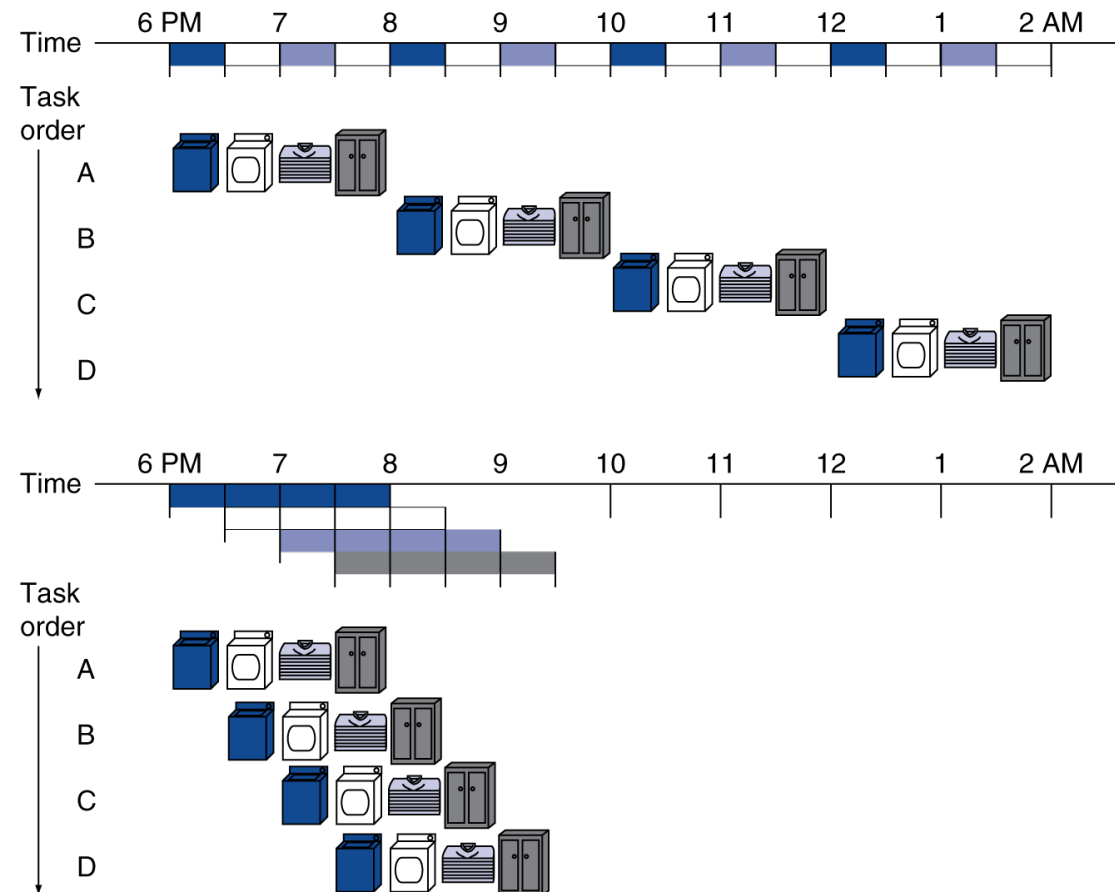
Instr.	I Mem	Reg Rd	ALU Op	Adder	D Mem	Reg Wr	Total
R-type							
load							
store							
beq							
jump							

# 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

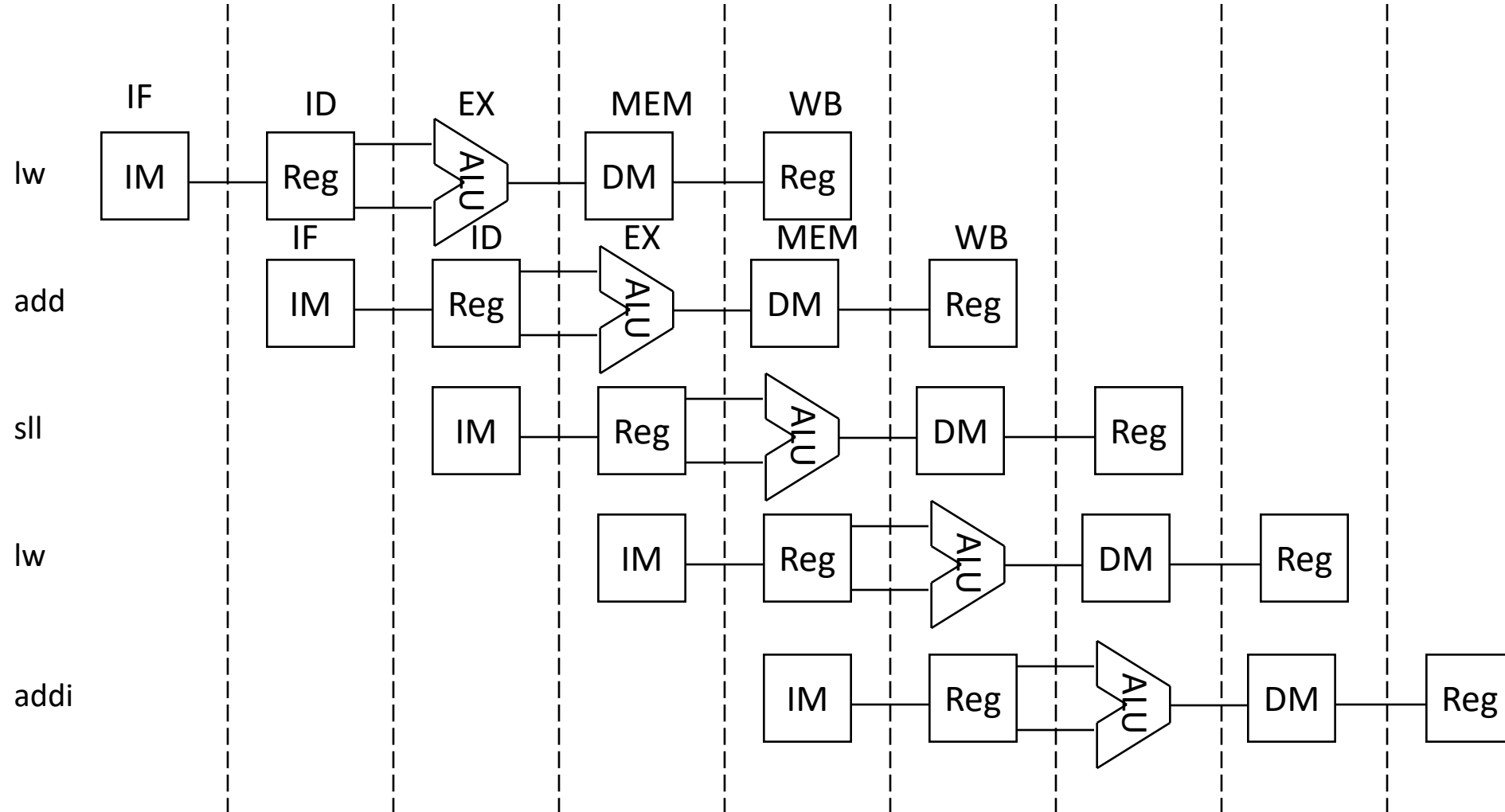
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



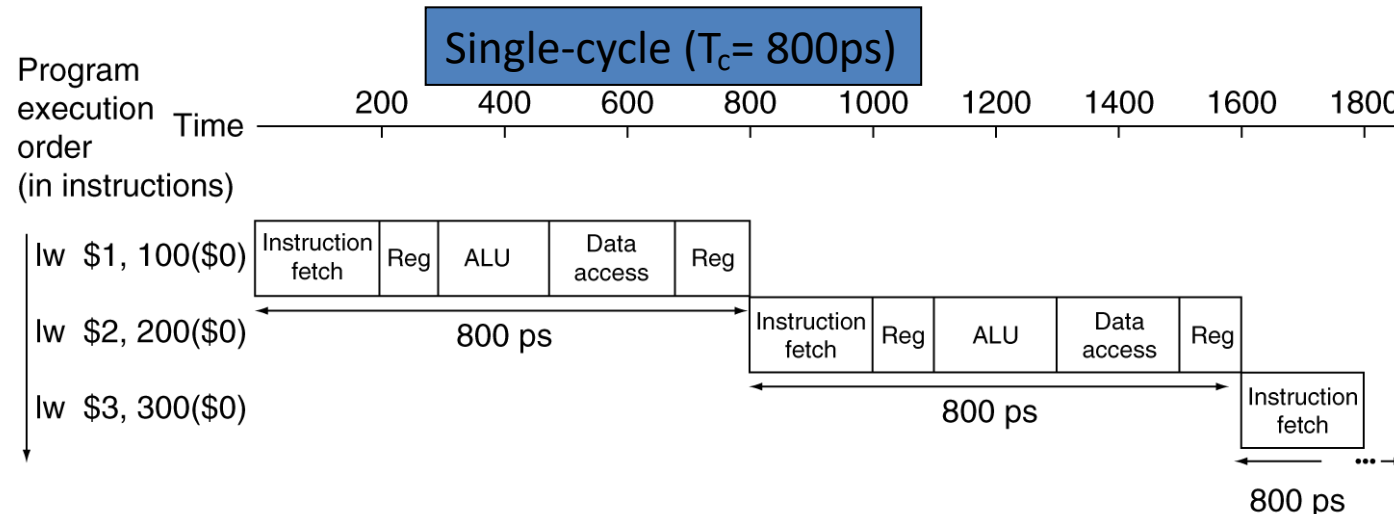
# MIPS Pipeline

- 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!)

# 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

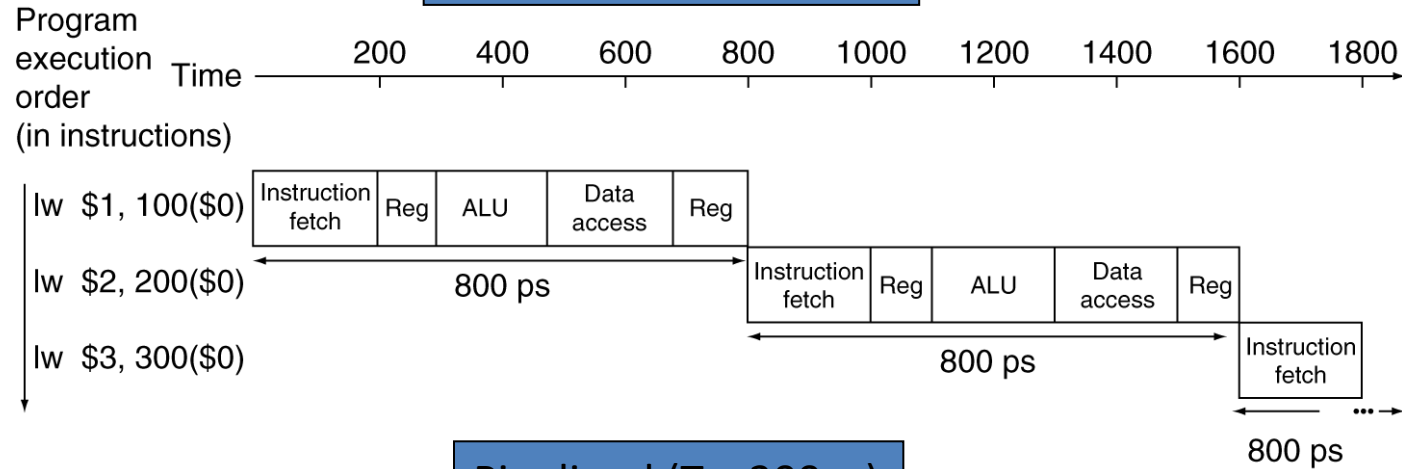
- A. 900 ps
- B. 1000 ps
- C. 1200 ps
- D. 1600 ps

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- TIP: try drawing the new diagram

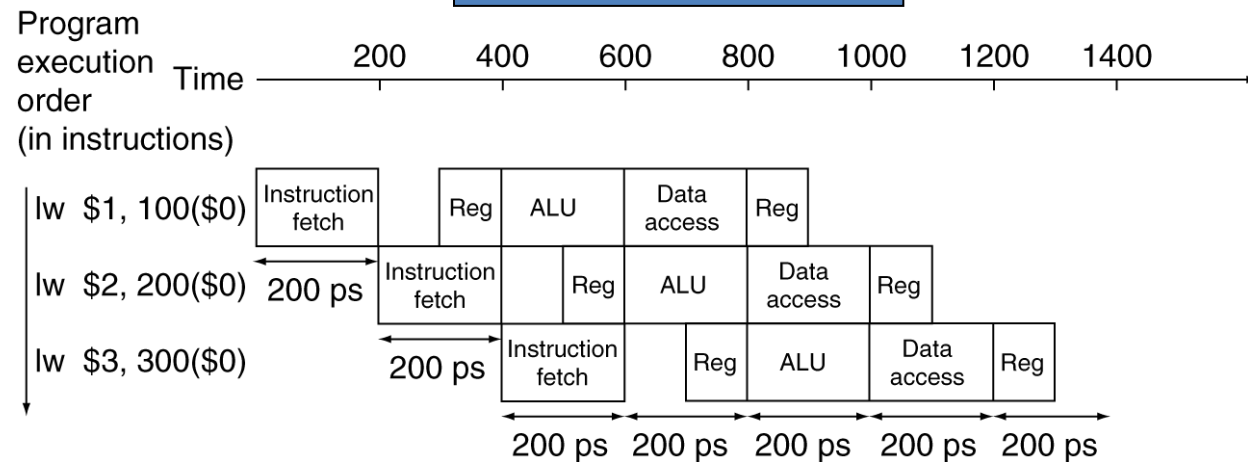


# Pipeline Performance

Single-cycle ( $T_c = 800\text{ps}$ )



Pipelined ( $T_c = 200\text{ps}$ )



# Maximum Pipeline Speedup

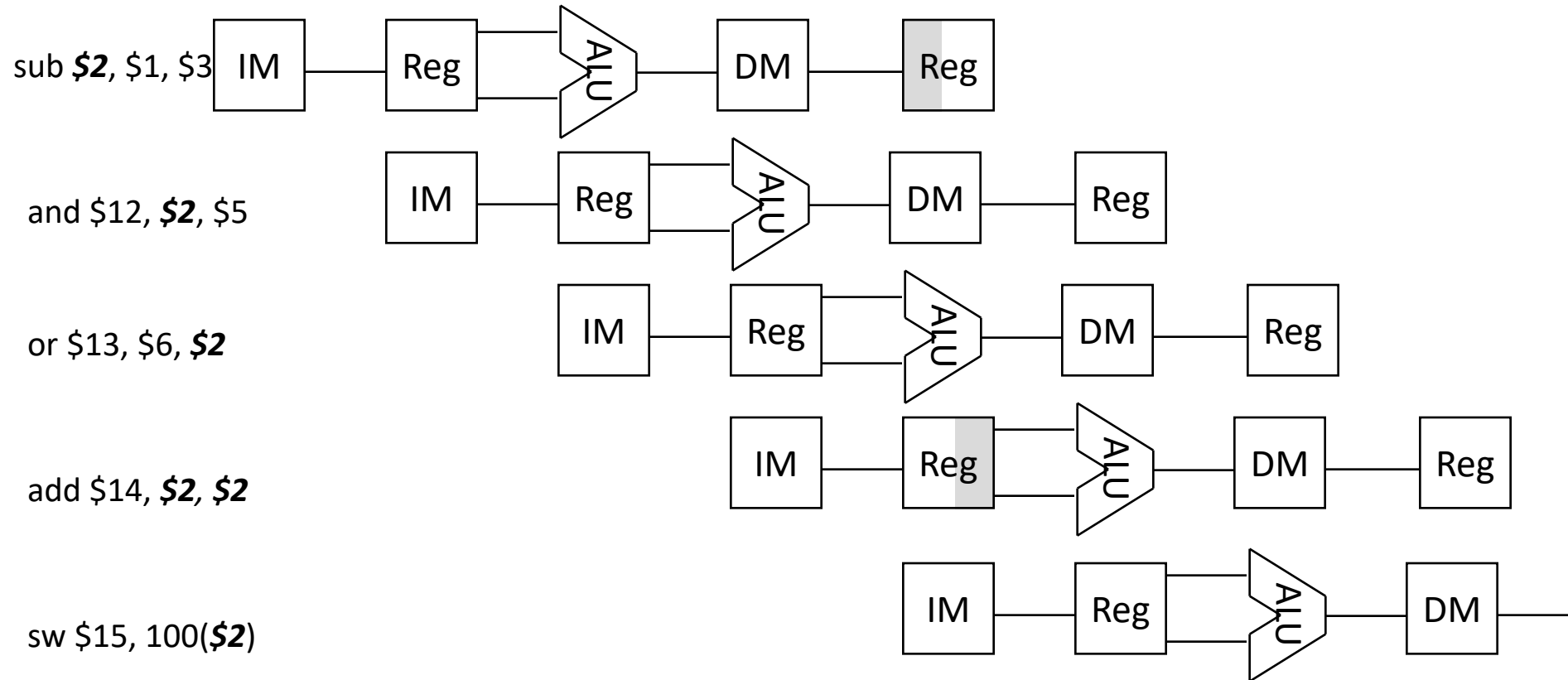
- If all stages are balanced
  - i.e., all take the same time

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

- 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
    - c.f. x86: 1- to 16-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



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

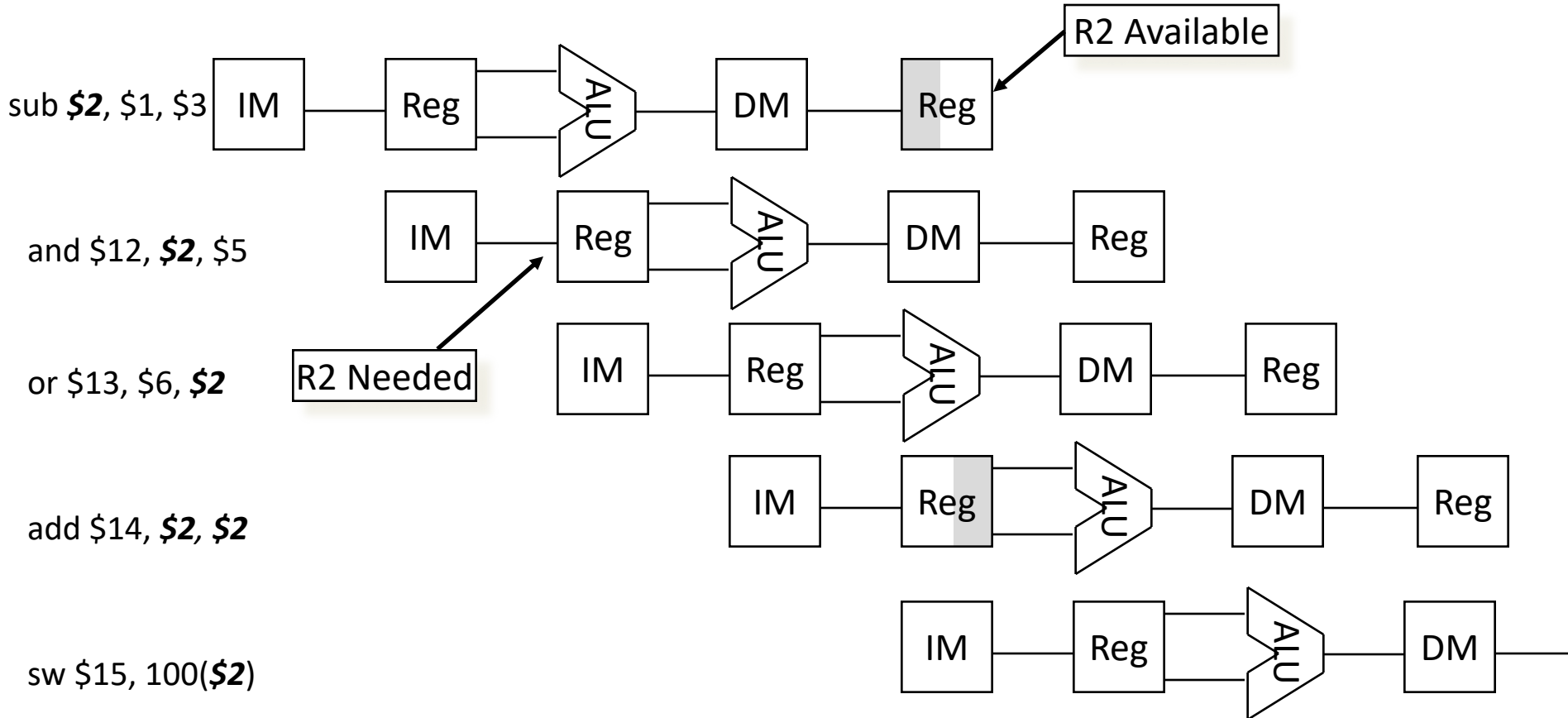
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

# 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 “caches” which we’ll talk about later)

# Data Hazards

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



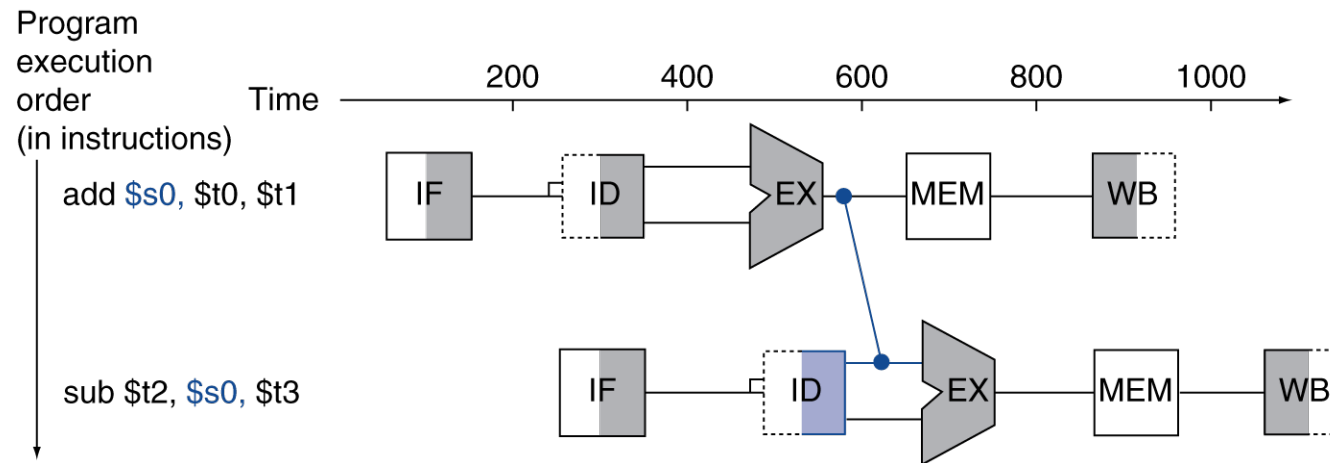
# We could solve data hazards by

- 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 datapath



Would forwarding work if our instructions were

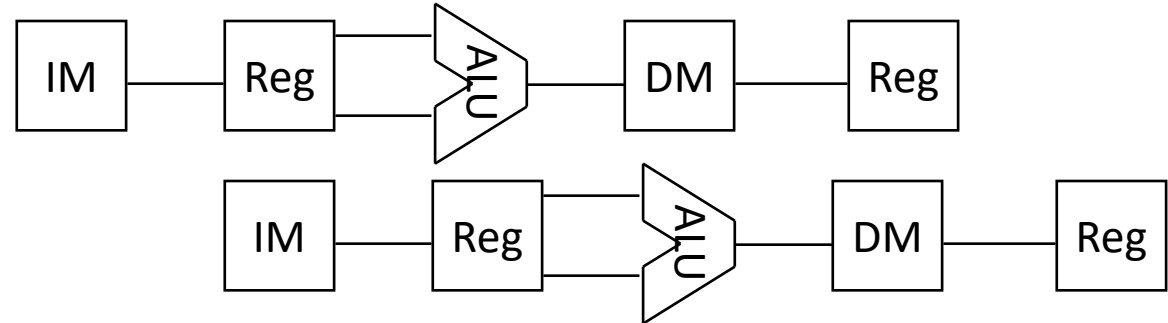
`lw $s0, 20($1)`

`sub $t2, $s0, $t3`

A. Yes

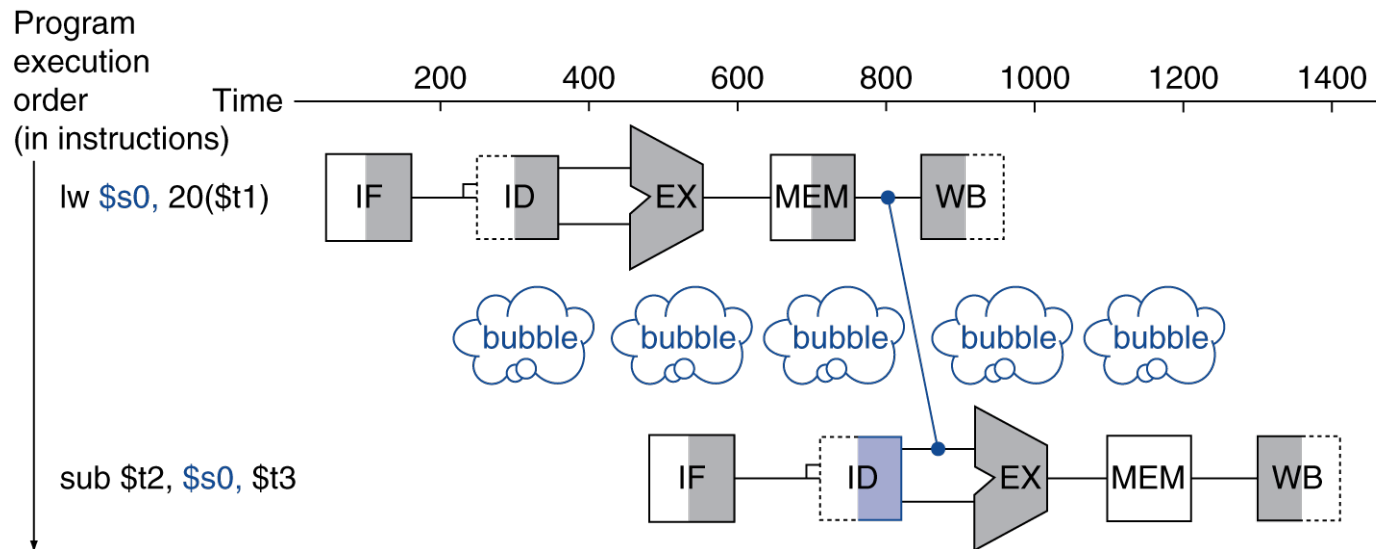
B. No

C. Depends on the value loaded



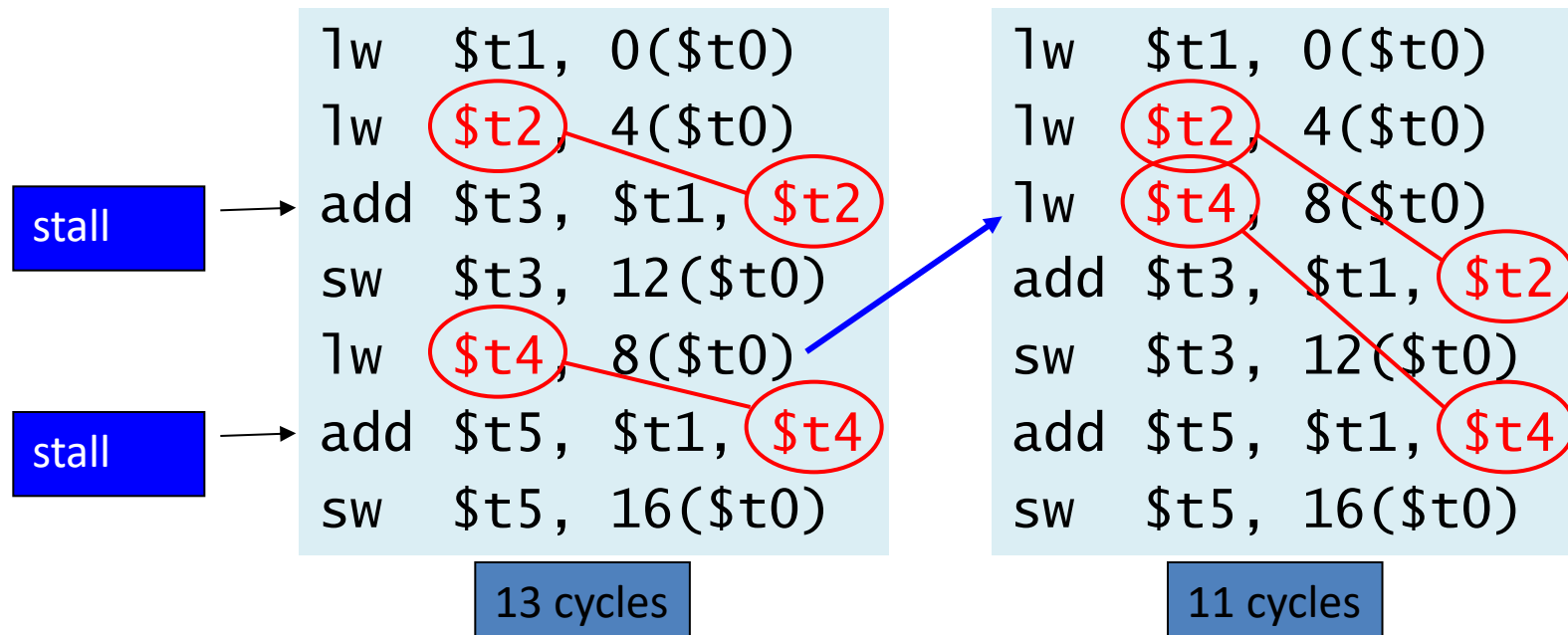
# Load-Use Data Hazard

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



# Control Hazards

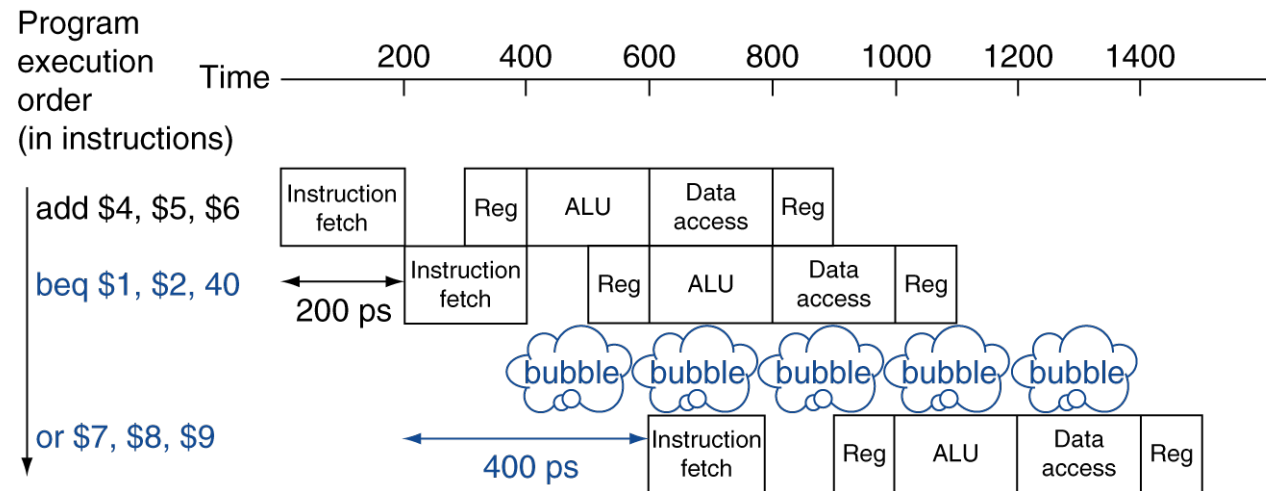
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it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

# To solve control hazards, we could

- A. Rearrange instructions.
- B. Guess what the branch will do.
- 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 17% 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 1.17!
- Even worse if we can't resolve it 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

# Pipeline 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

# Reading

- Next lecture: Pipelined Datapath
  - Section 5.7–5.7.10
- Problem Set 9 due Friday
- Lab 8 due Sunday, January 2, 2022