# Chapter 4

## The Processor

# Introduction

3 factors affect computer performance
- 2 decided by the processor



- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

compiler converts C → ASM

- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

I-type

R-type

branch type

# Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

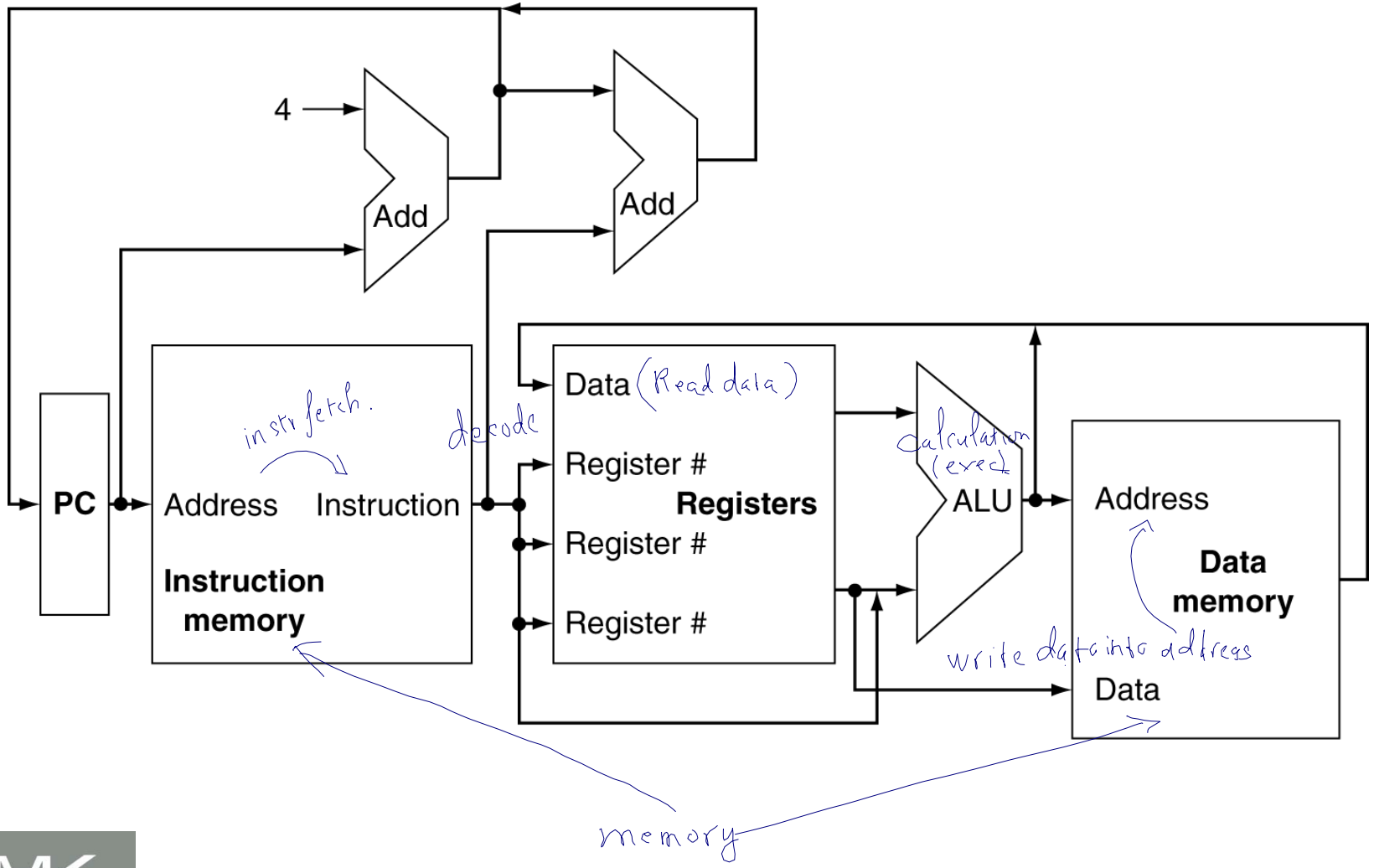
5 stages of each inst

1. instruction fetch (IF)
2. instruction decode
3. Execution
4. Memory access
5. Write to registers

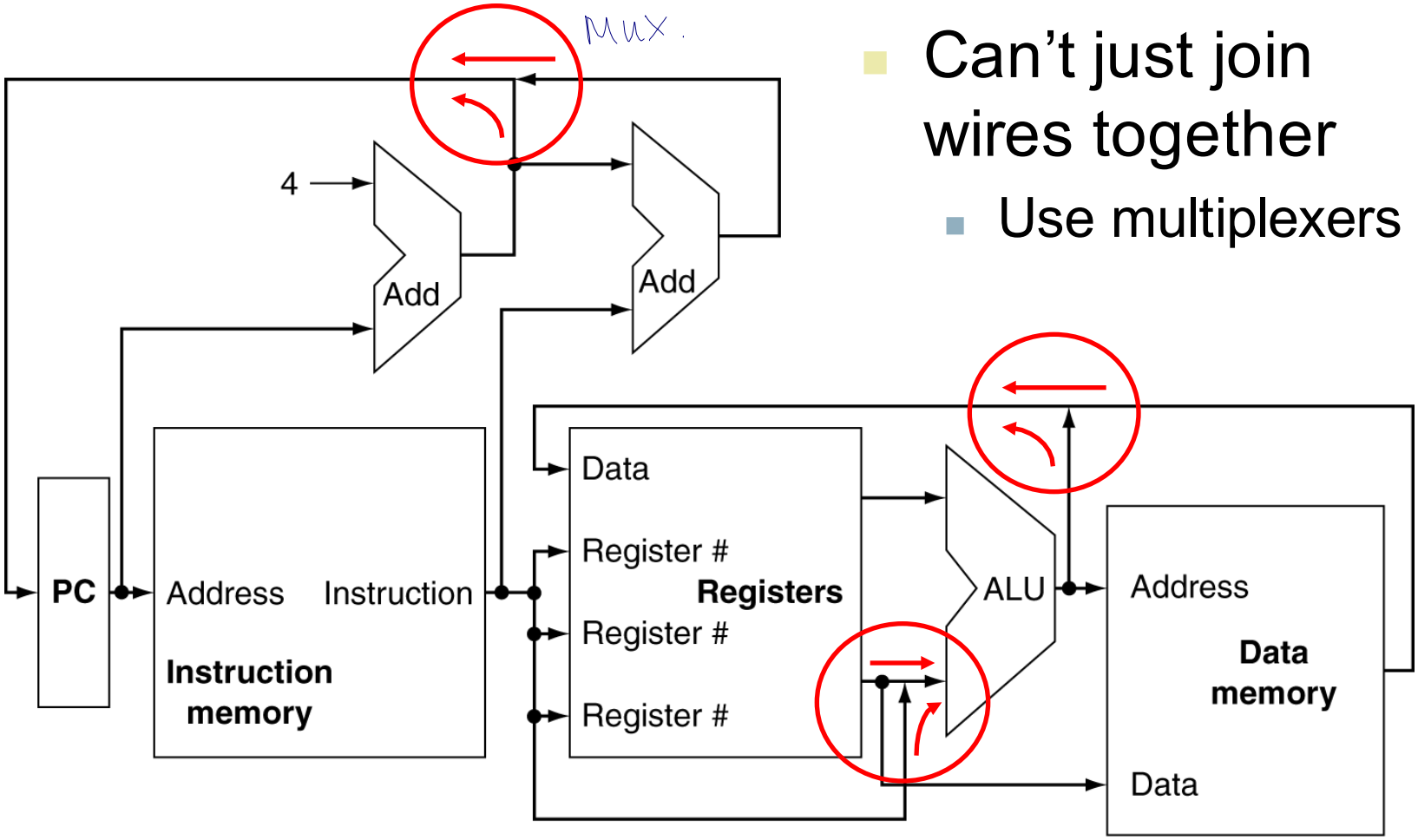
4 bytes

# CPU Overview

HW view ( input, output, datapath, memory, ? )

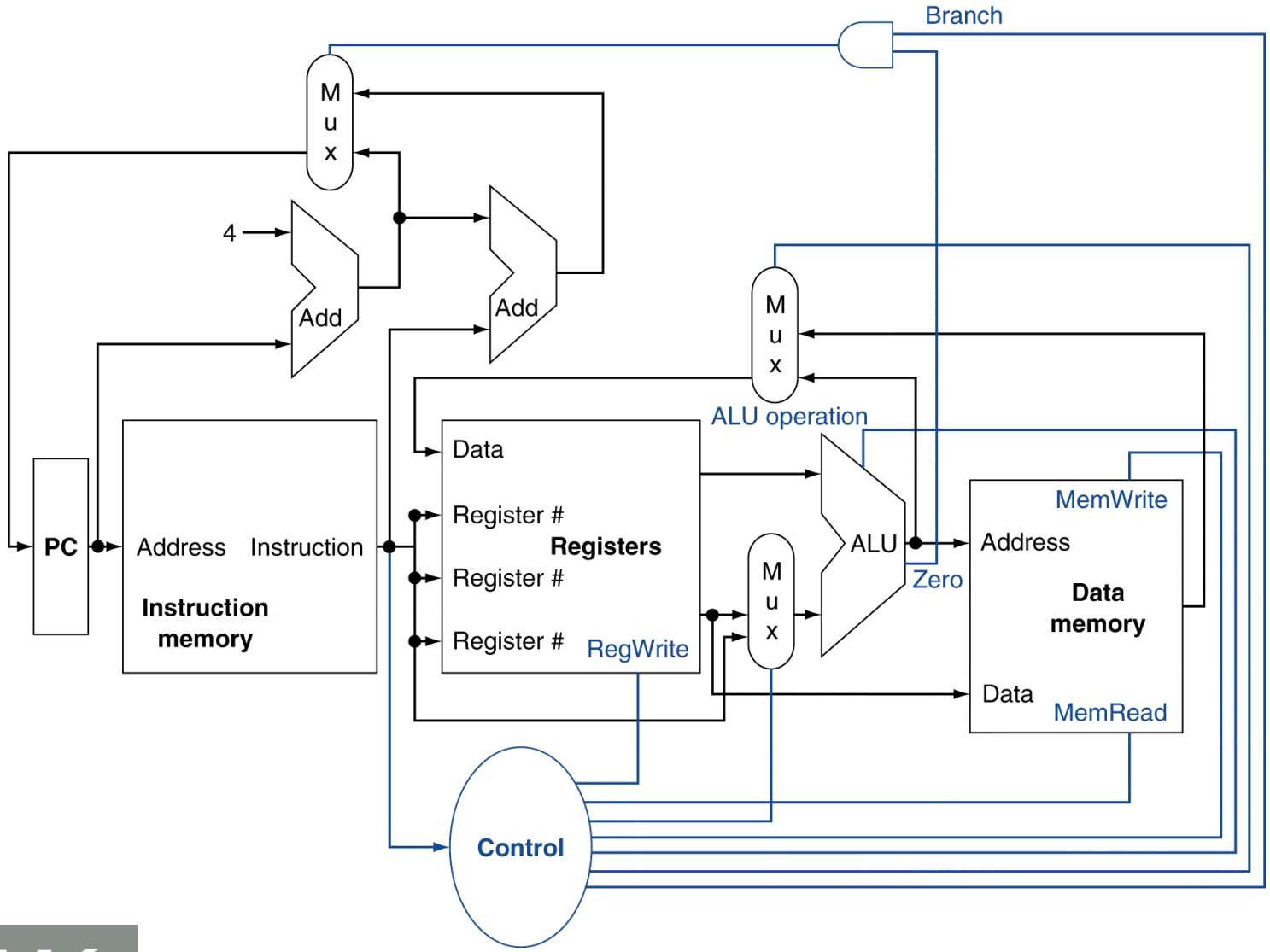


# Multiplexers




- Can't just join wires together
  - Use multiplexers

# Control


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

# Logic Design Basics

all registers use seq cell.

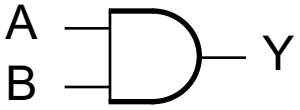
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element ← cell used in comb circuit. input change → output chang.
  - Operate on data
  - Output is a function of input
- State (sequential) elements ← cell used in comb circuit.
  - Store information

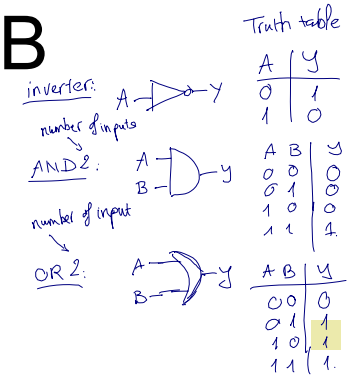
output depends on. both | input. previous output aka feedback (use clock to know old, new output) has feedback circuit.

Differences
- seq remembers old result by | clock. feedback circuit.
- comb does not have clock nor feedback circuit.

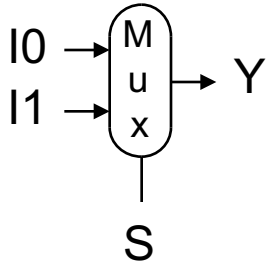
# Combinational Elements

- AND-gate
  - Y = A & B

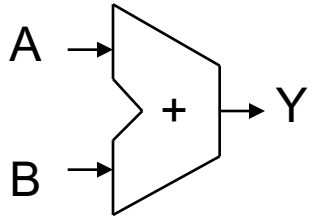


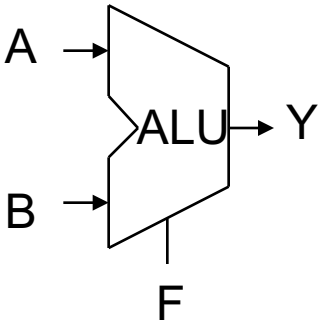

- Multiplexer
  - Y = S ? I1 : I0




- Adder
  - Y = A + B

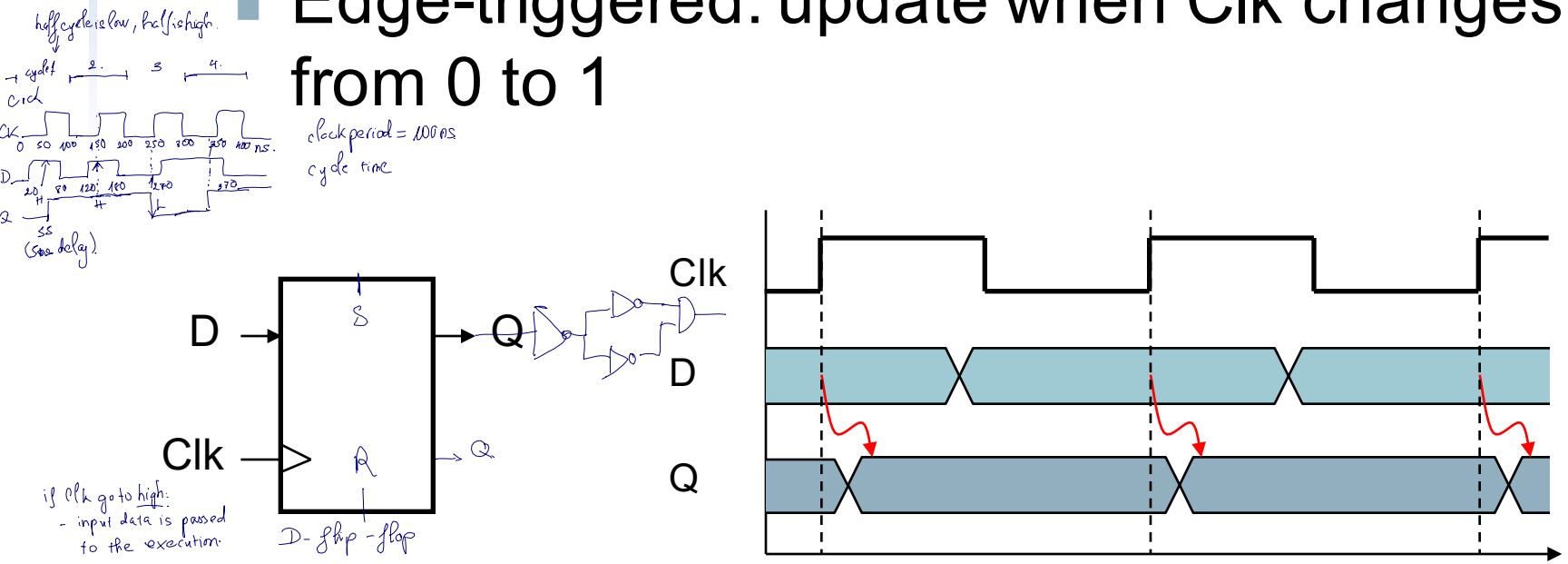



- Arithmetic/Logic Unit
  - Y = F(A, B)



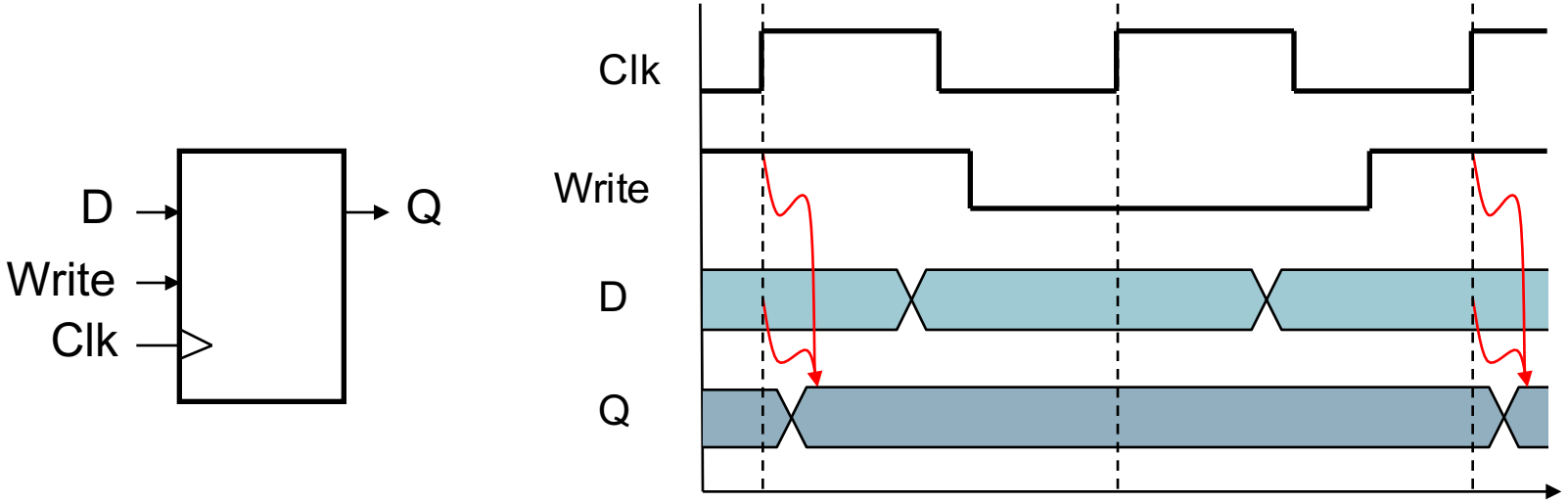
# Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1



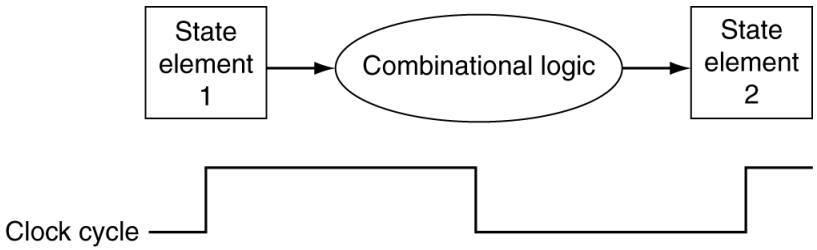
# Sequential Elements

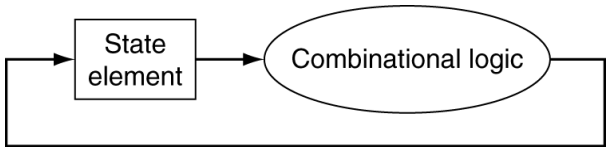
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



# Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period



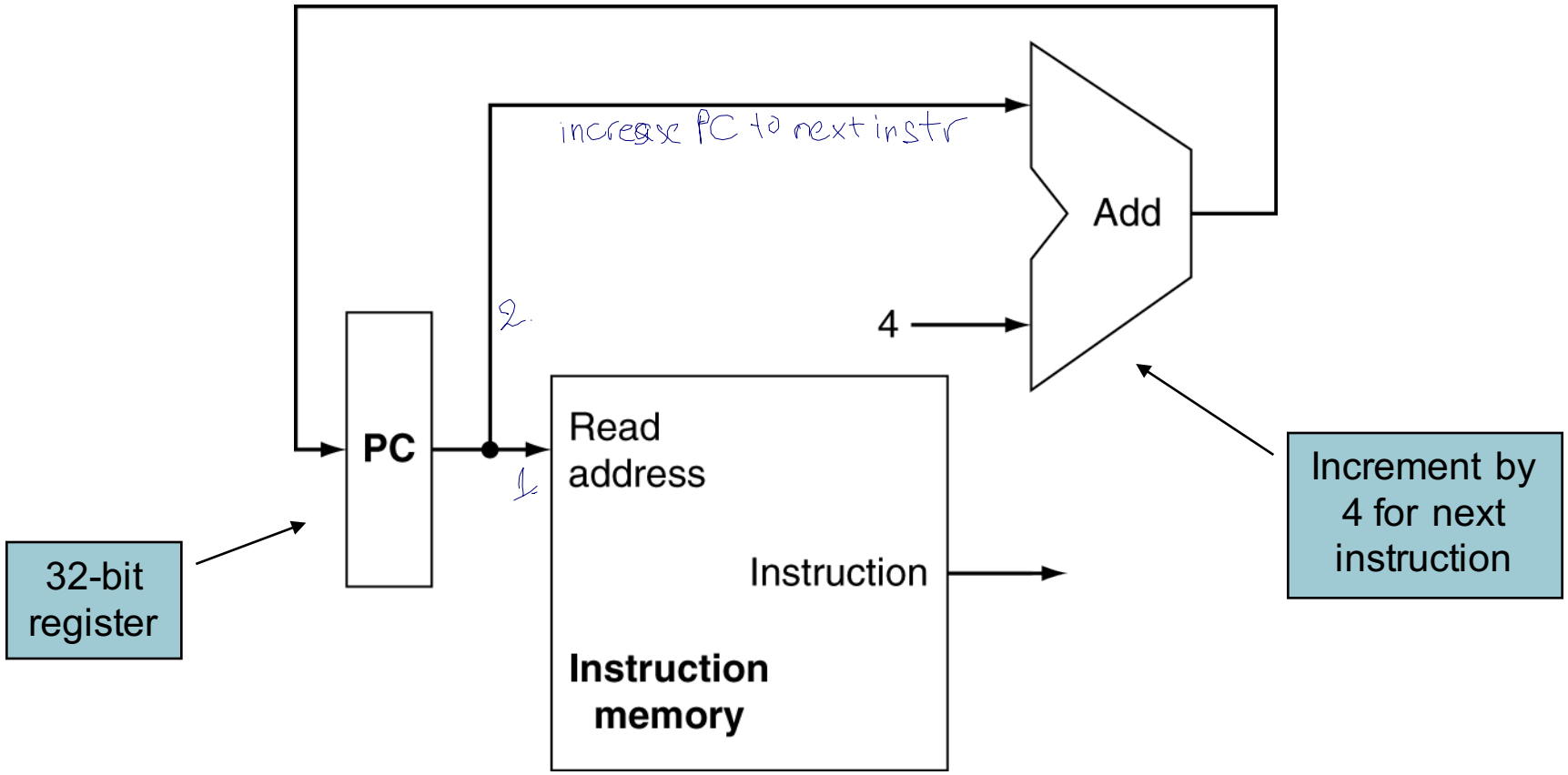



# Building a Datapath

- Datapath ← all hardware component that it goes thru. (instr)
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

# Instruction Fetch

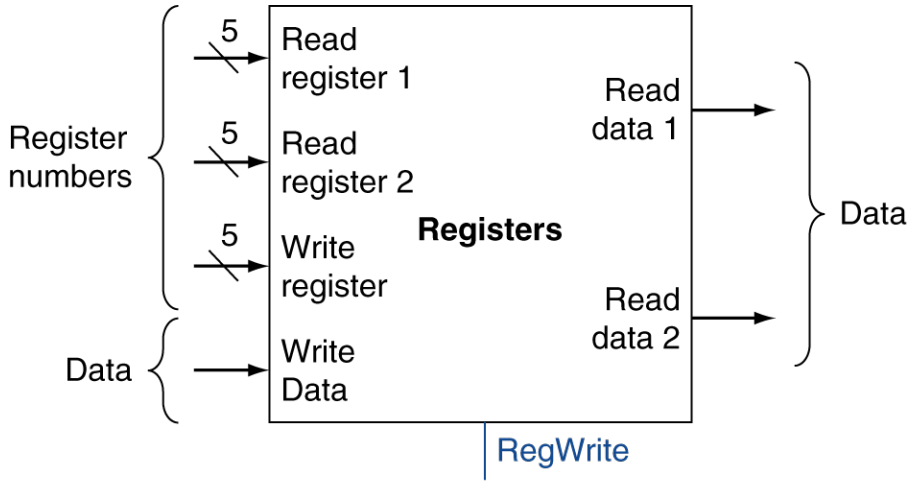
common stage in 5 stages for all instruction.



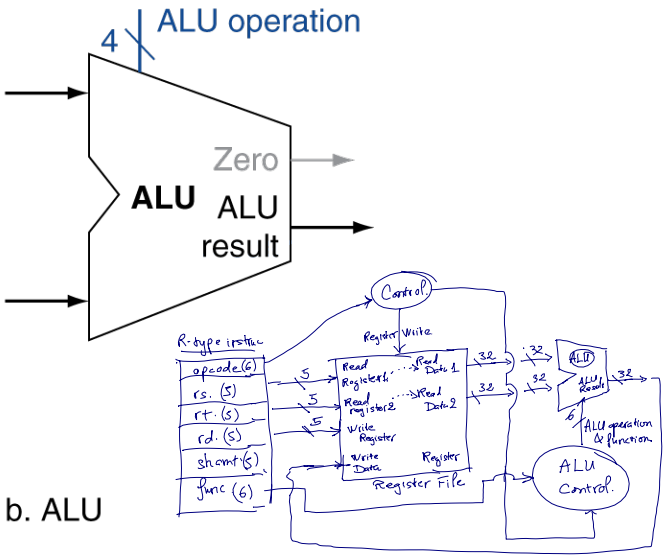
# R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

Decode the instruction.



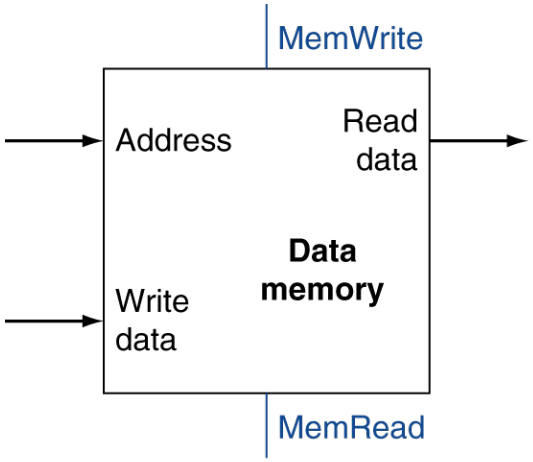

a. Registers



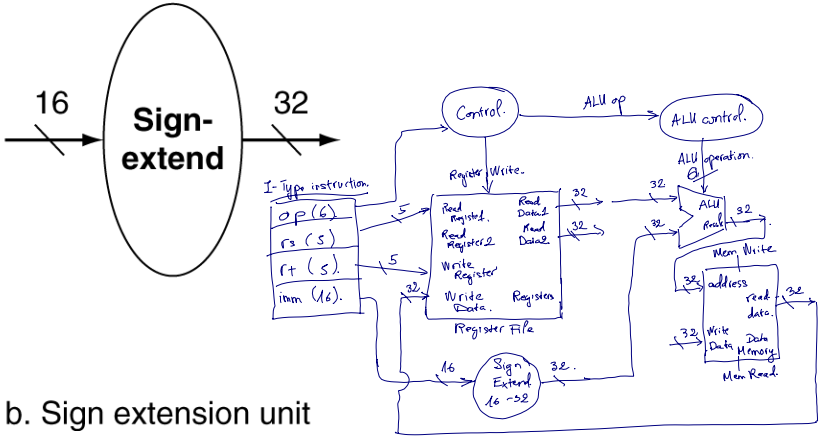

b. ALU

# Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory




a. Data memory unit

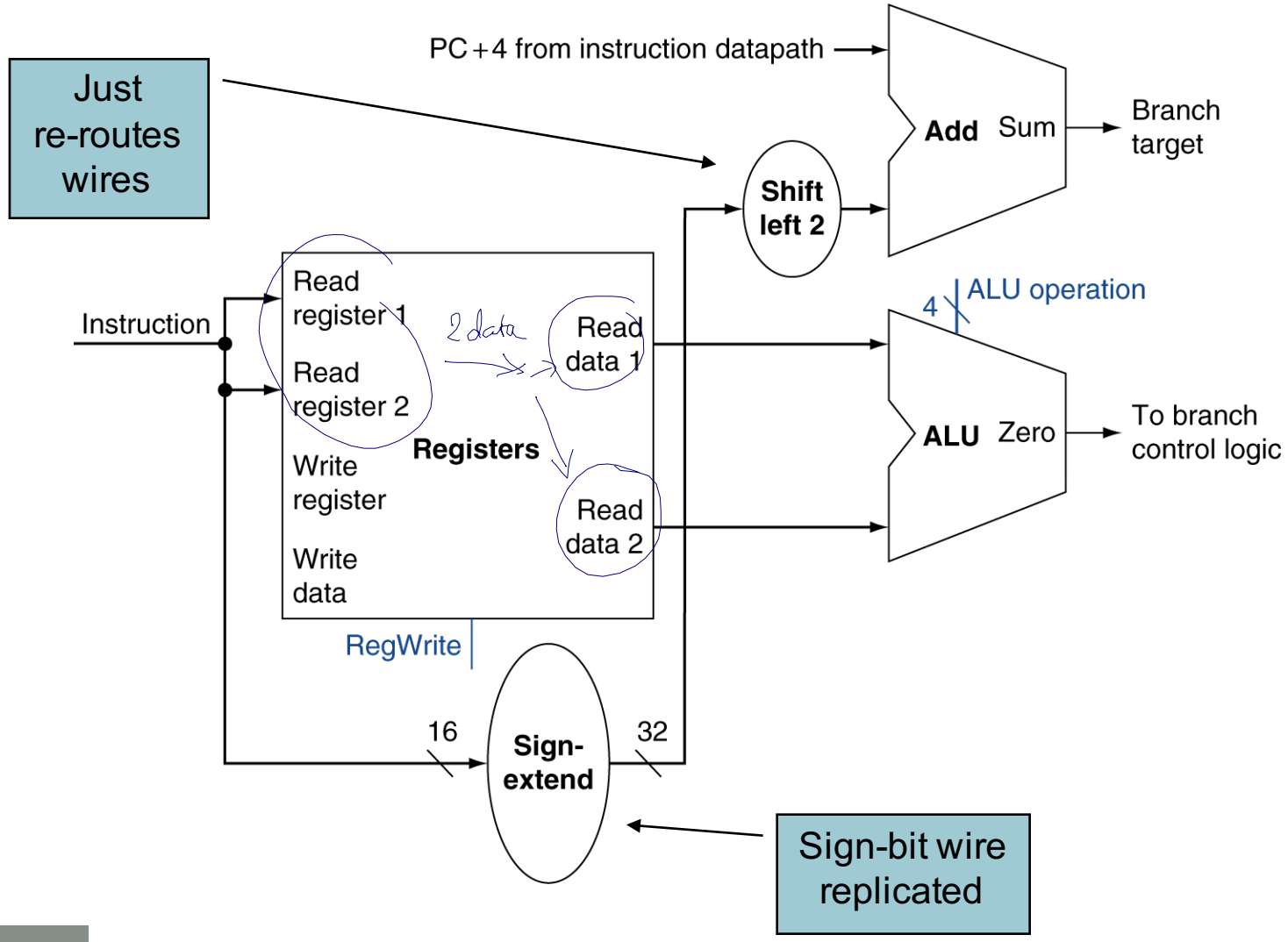



b. Sign extension unit

# Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

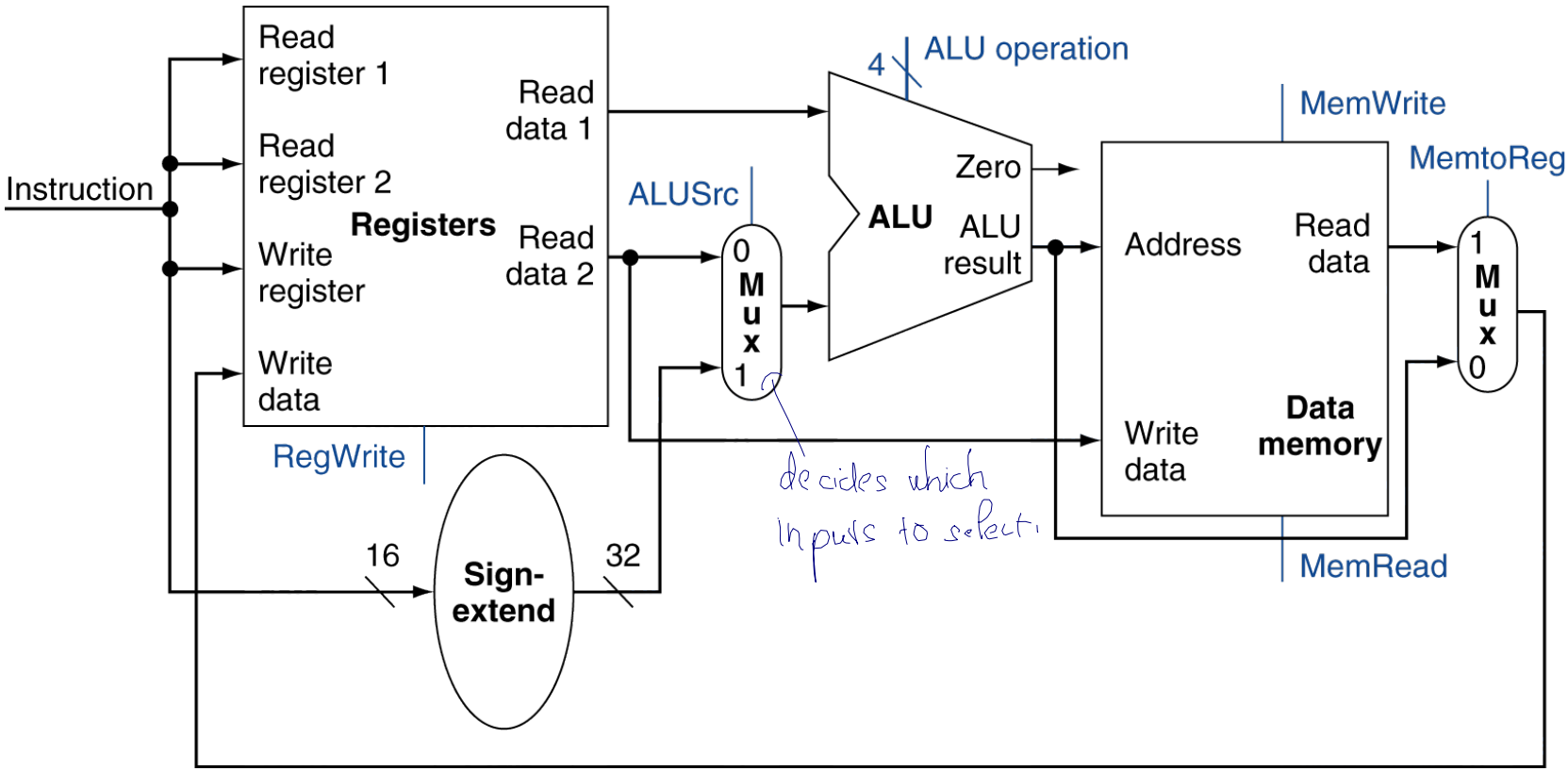
# Branch Instructions



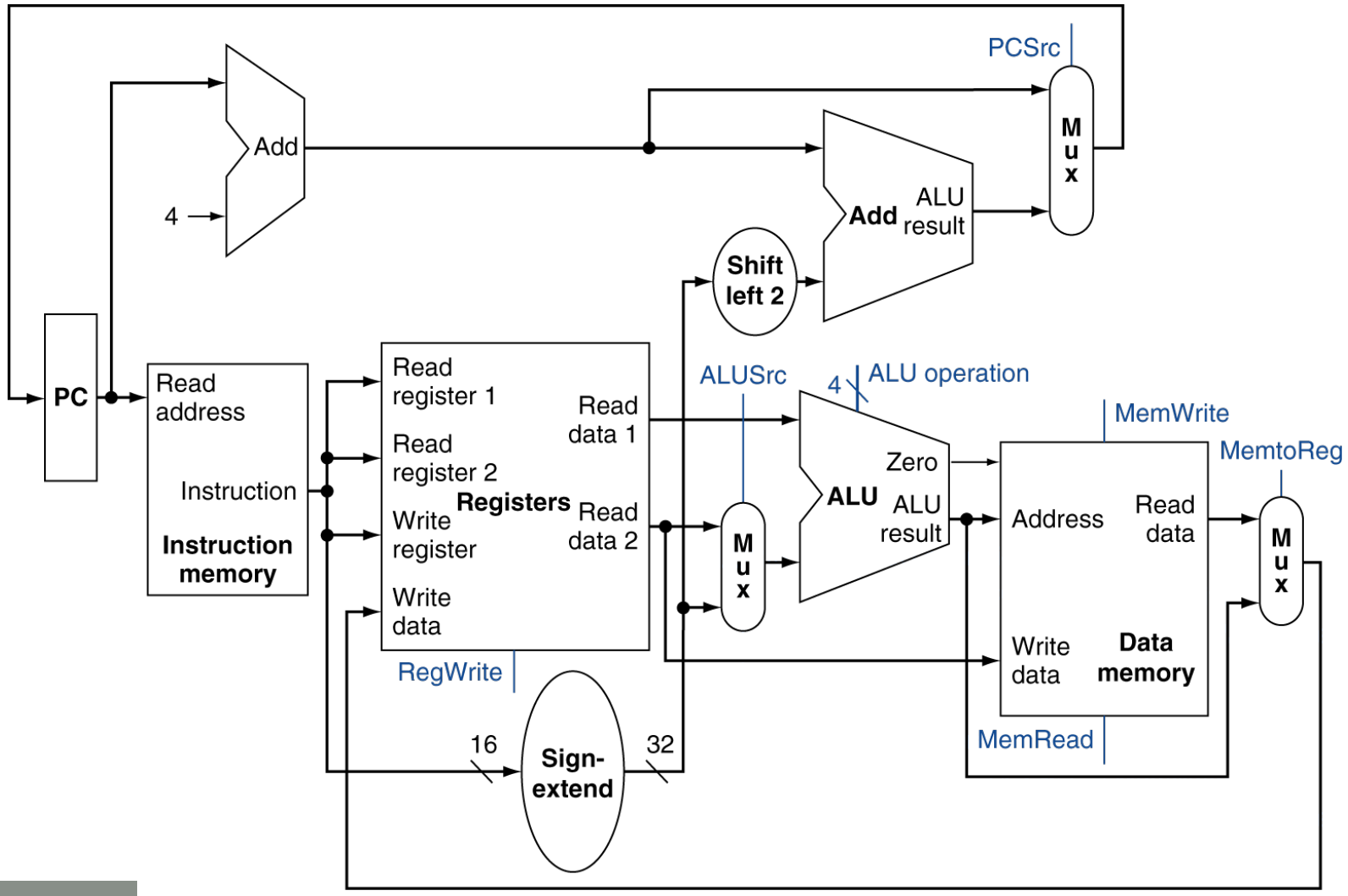
# Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

# R-Type/Load/Store Datapath



# Full Datapath



# ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

use control code to find out which func to execute.

| ALU control | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set-on-less-than |
| 1100 | NOR |

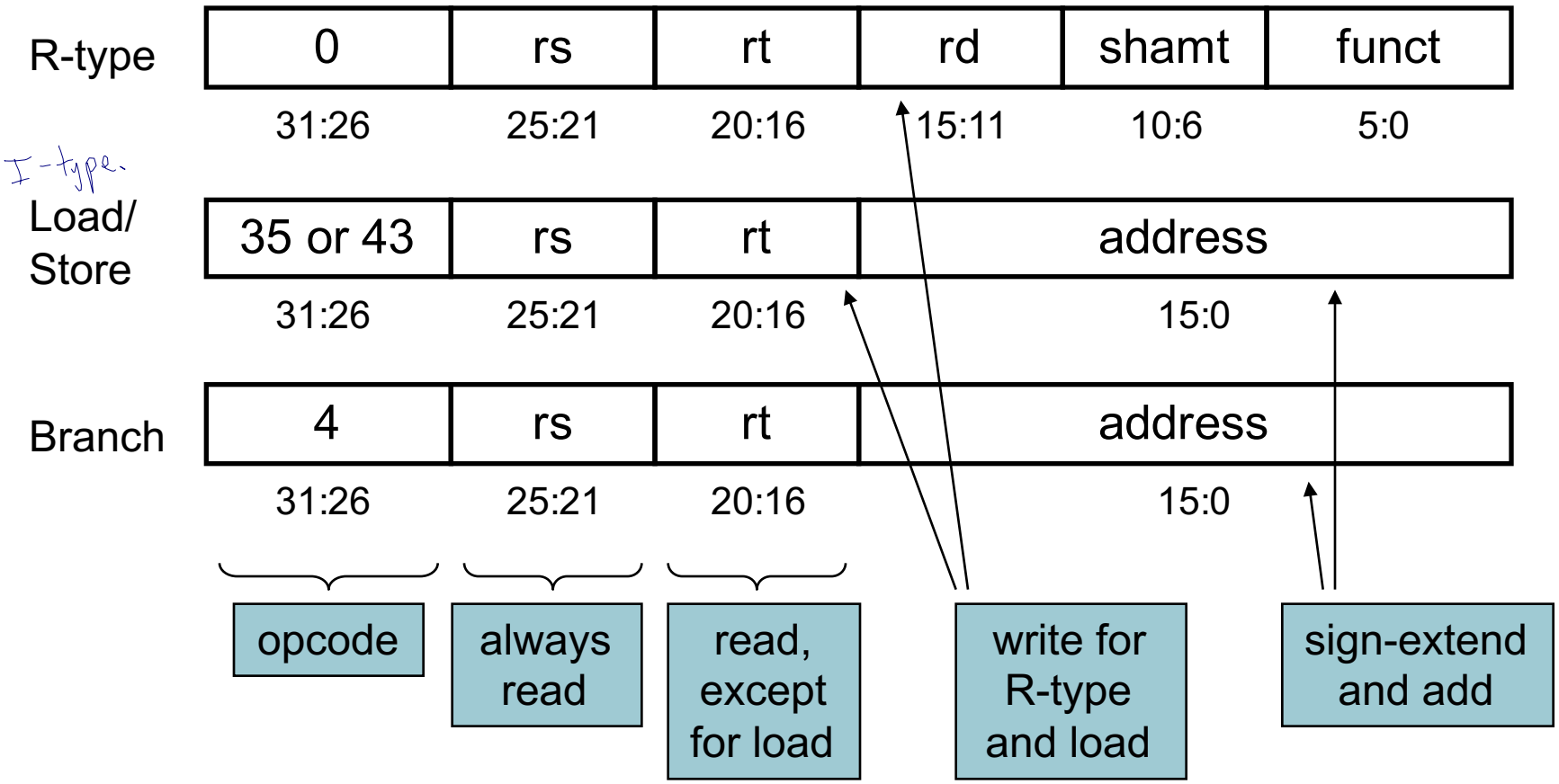
# ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

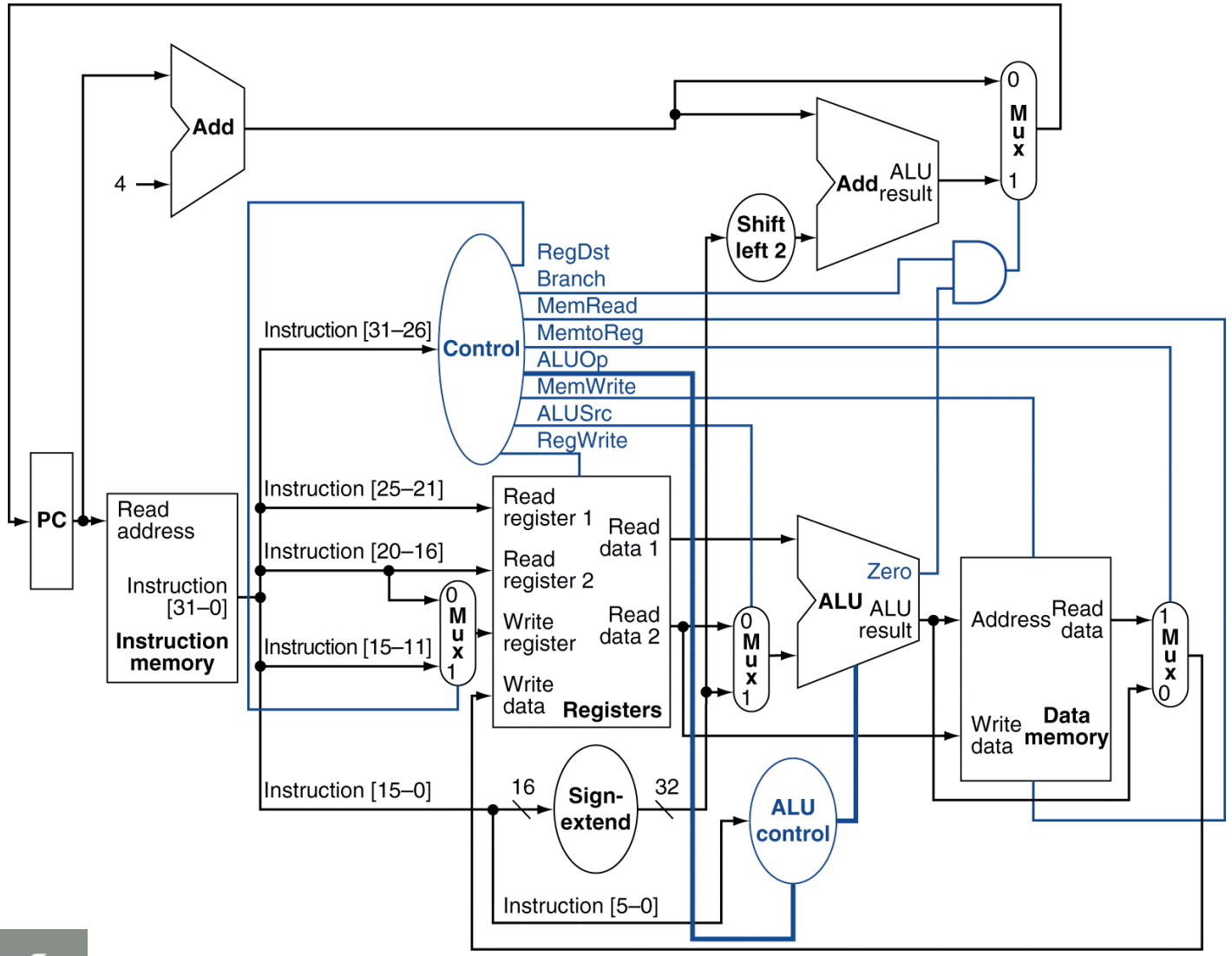
| opcode | ALUOp | Operation | funct | ALU function | ALU control |
|---|---|---|---|---|---|
| lw | 00 | load word | XXXXXX | add | 0010 |
| sw | 00 | store word | XXXXXX | add | 0010 |
| beq | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| | | subtract | 100010 | subtract | 0110 |
| | | AND | 100100 | AND | 0000 |
| | | OR | 100101 | OR | 0001 |
| | | set-on-less-than | 101010 | set-on-less-than | 0111 |

# The Main Control Unit

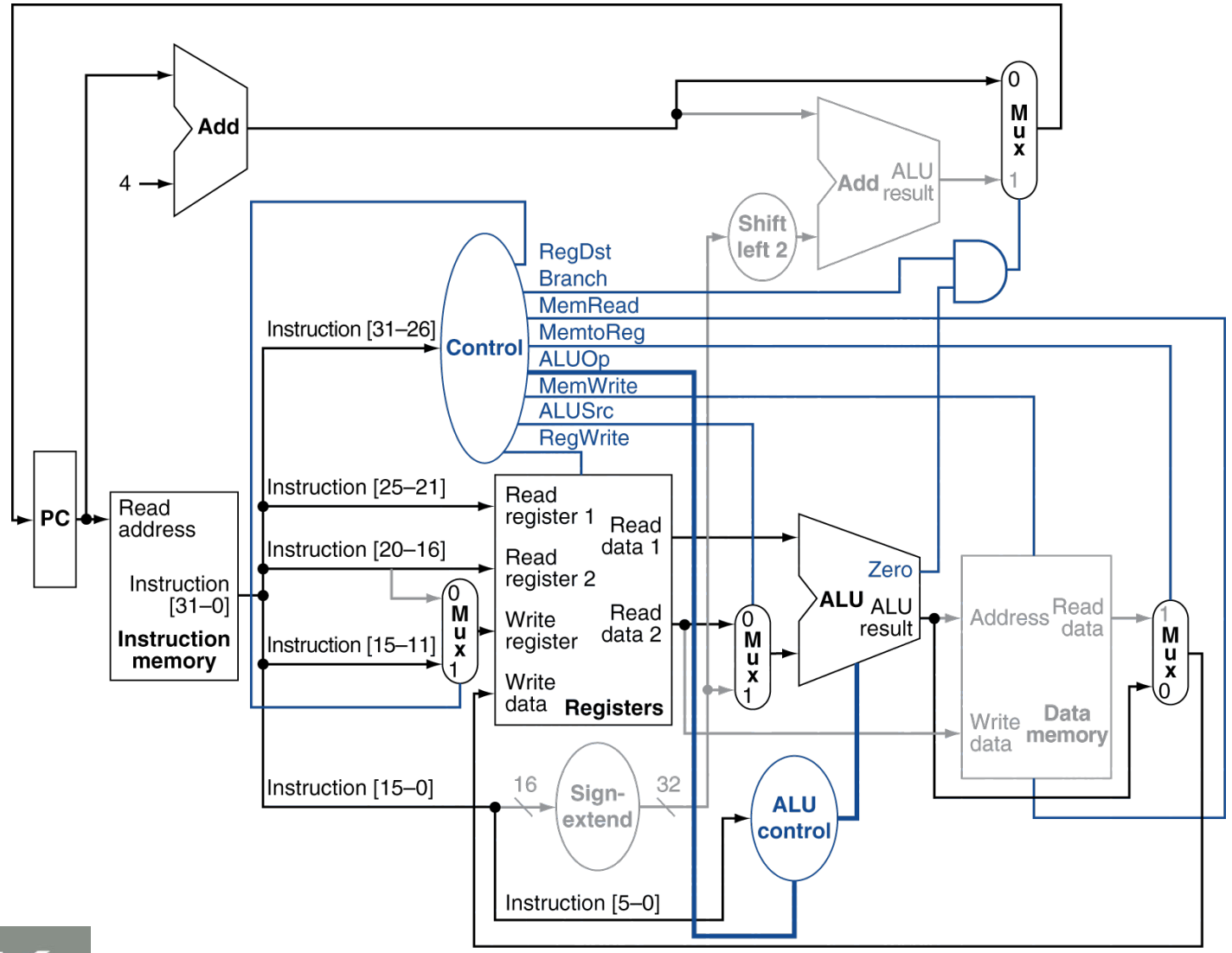
- Control signals derived from instruction



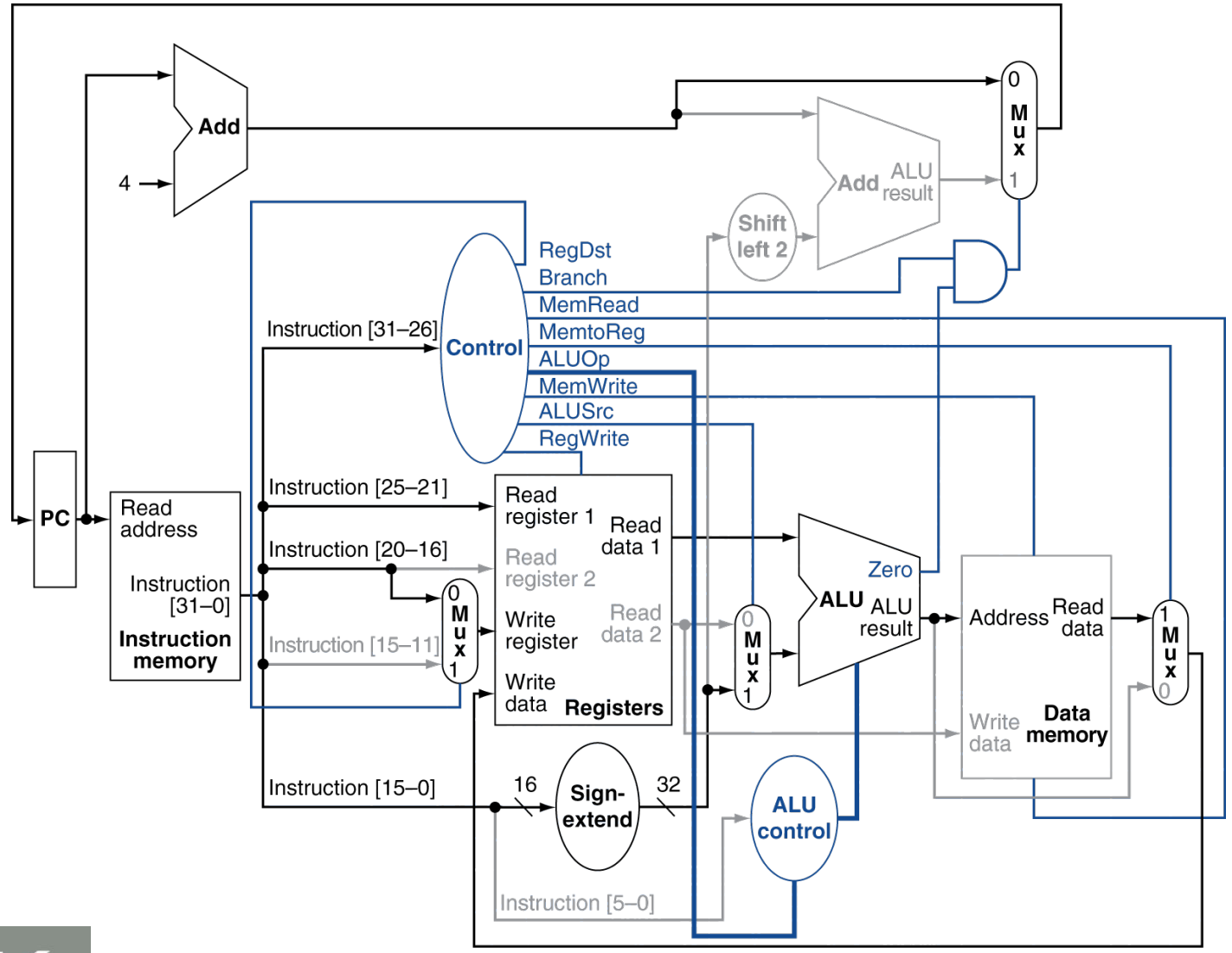
# Datapath With Control



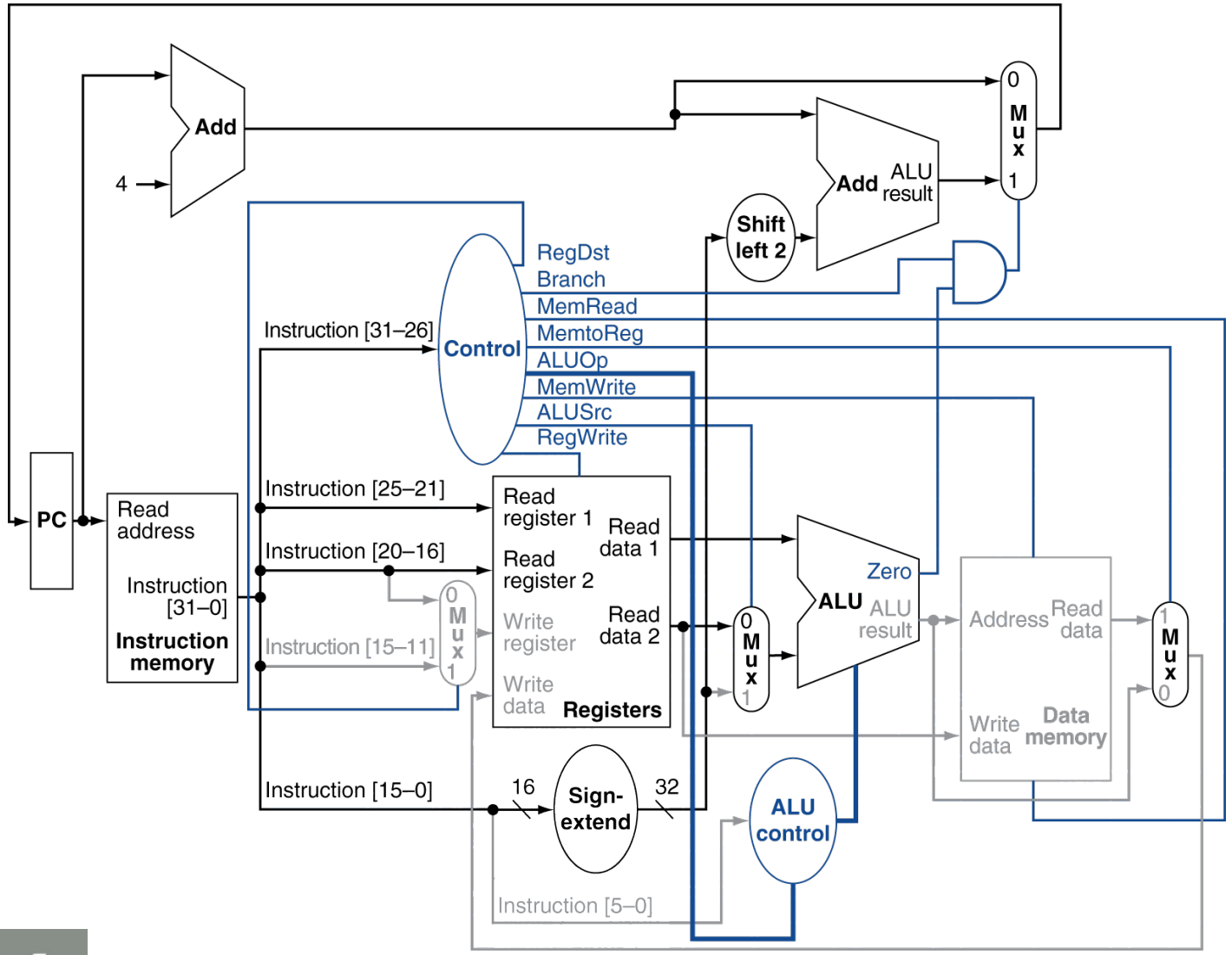
# R-Type Instruction



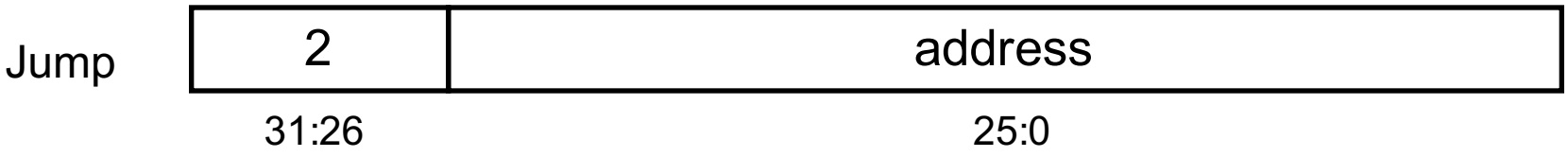
# Load Instruction



# Branch-on-Equal Instruction

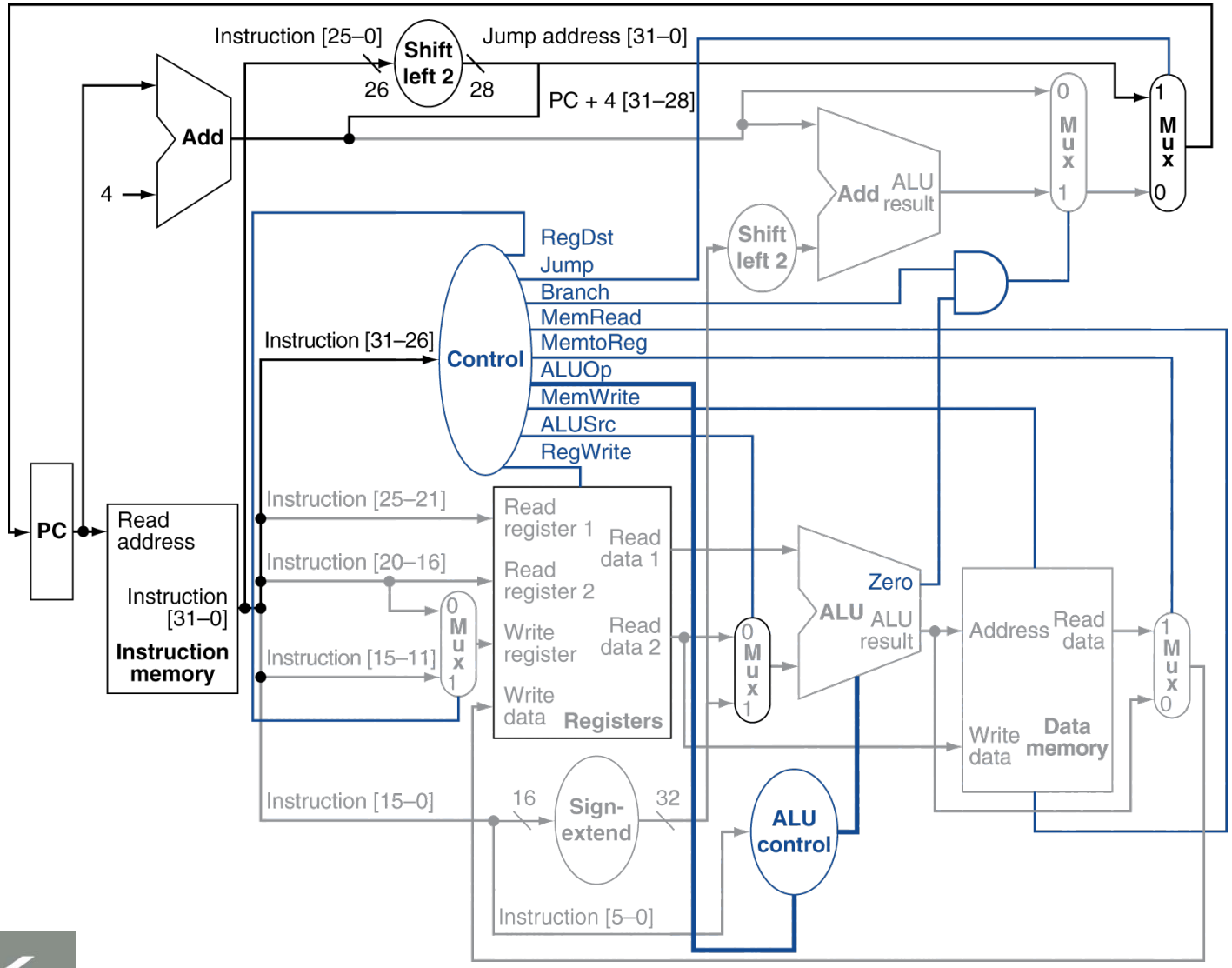


# Implementing Jumps




- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

# Datapath With Jumps Added



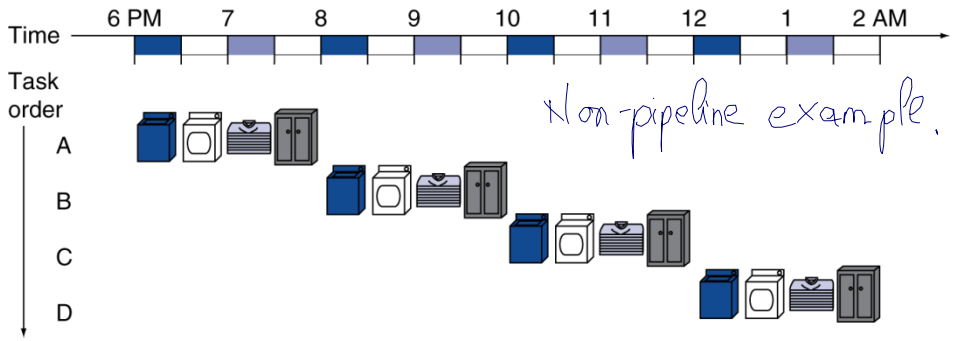
# Performance Issues

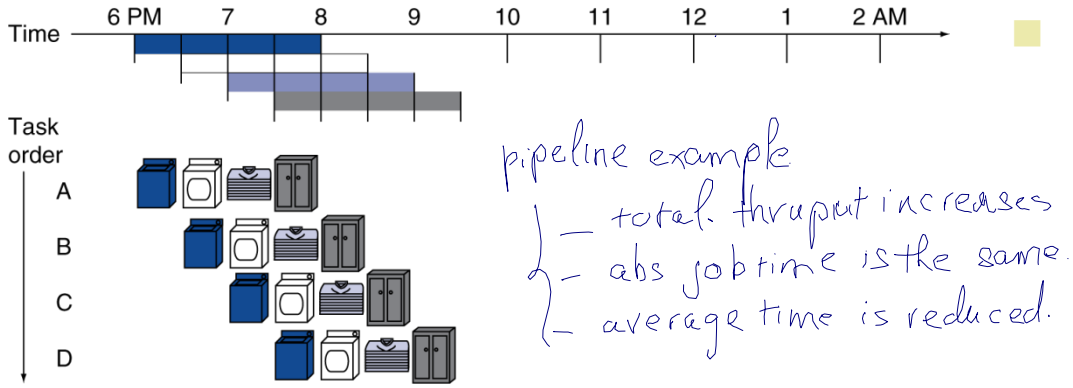
Each instructions takes different time to finish.

- 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 = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 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

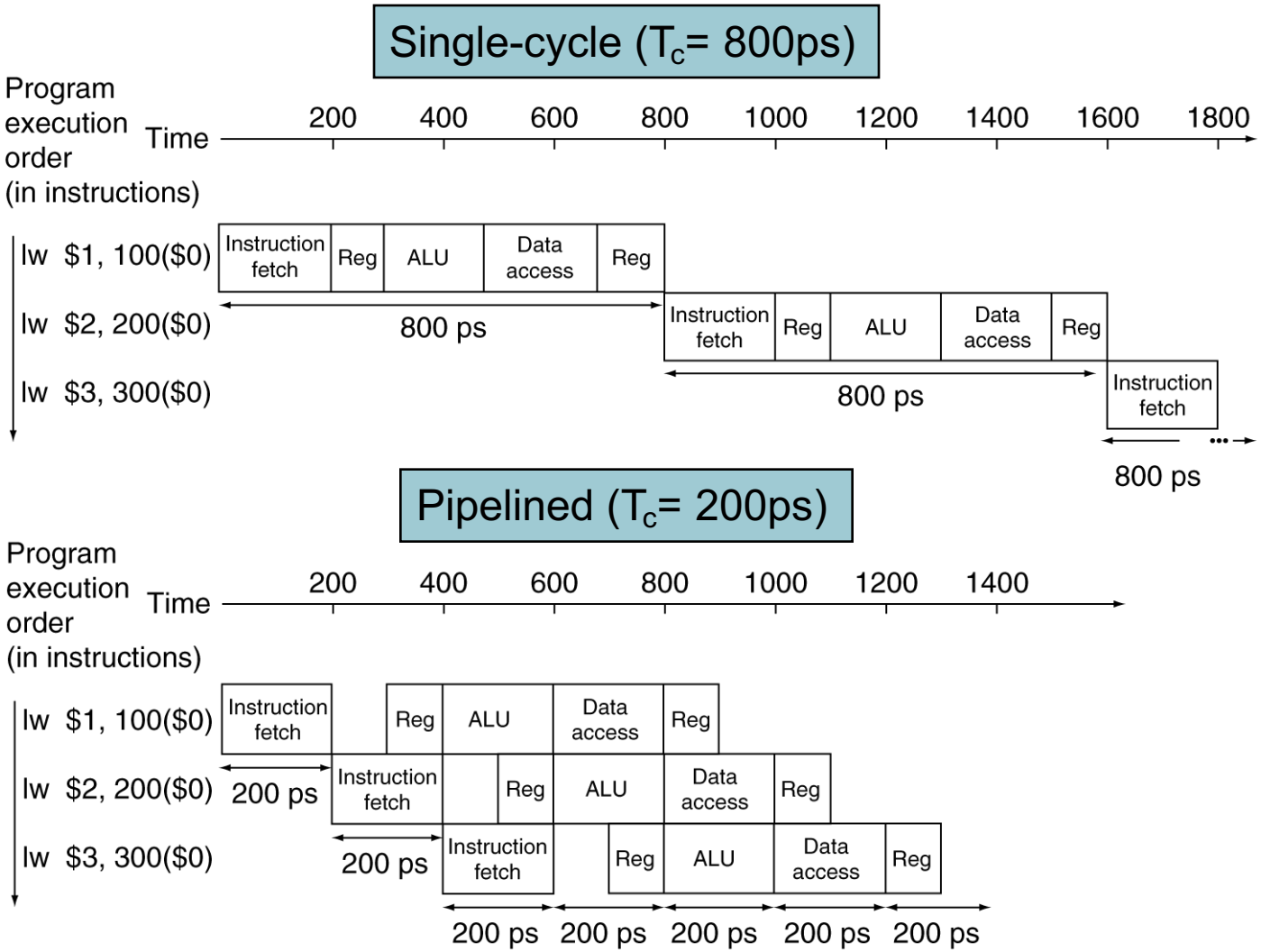
# Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

hazard error.

| 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$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- 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 17-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 prevent starting the next instruction in the next cycle
- Structure hazards ← hardware not there
  - A required resource is busy
- Data hazard ← next command still wait for output from prev command
  - Need to wait for previous instruction to complete its data read/write
- Control hazard ← similar to data hazard.
  - Deciding on control action depends on previous instruction

# Structure Hazards

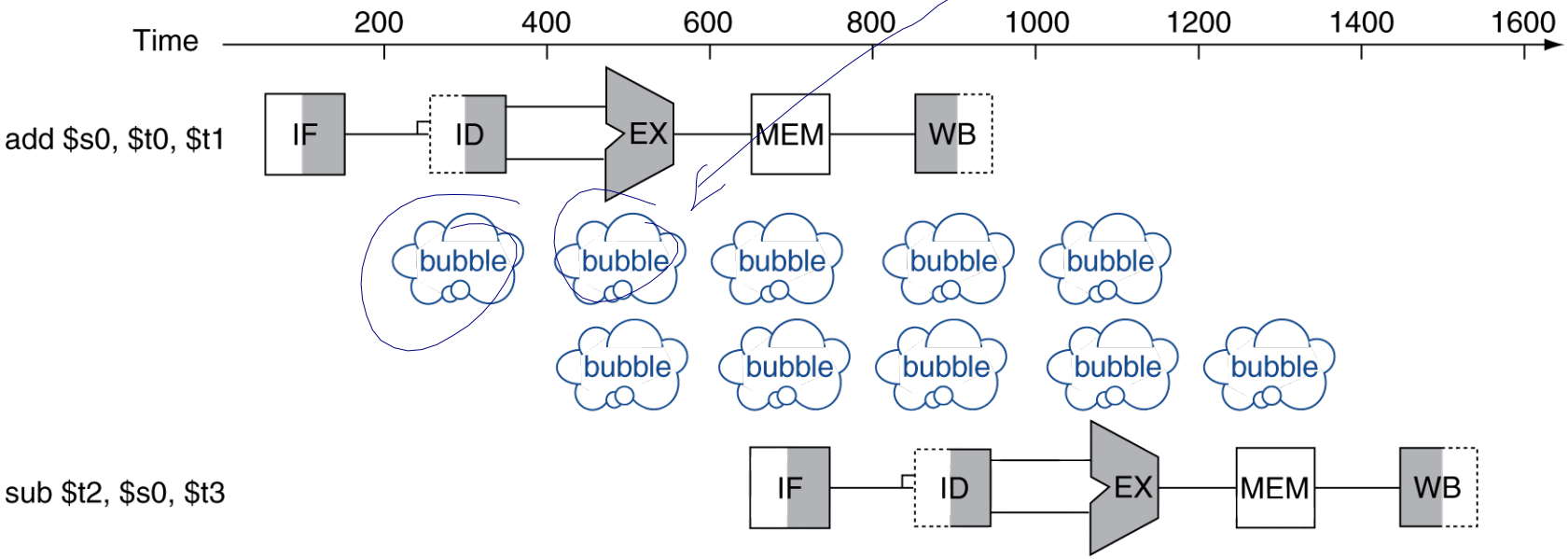
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"

wait for 1 cycle until res is ready

- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

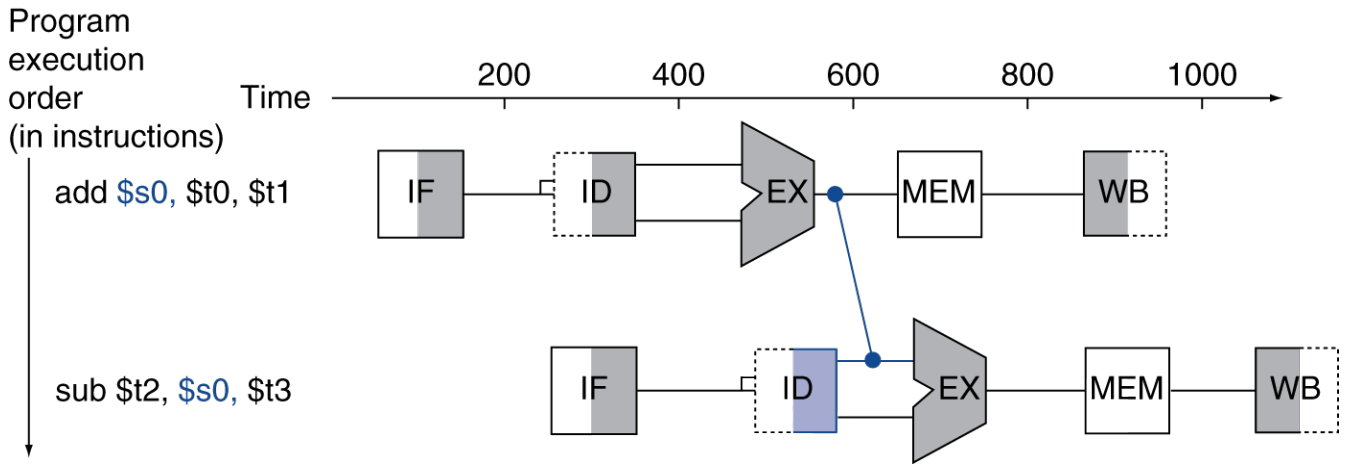
# Data Hazards

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



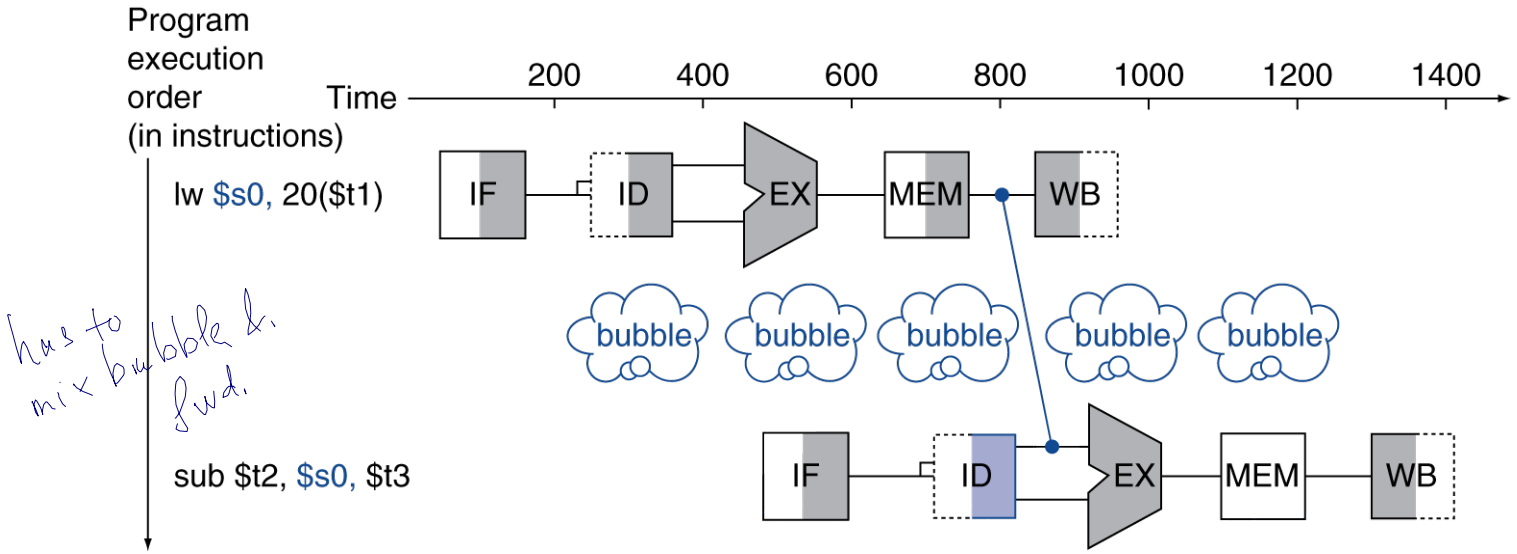
# Forwarding (aka Bypassing)

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



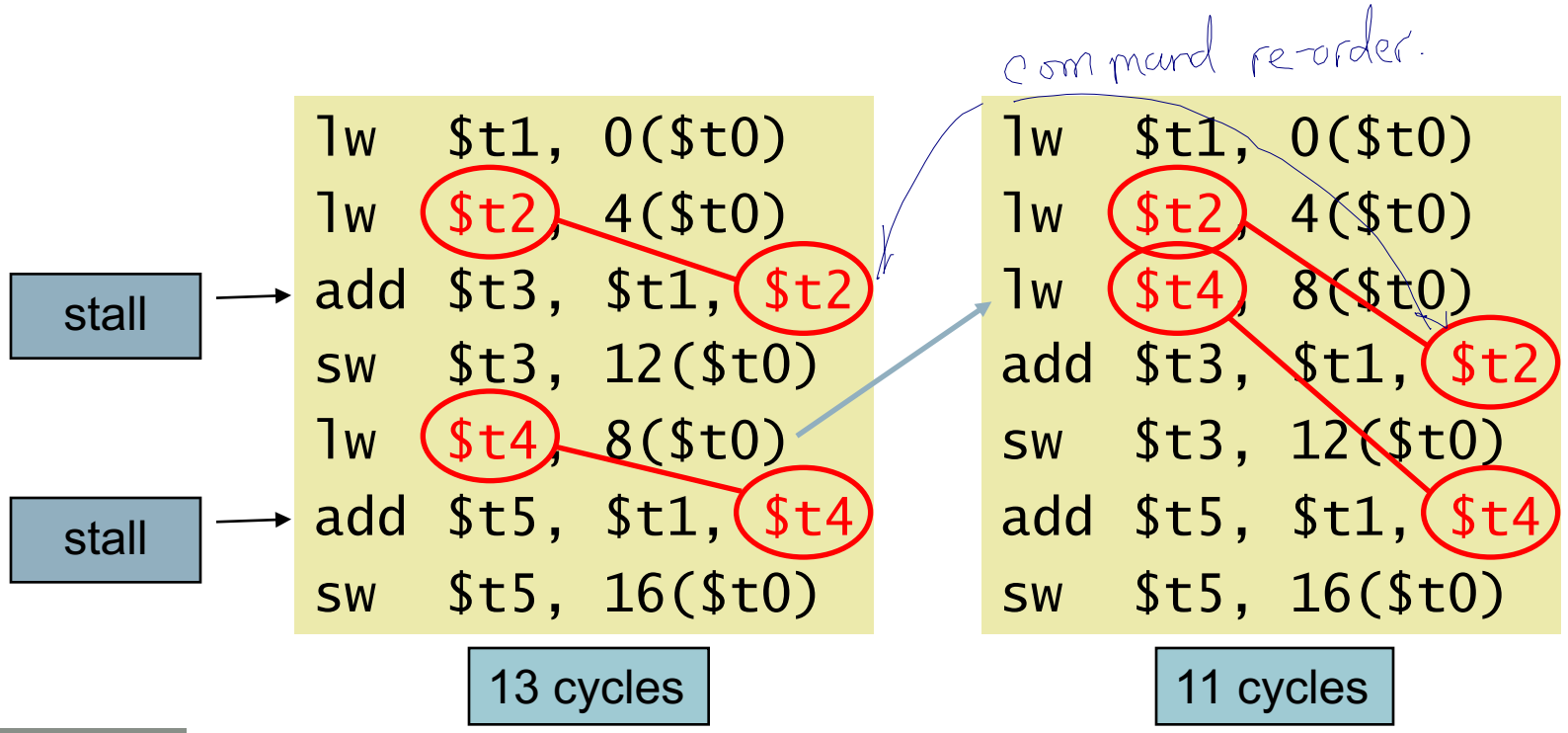
# Load-Use Data Hazard

- 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;`



# Structure Hazards

1. How many types do MIPS instructions formats have ?
2. What are the commons for all types MIPS instructions ?
3. What are the benefits for doing the pipelined operations for MIPS instructions ?
4. What are the general five stages for a MIPS instruction ?
5. Describe the datapath flow for MIPS instruction “lw”