# Chapter 4

## The Processor

# The Processor

- Introduction
- Logic Design Conventions
- Building a Datapath
- A Simple Implementation Scheme
- An Overview of Pipelining
- Pipeline Summary

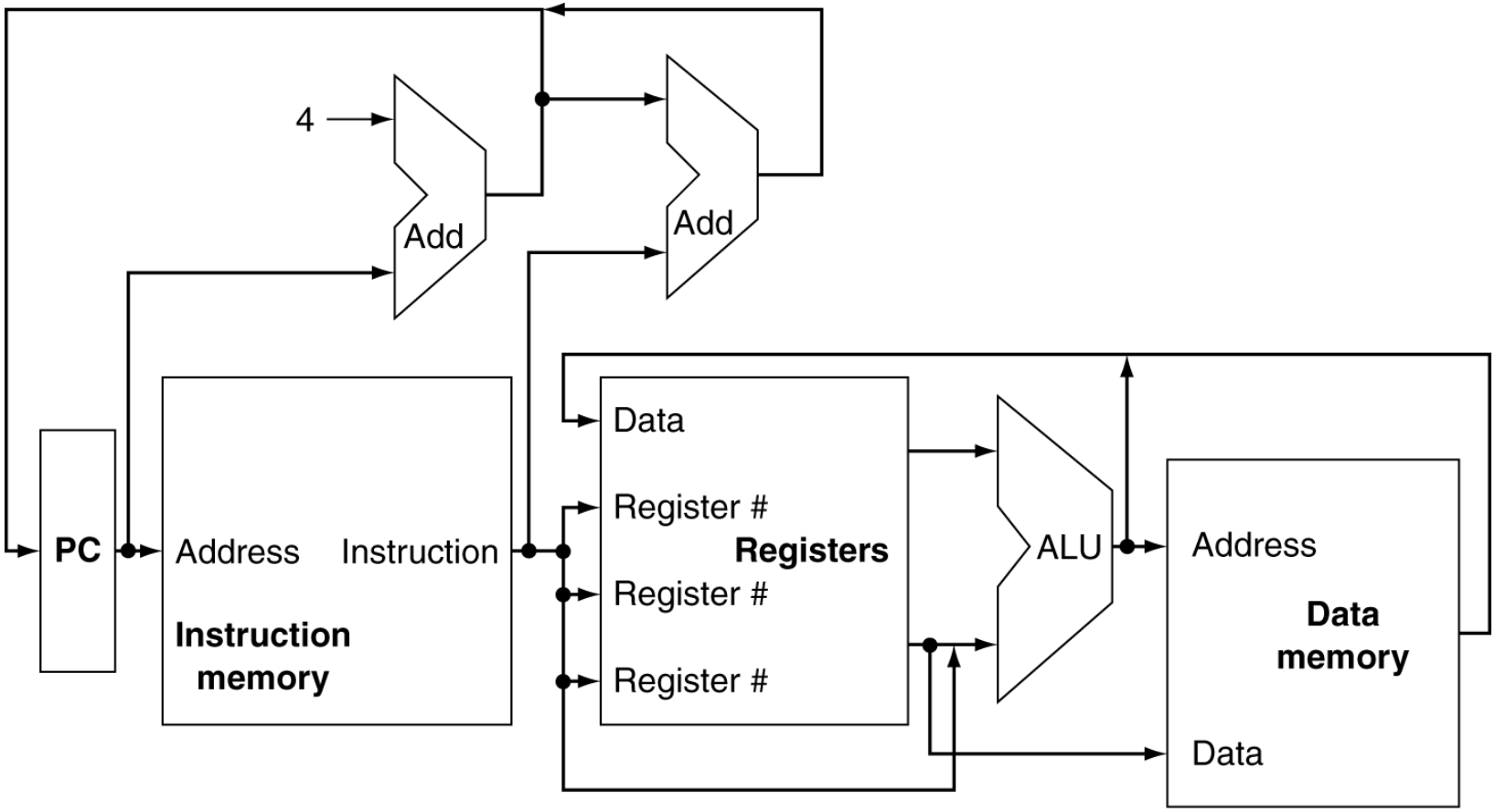
# Introduction

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

# Instruction Execution

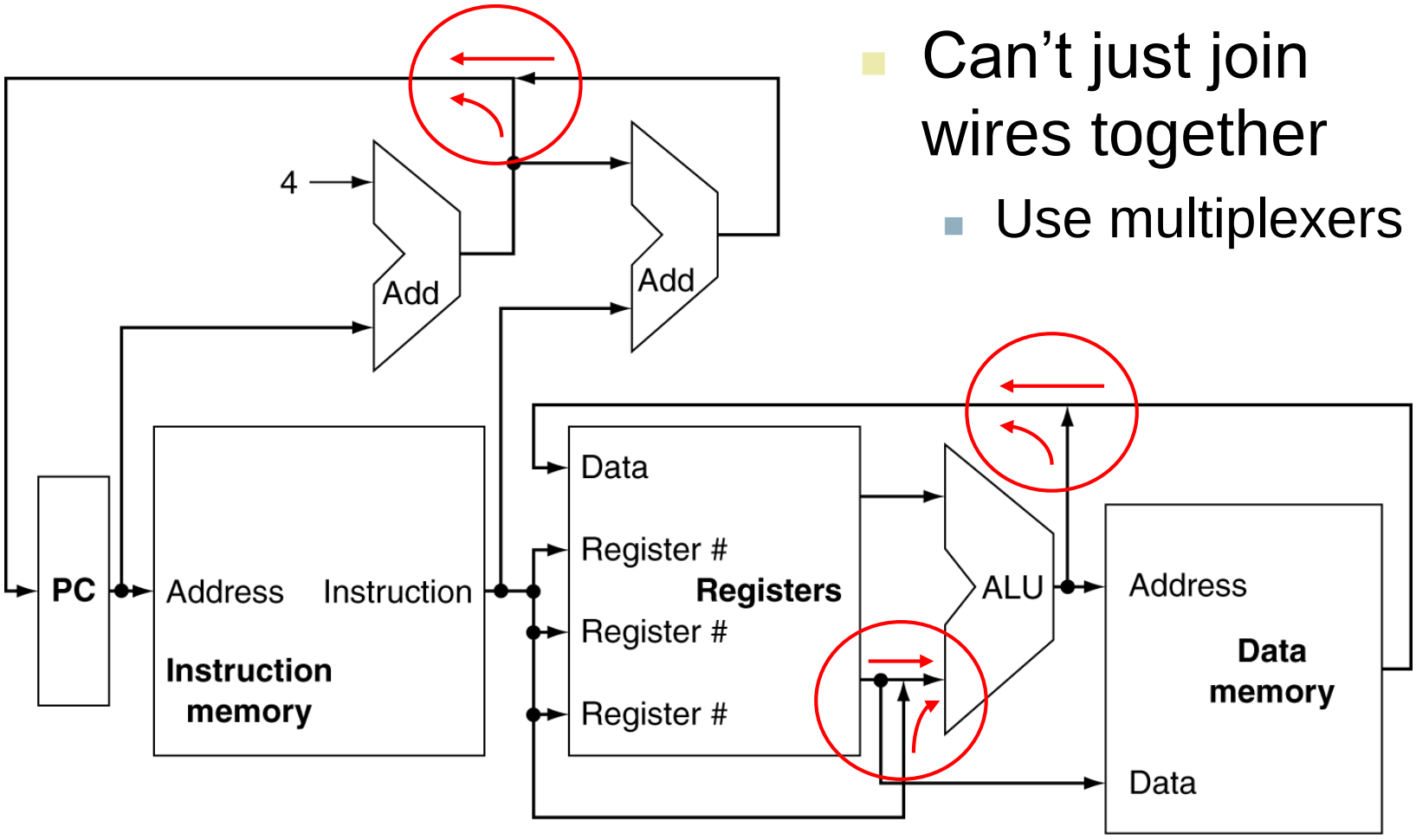
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address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

# CPU Overview



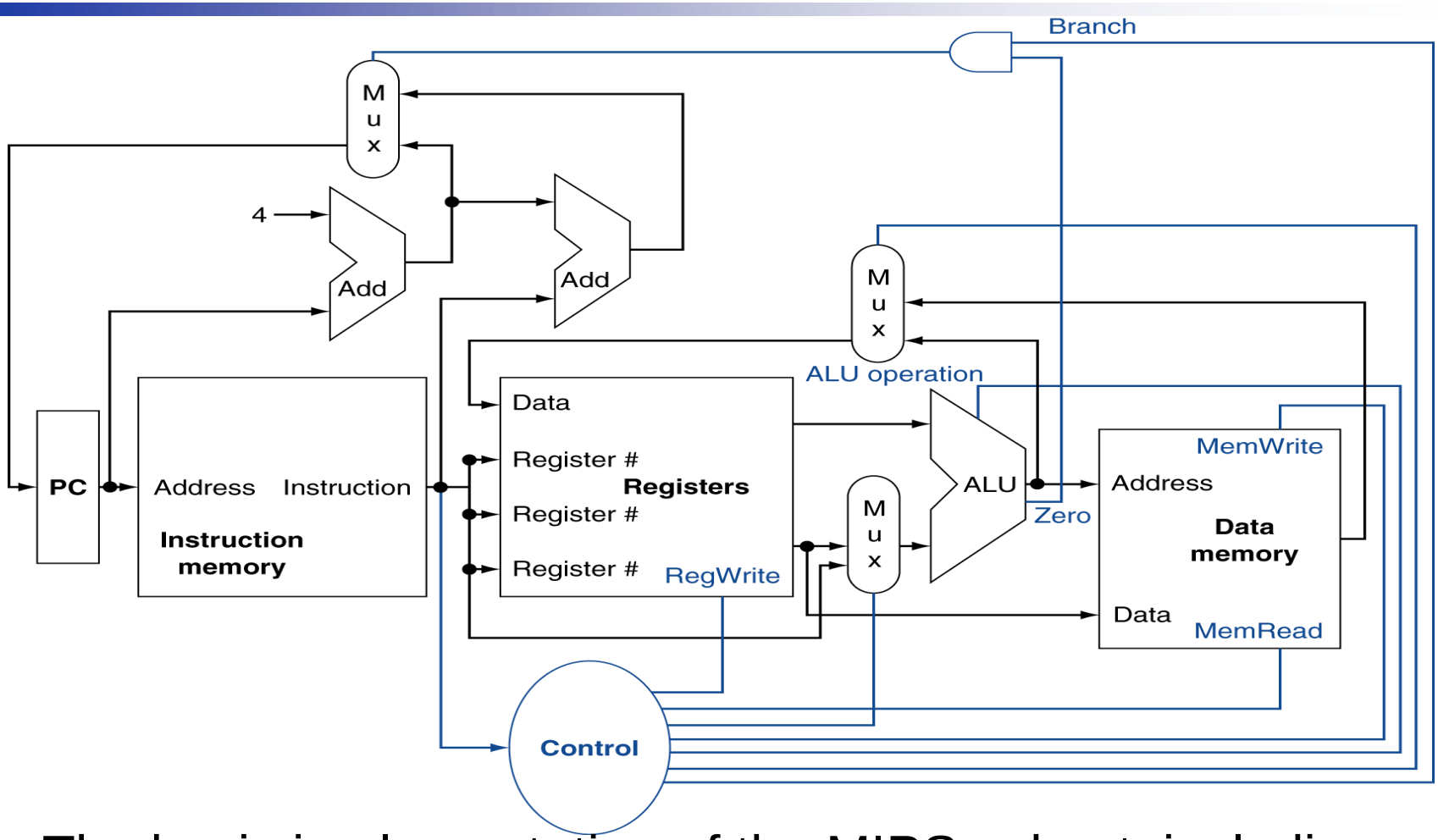

An abstract view of the implementation of the MIPS subset

# Multiplexers




- Can't just join wires together
  - Use multiplexers

# Control




The basic implementation of the MIPS subset, including the necessary multiplexors and control lines

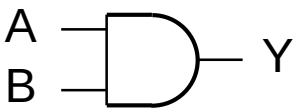
# Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (Sequential) elements
  - Store information

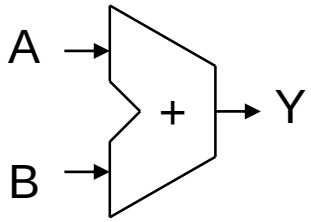
# Combinational Elements



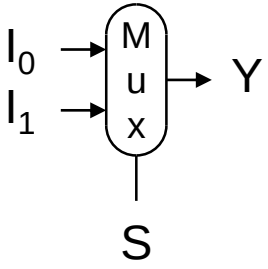
- AND-gate
  - Y = A & B



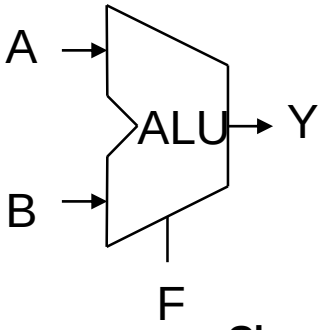

- Adder
  - Y = A + B




- Multiplexer
  - Y = S ? $I_1$ : $I_0$

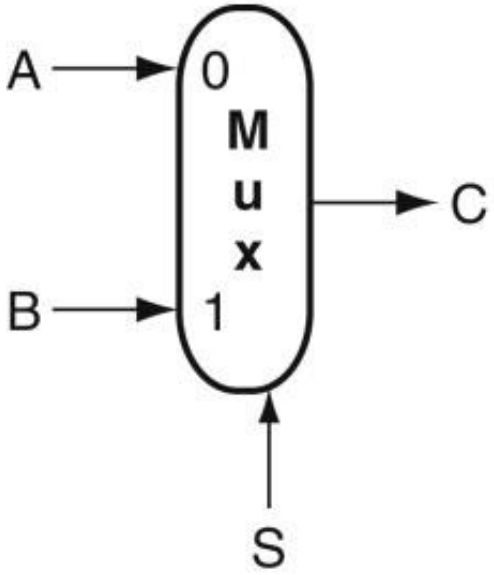


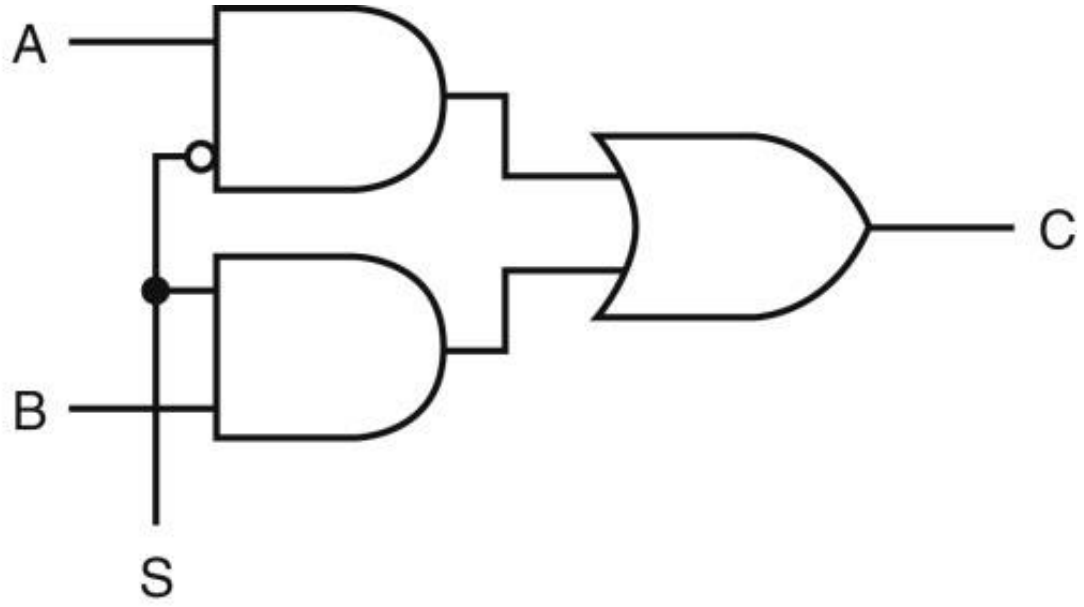

- Arithmetic/Logic Unit
  - Y = F(A, B)



# Multiplexors(1)



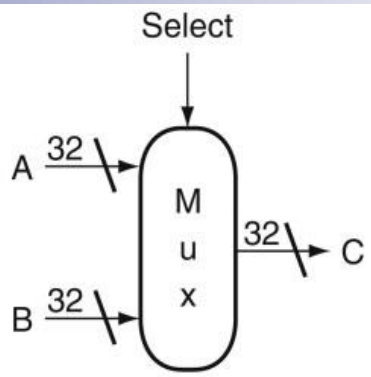


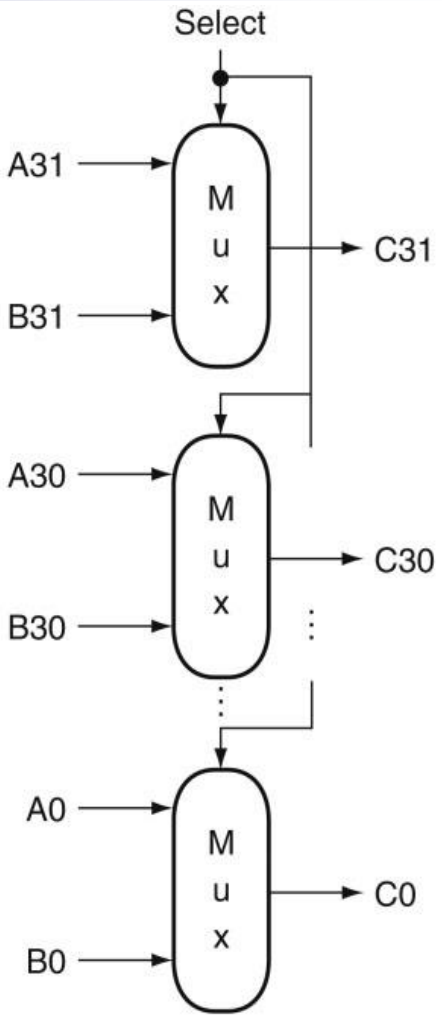

A two-input multiplexor has two data inputs (*A* and *B*) labeled *0* and *1*, one selector input (*S*), and an output *C*.

# Multiplexors(2)

- A multiplexor is arrayed 32 times to perform a selection between two 32-bit inputs.
- One data selection signal used for all 32 1-bit multiplexors.



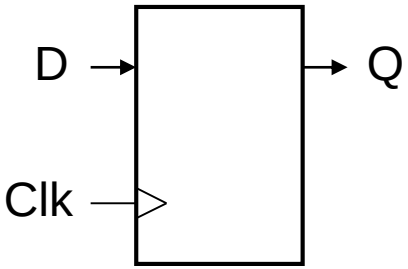

a. A 32-bit wide 2-to-1 multiplexor

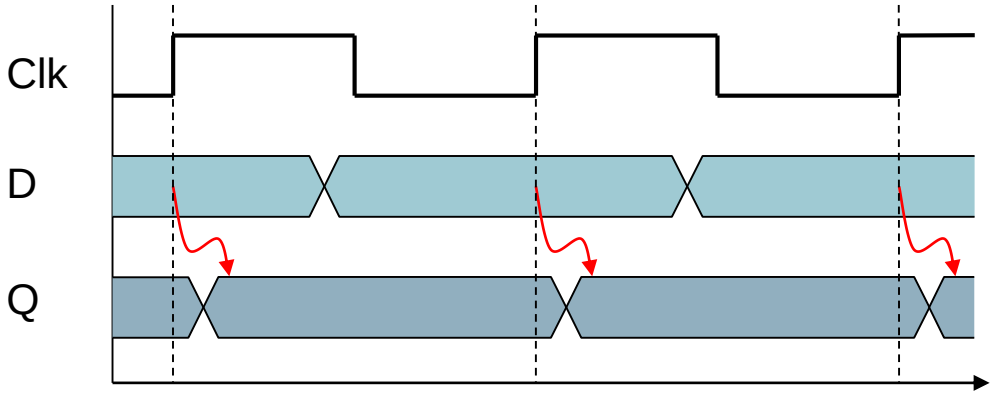



b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

# Sequential Elements

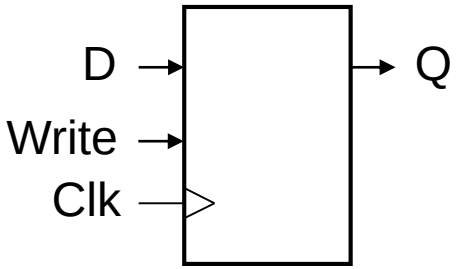
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

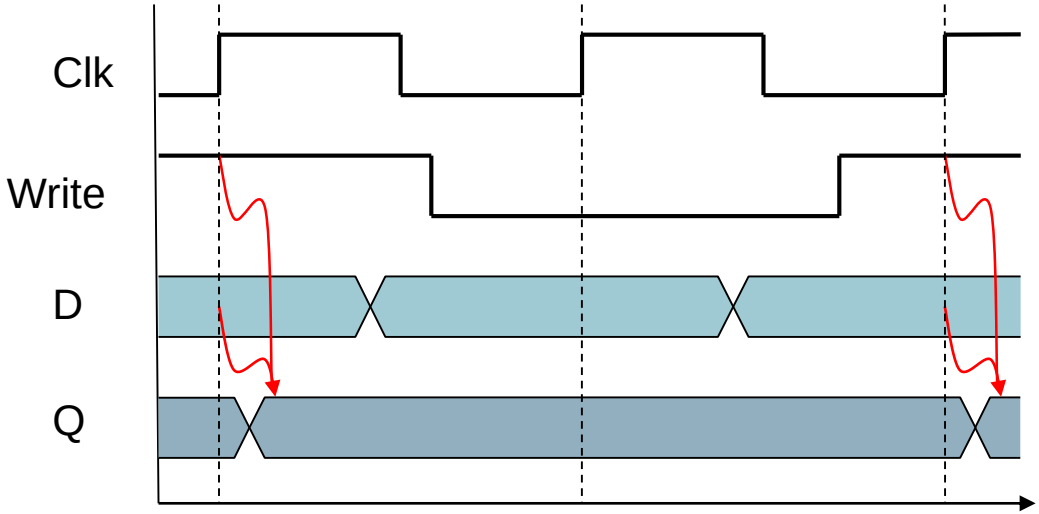


# Sequential Elements (2)

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

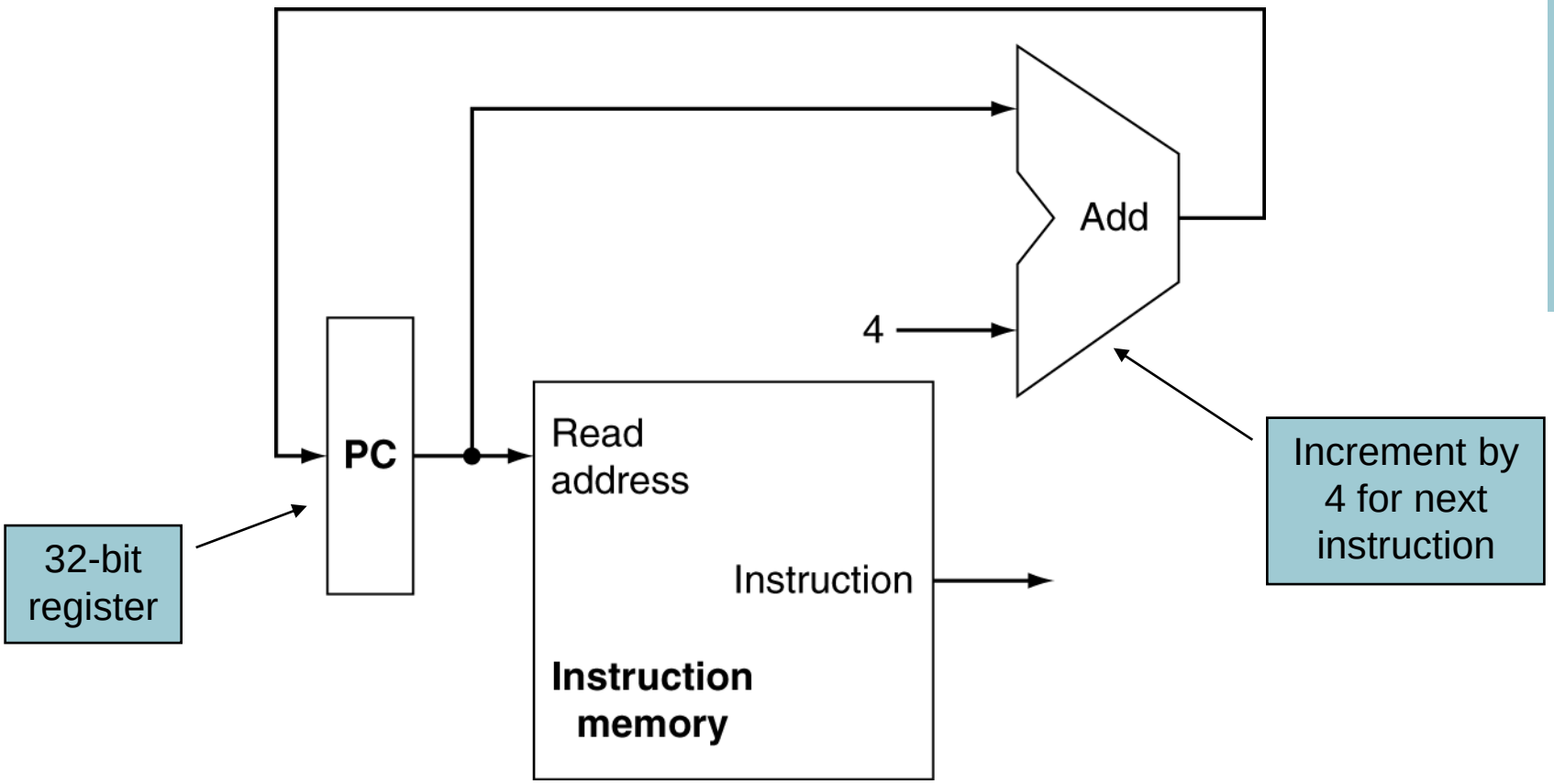


# Building a Datapath

- Datapath
  - 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

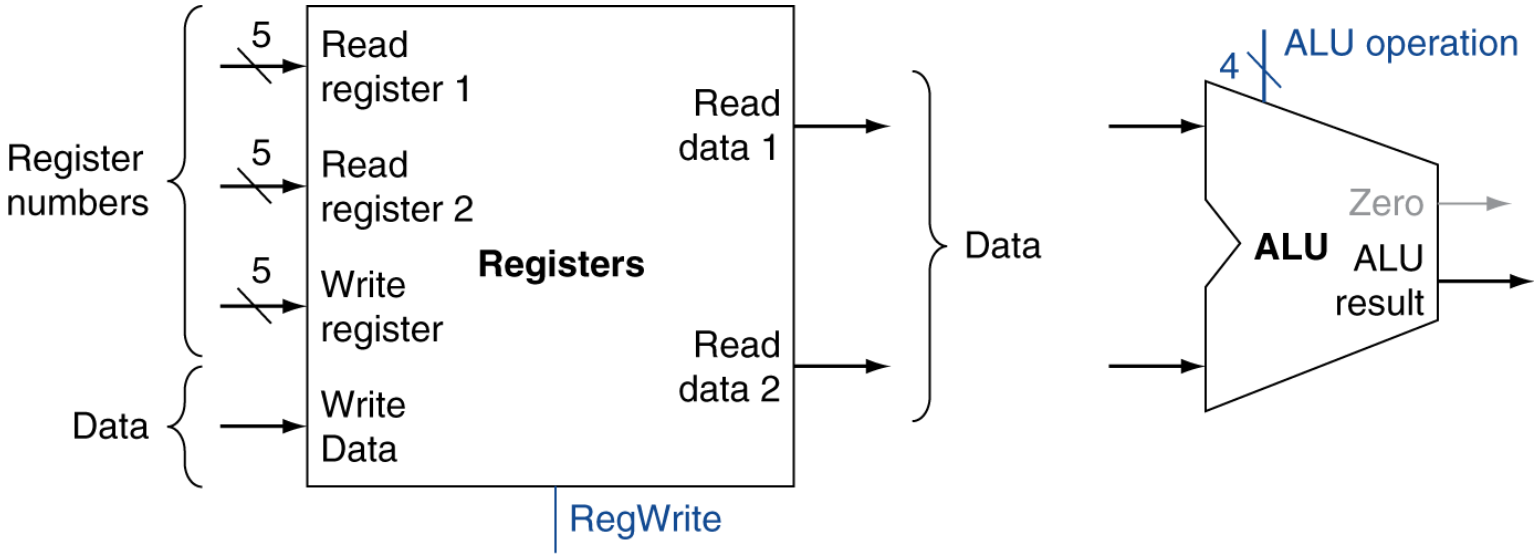
# Instruction Fetch




A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

# R-Format Instructions

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

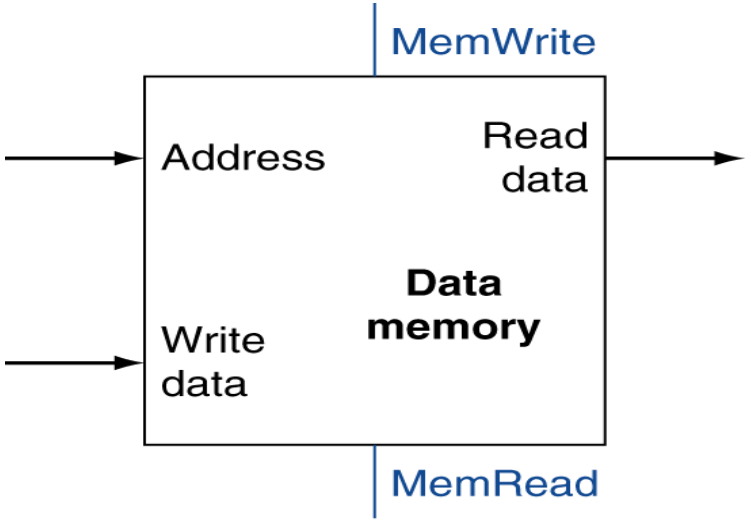



a. Registers

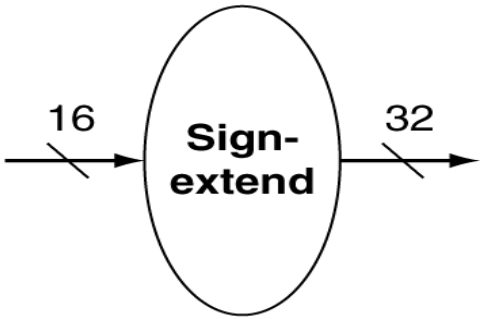
b. ALU

# Load/Store Instructions

- Read register operands
- Calculate address using ALU: 32-bit register and 16-bit sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory




a. Data memory unit

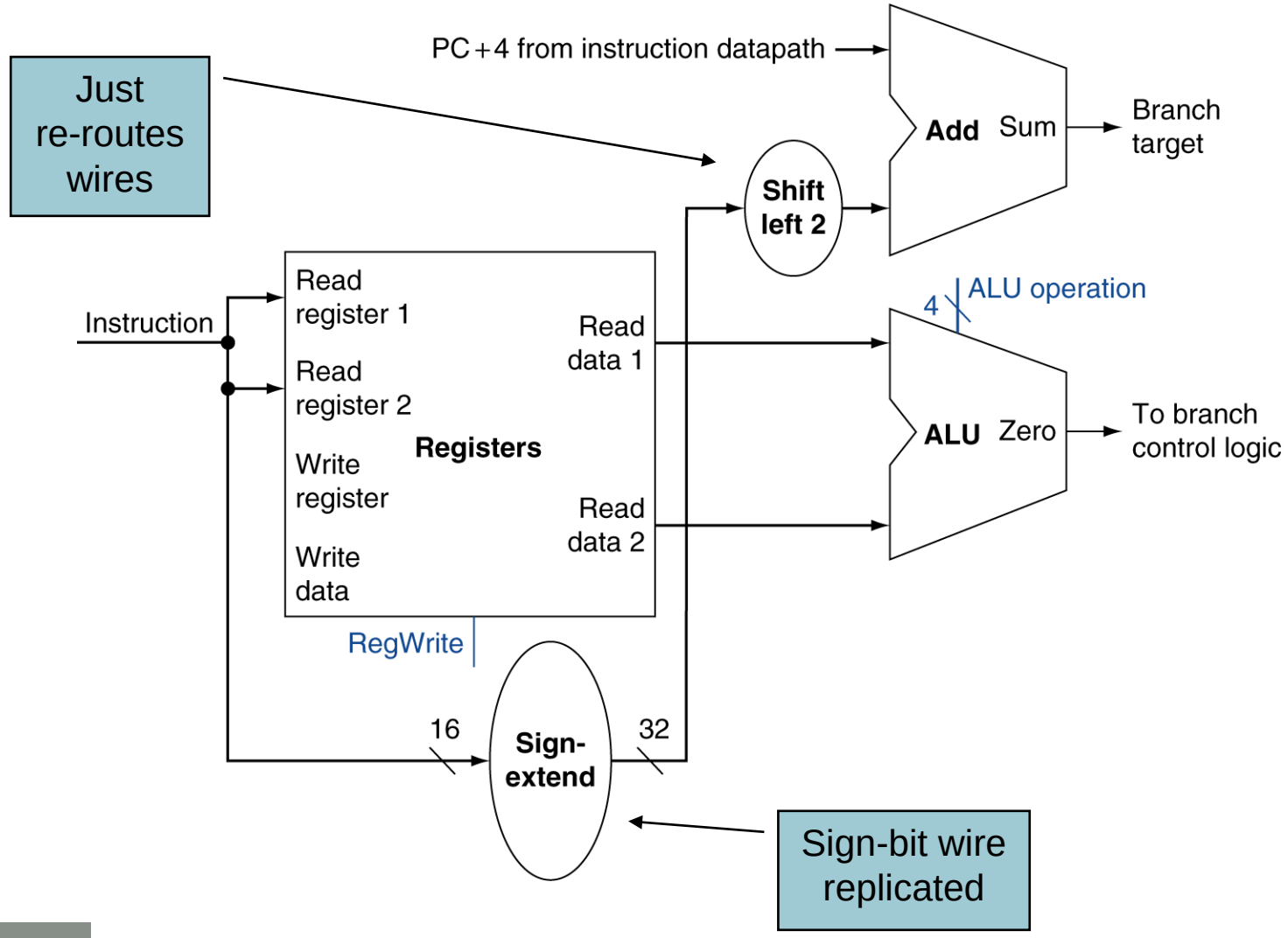



b. Sign extension unit

# 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 (2)

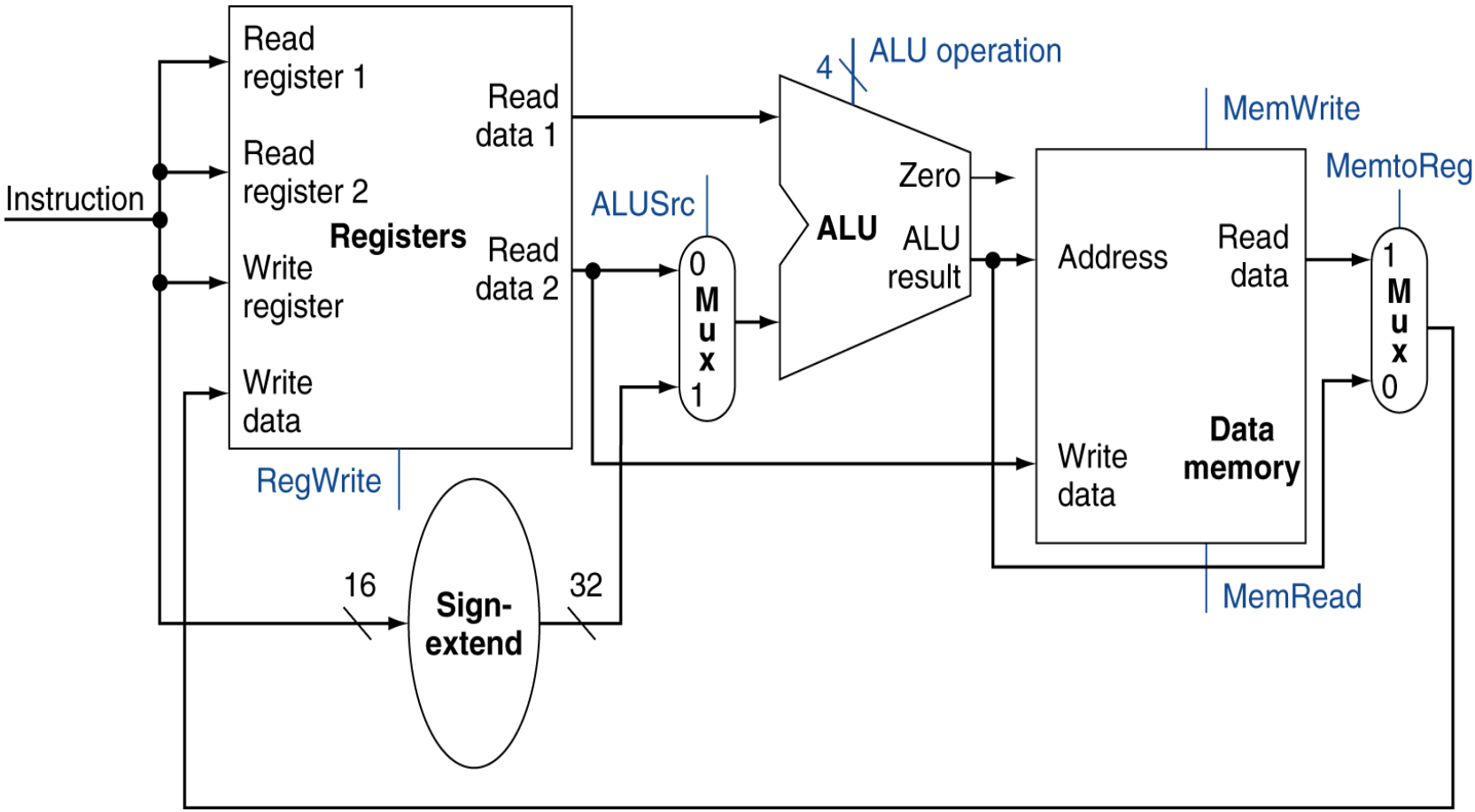


# Composing the Elements

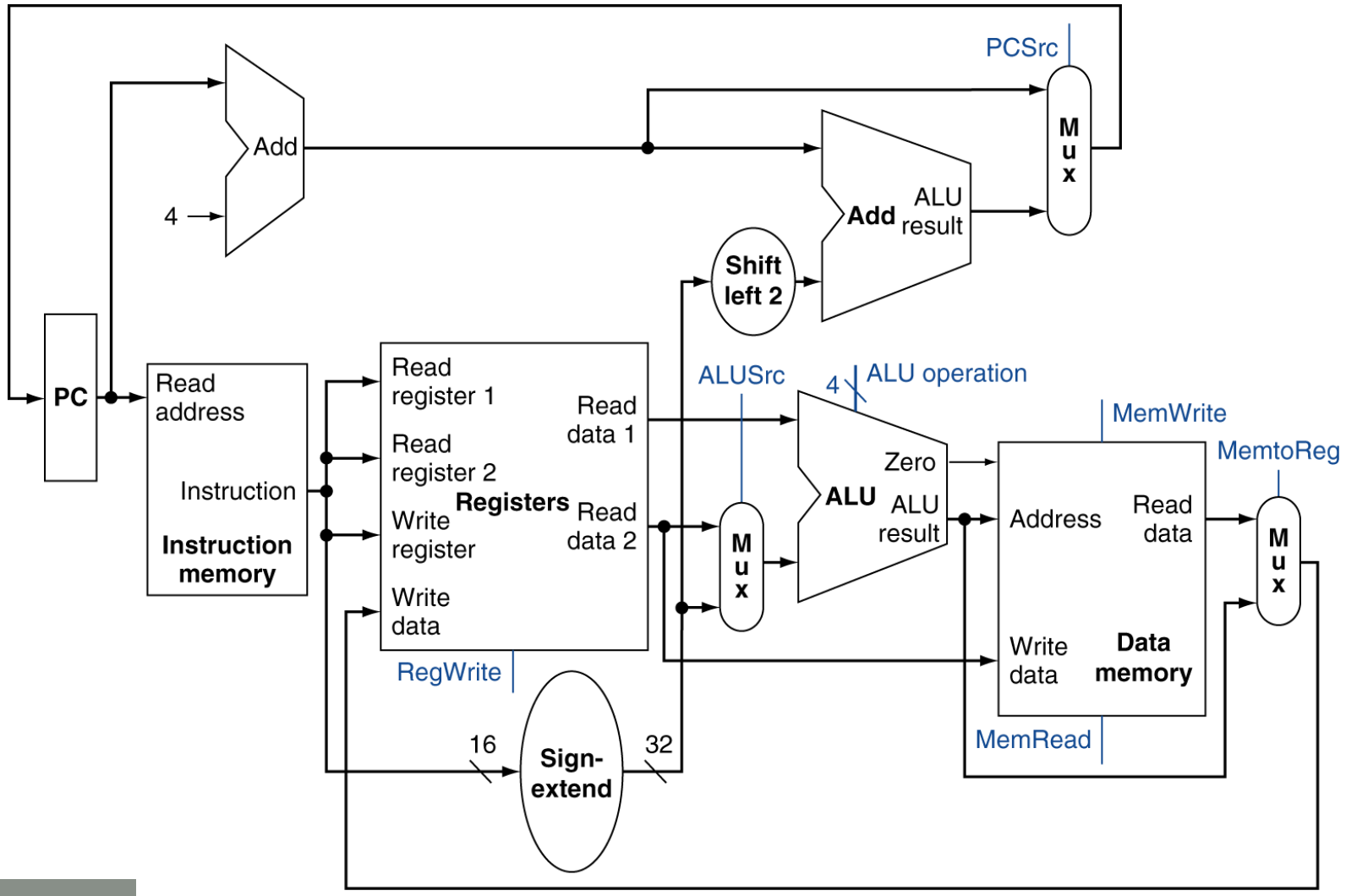
- Simple datapath does one instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

# 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

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

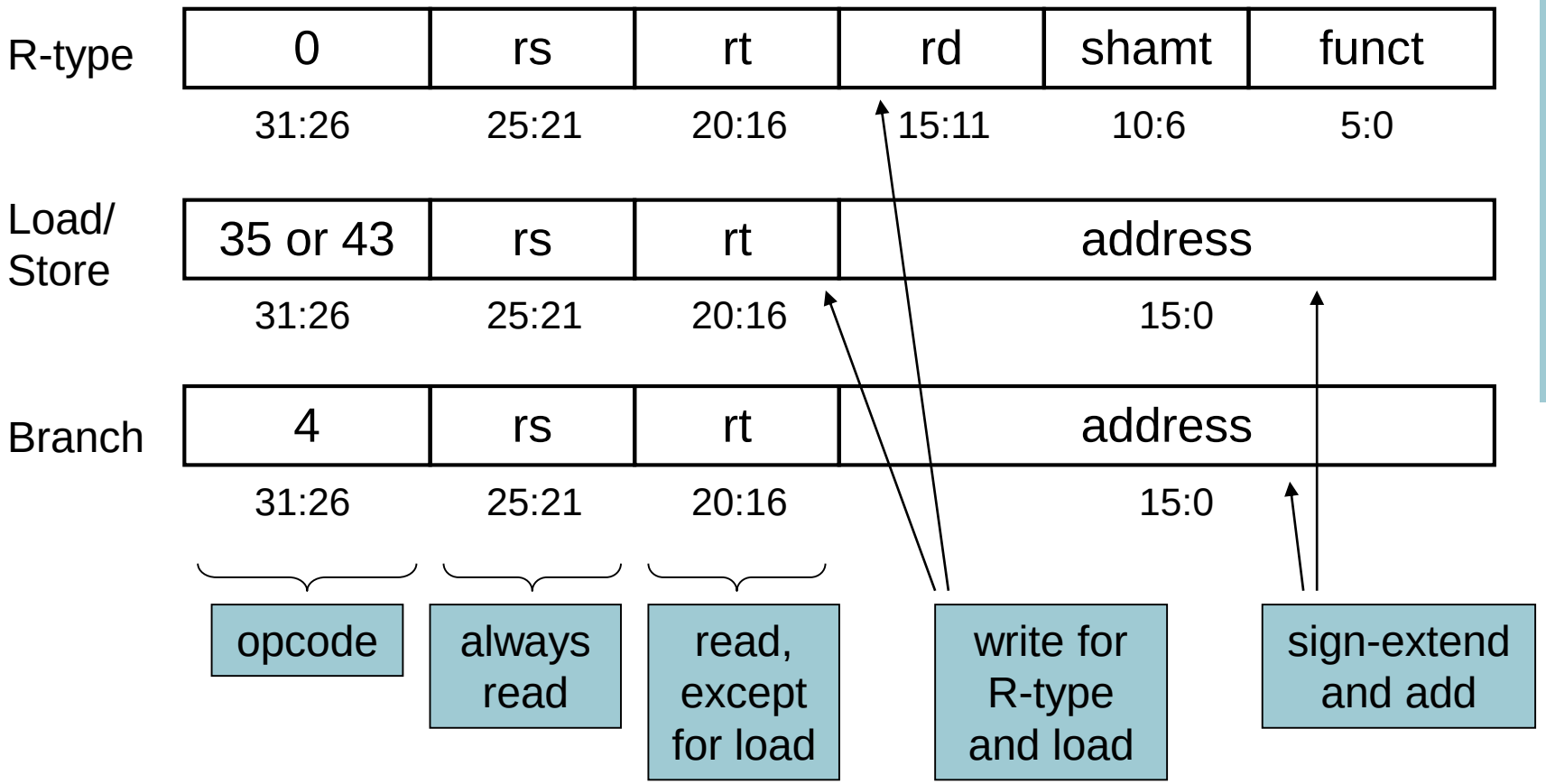
# ALU Control (2)

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

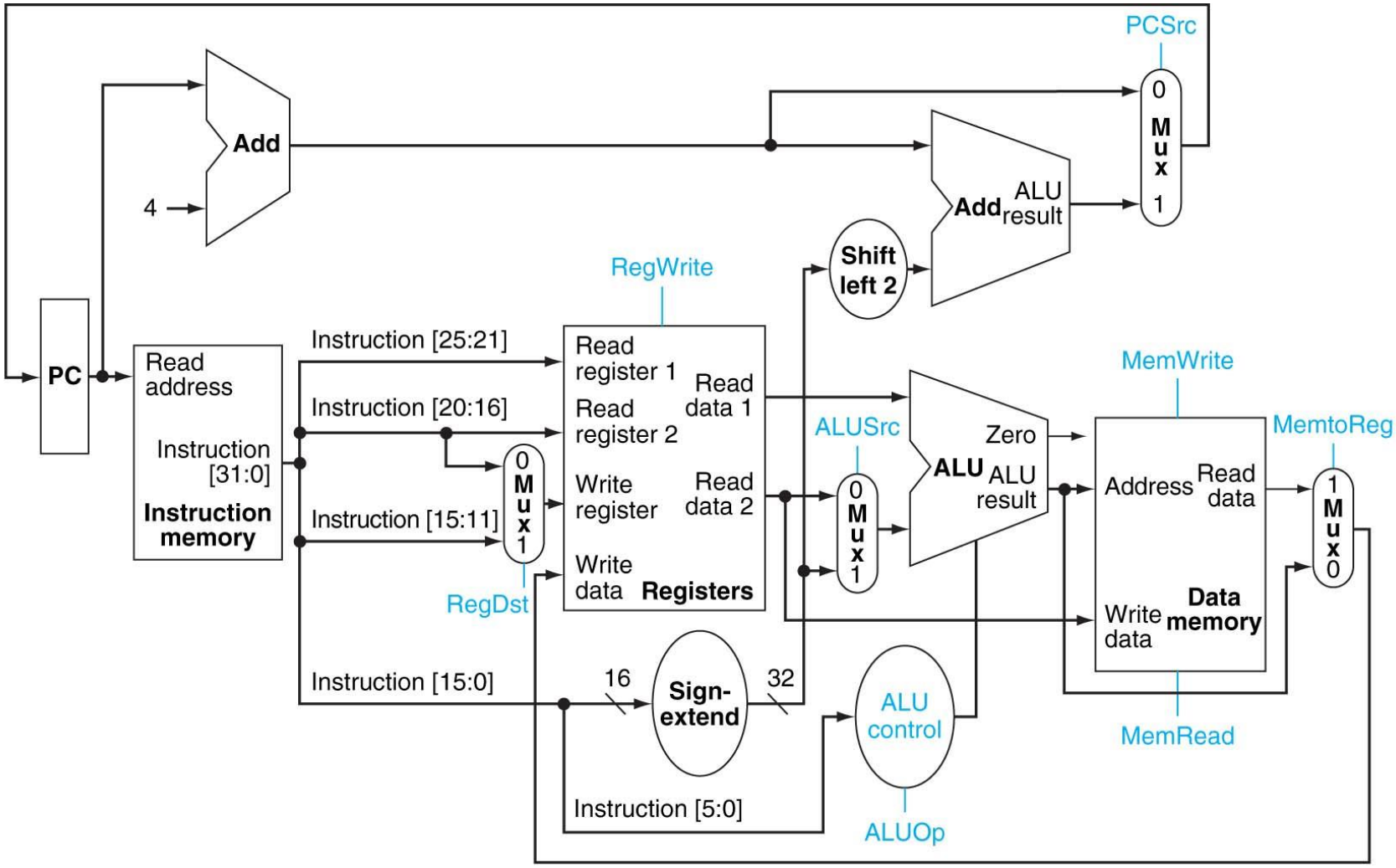
| opcode | ALUOp | Operation | funct | ALU function | ALU control |
|---|---|---|---|---|---|
| lw | 00 | load word | XXXXXX | add | 0010 |
| sw | 00 | store word | XXXXXX | add | 0010 |
| beq | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| | | subtract | 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



# The Main Control Unit (2)

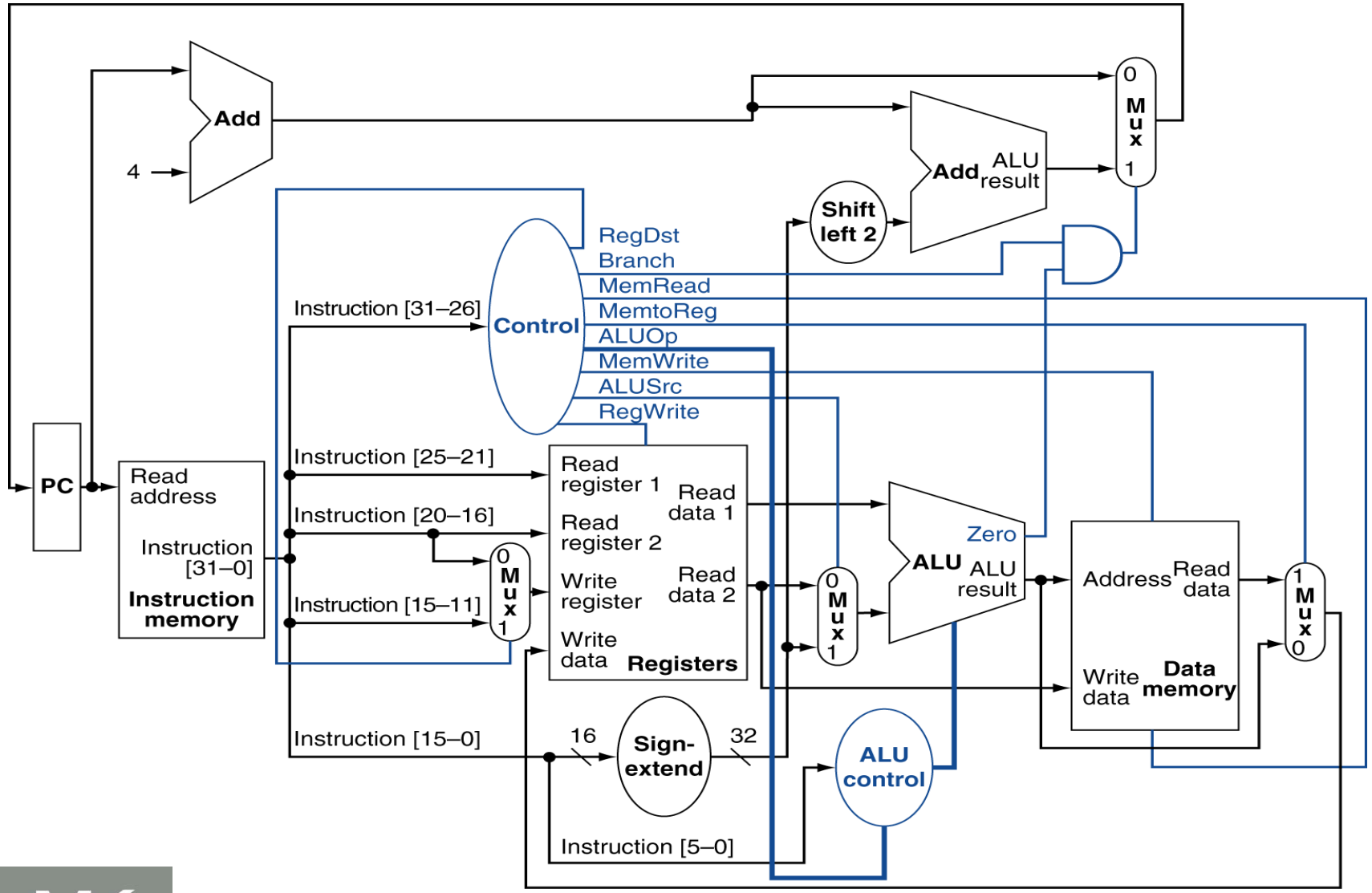


# The Main Control Unit (3)



| Signal name | Effect when deasserted | Effect when asserted |
|---|---|---|
| RegDst | The register destination number for the Write register comes from the rt field (bits 20:16). | The register destination number for the Write register comes from the rd field (bits 15:11). |
| RegWrite | None. | The register on the Write register input is written with the value on the Write data input. |
| ALUSrc | The second ALU operand comes from the second register file output (Read data 2). | The second ALU operand is the sign-extended, lower 16 bits of the instruction. |
| PCSrc | The PC is replaced by the output of the adder that computes the value of PC + 4. | The PC is replaced by the output of the adder that computes the branch target. |
| MemRead | None. | Data memory contents designated by the address input are put on the Read data output. |
| MemWrite | None. | Data memory contents designated by the address input are replaced by the value on the Write data input. |
| MemtoReg | The value fed to the register Write data input comes from the ALU. | The value fed to the register Write data input comes from the data memory. |

# Datapath With Control



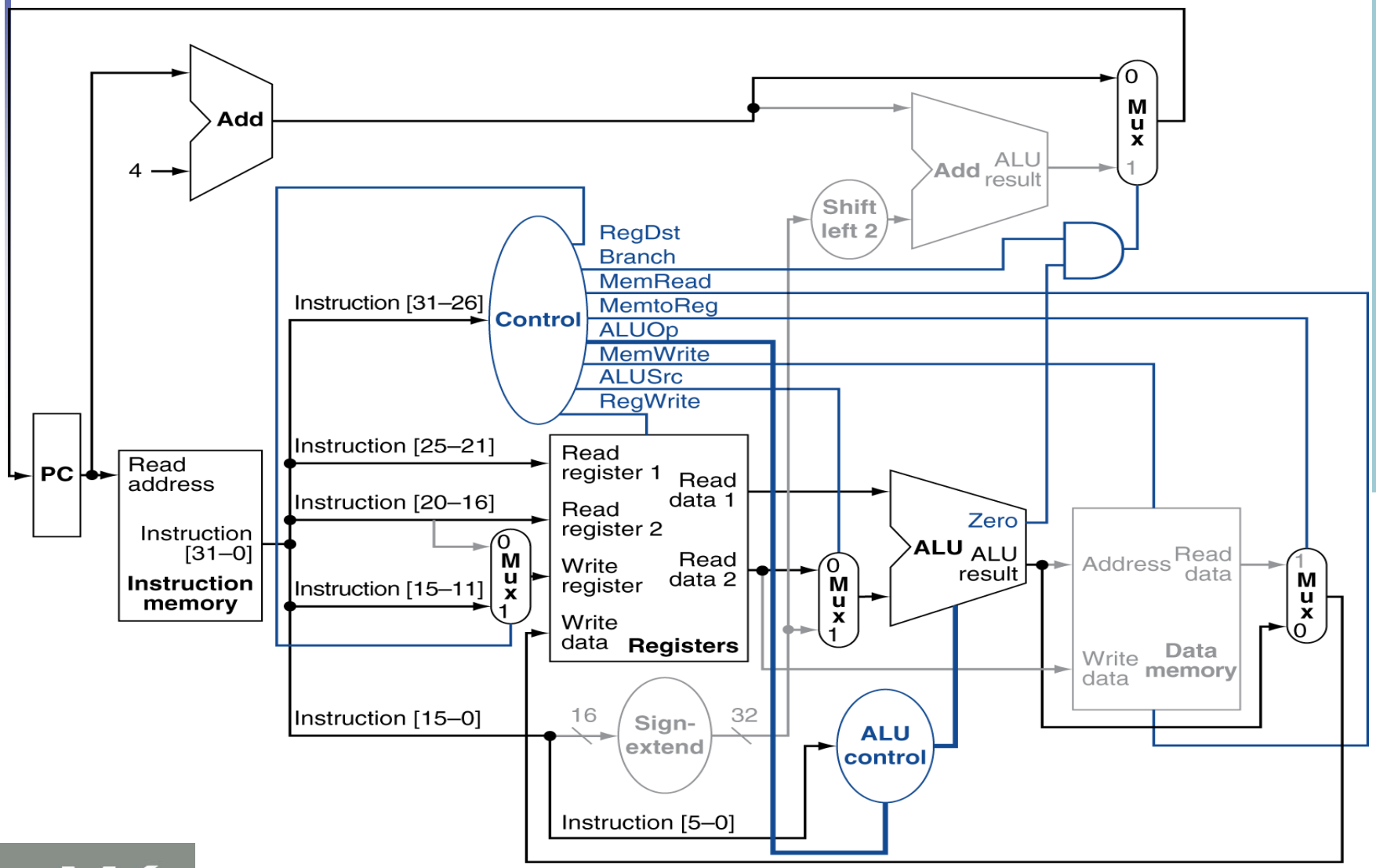
# Datapath With Control (2)



| Instruction | RegDst | ALUSrc | Memto-Reg | Reg-Write | Mem-Read | Mem-Write | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

- The setting of the control lines is completely determined by the opcode fields of the instruction
- The first row of the table corresponds to the R-format instructions (add, sub, AND, OR, and slt). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. R-type instruction writes a register (RegWrite=1) but neither reads nor writes data memory
- When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high
- The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field

# R-Type Instruction



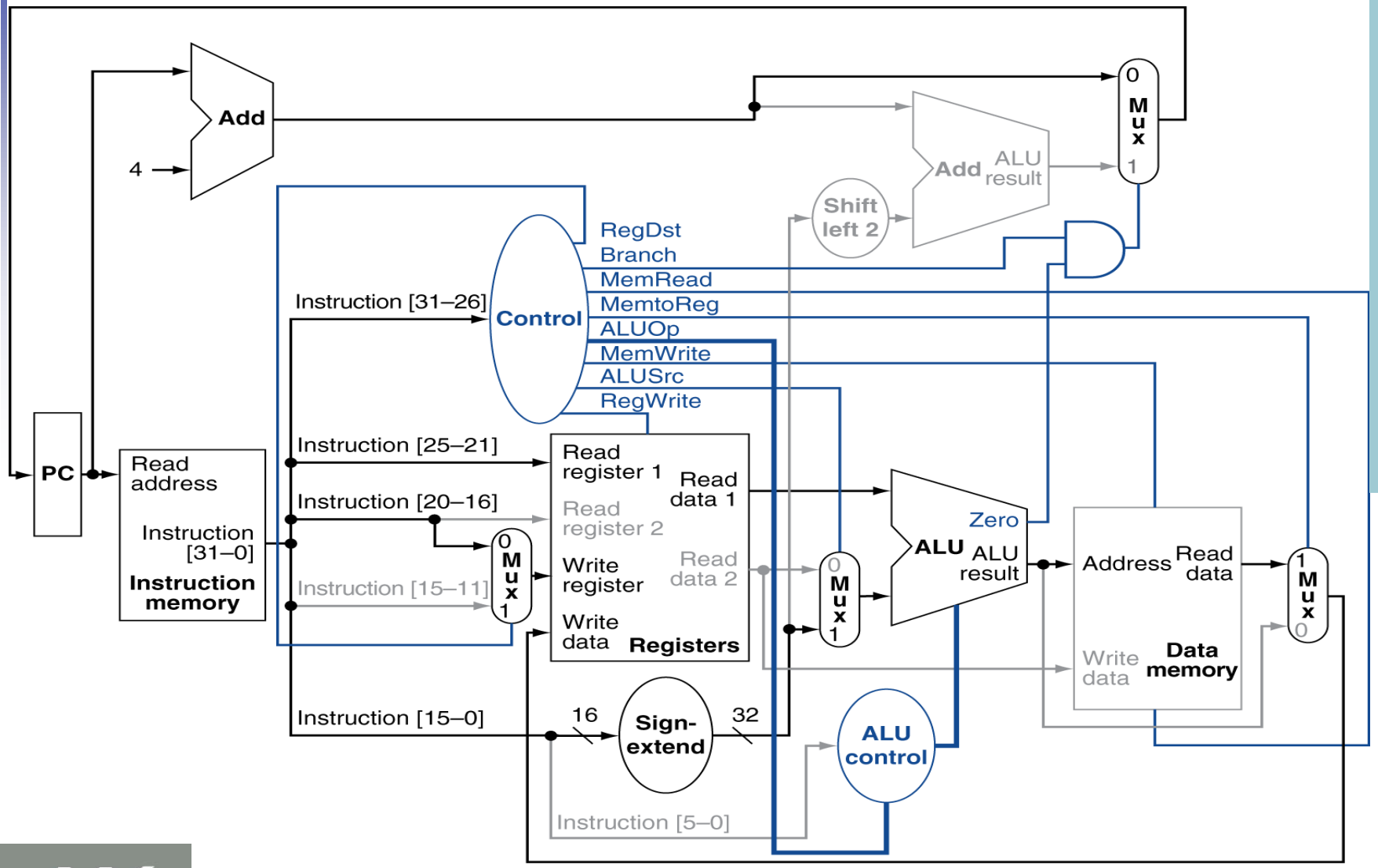
# R-Type Instruction (2)

For example, add $t1, $t2,$t3

Four steps to execute the instruction in one clock cycle

1. The instruction is fetched, and the PC is incremented
2. Registers $t2 and $t3 are read from the register file. Also, the main control unit computes the setting of the control lines during this step
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, funct field) to generate the ALU function
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1)

# Load Instruction



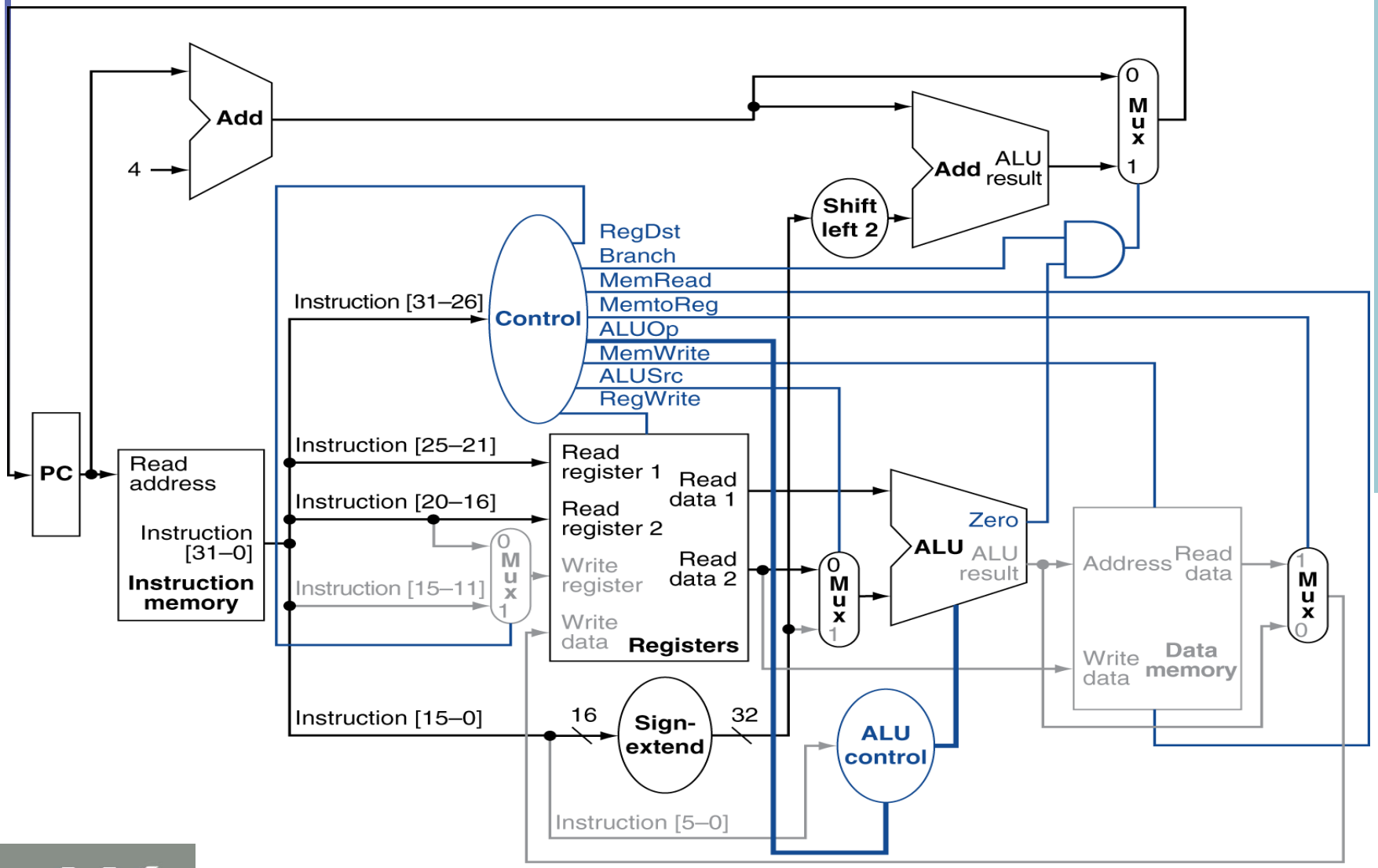
# Load Instruction (2)

For example, lw $t1, offset($t2)

Five steps to execute the instruction in one clock cycle

1. Instruction is fetched from the instruction memory and PC is Incremented
2. Register ($t2) value is read from the register file
3. ALU computes the sum of the value read from register file and sign-extended offset
4. Sum from ALU is used as the address for the data memory
5. Data from the memory unit is written into register file. Register destination is given by bits 20:16 of the instruction ($t1)

# Branch-on-Equal Instruction



# Branch-on-Equal Instruction(2

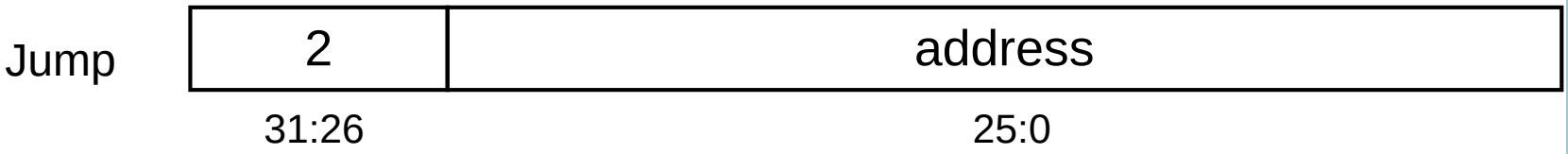


For example, beq $t1, $t2, offset

Four steps to execute the instruction in one clock cycle

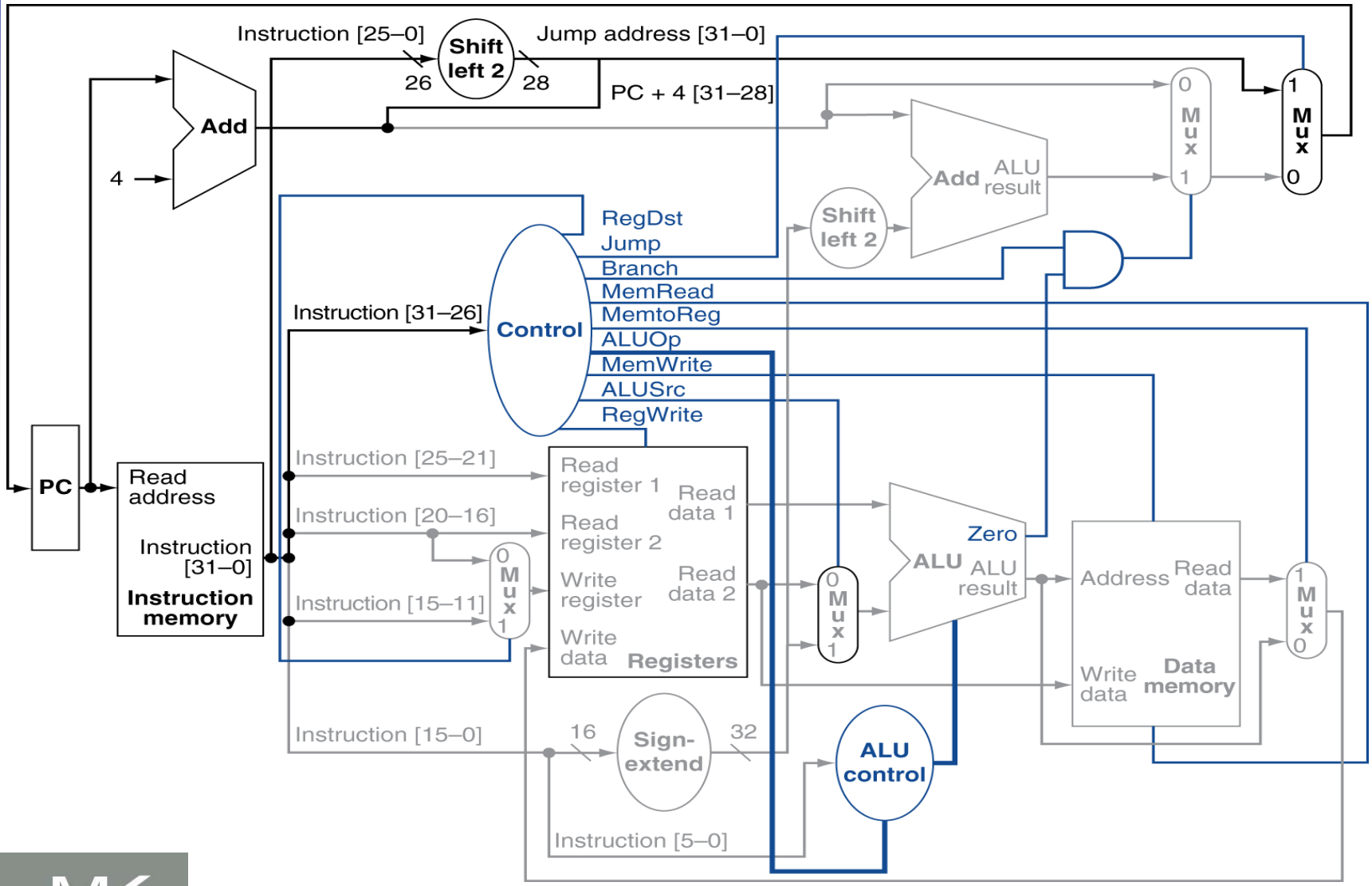
1. An instruction is fetched from instruction memory and PC is incremented
2. Registers $t1 and $t2 are read from the register file
3. ALU performs a subtract on the data values read from the register file. PC + 4 is added to sign-extended offset shifted left by two; result is branch target address
4. Zero result from ALU is used to decide which adder result to store into PC

# Implementing Jumps




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

# Datapath With Jumps Added

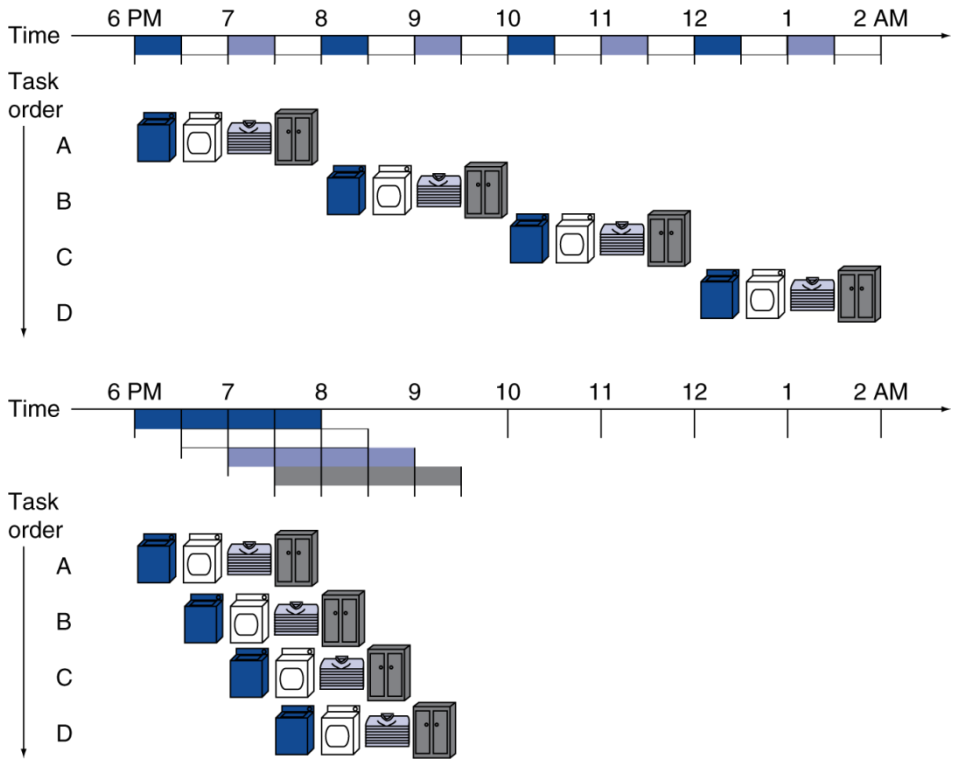


# Performance Issues

- 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 = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/[0.5(n-1)+2] ≈ 4 = number of stages

# MIPS Pipeline

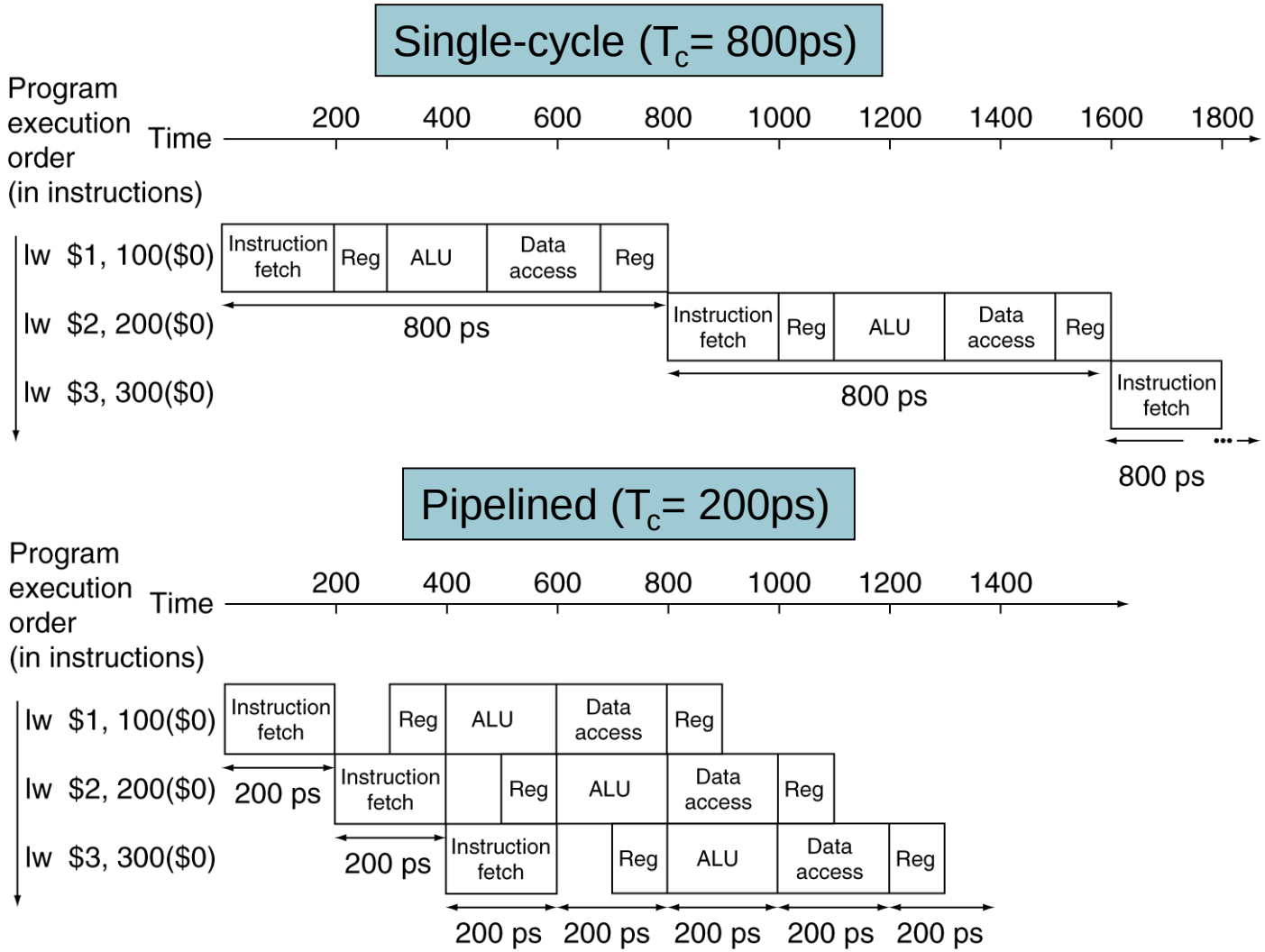
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

# Pipeline Performance

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

| 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$_{pipelined}$
    = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$ = 160ps
- If not balanced, speedup is less
  - = i.e., 800ps/200ps = 4
- 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 15-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

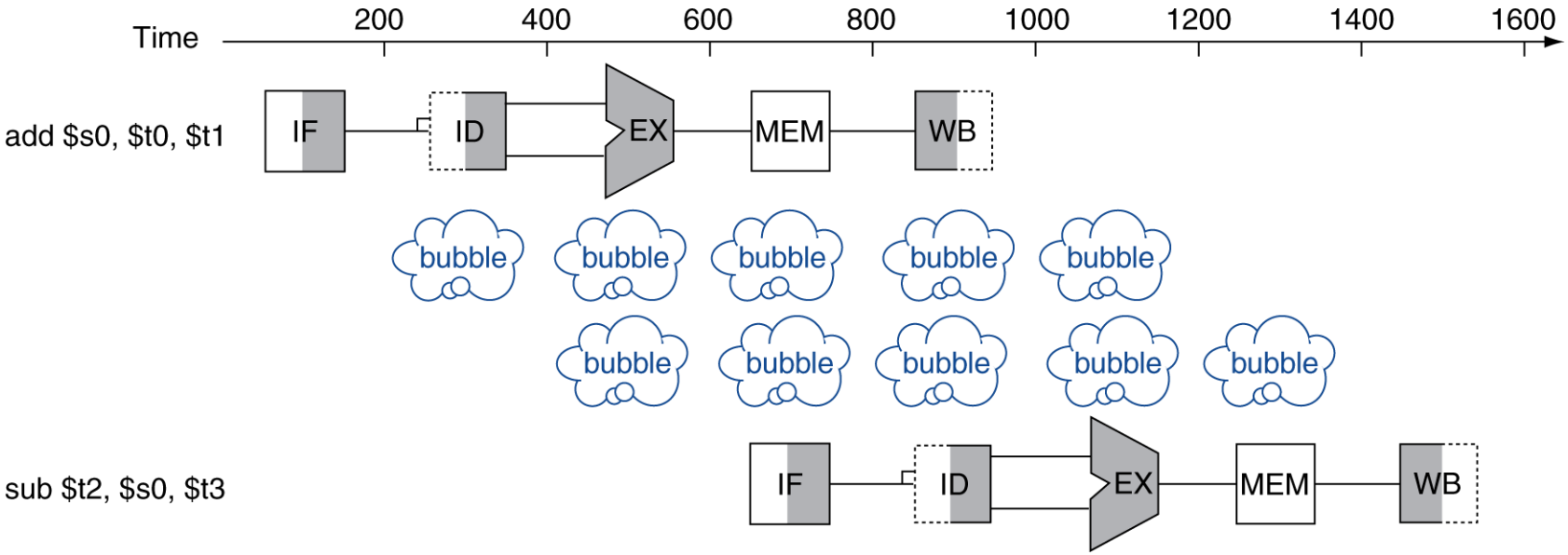
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

# 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

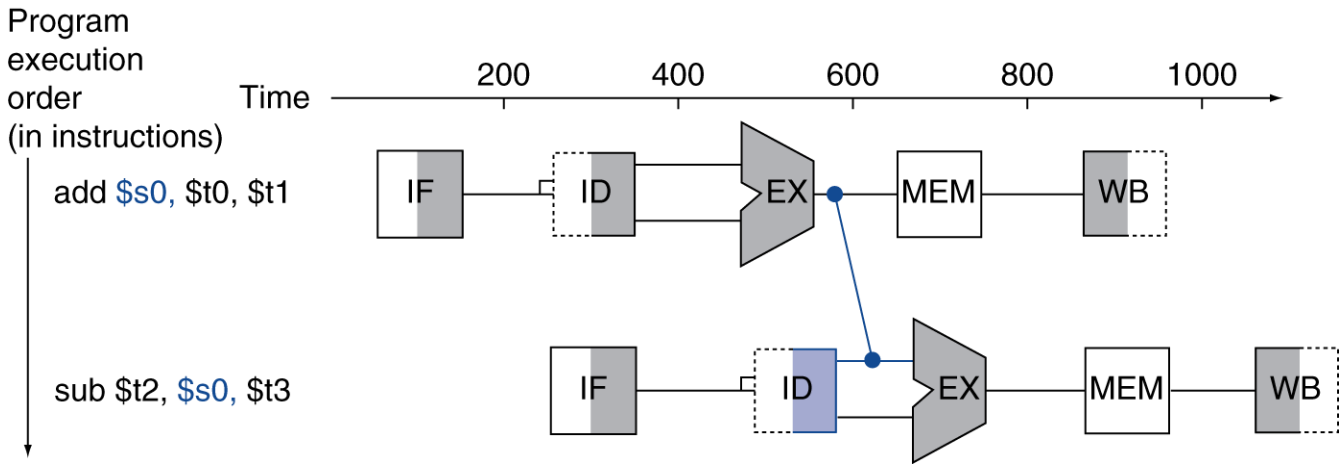
# Data Hazards

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



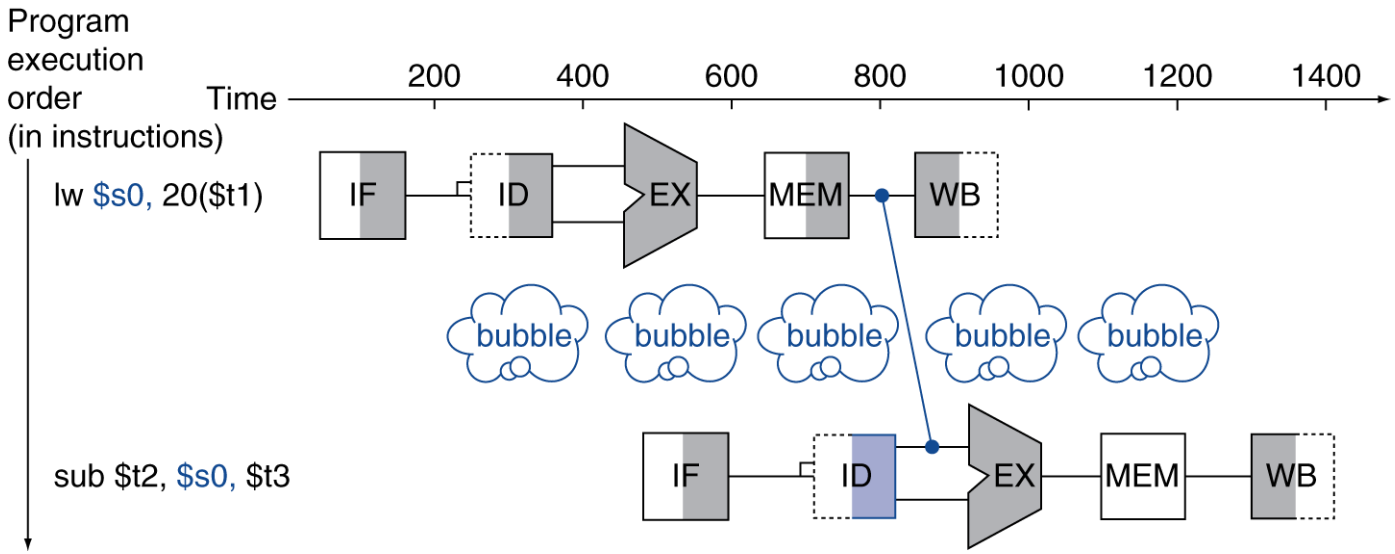
# Forwarding (aka Bypassing)

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



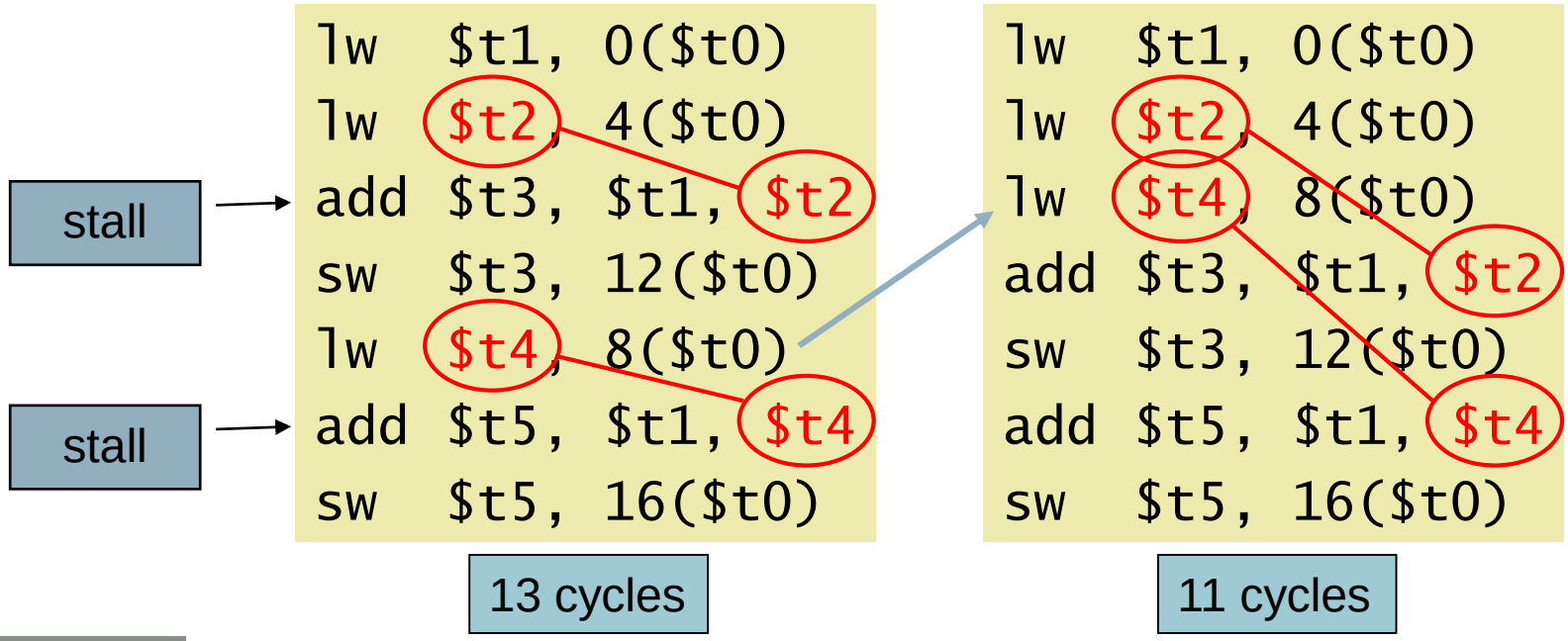
# Load-Use Data Hazard

- 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;`

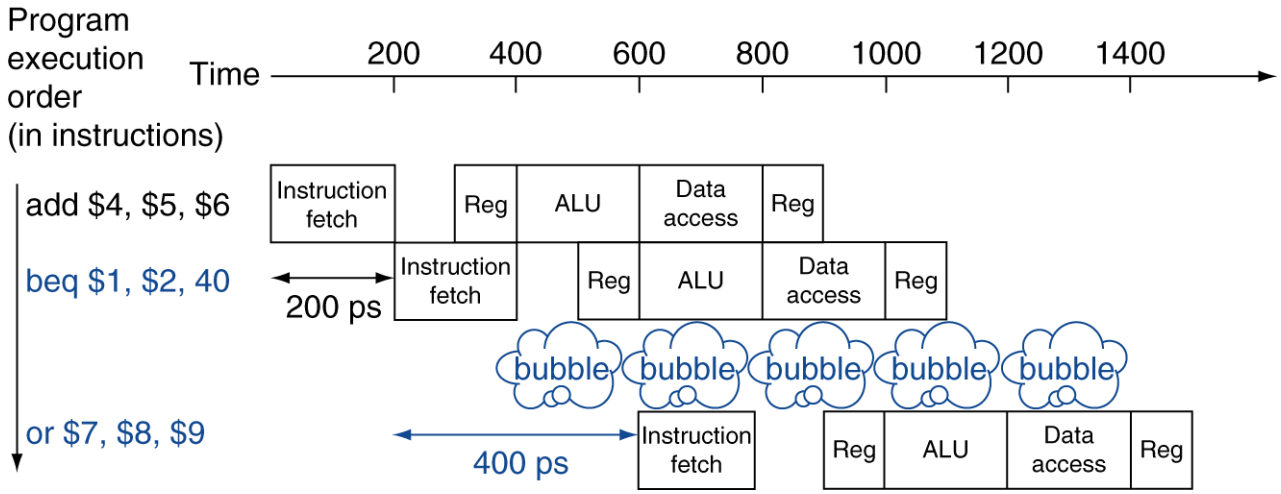


# Control Hazards

- 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



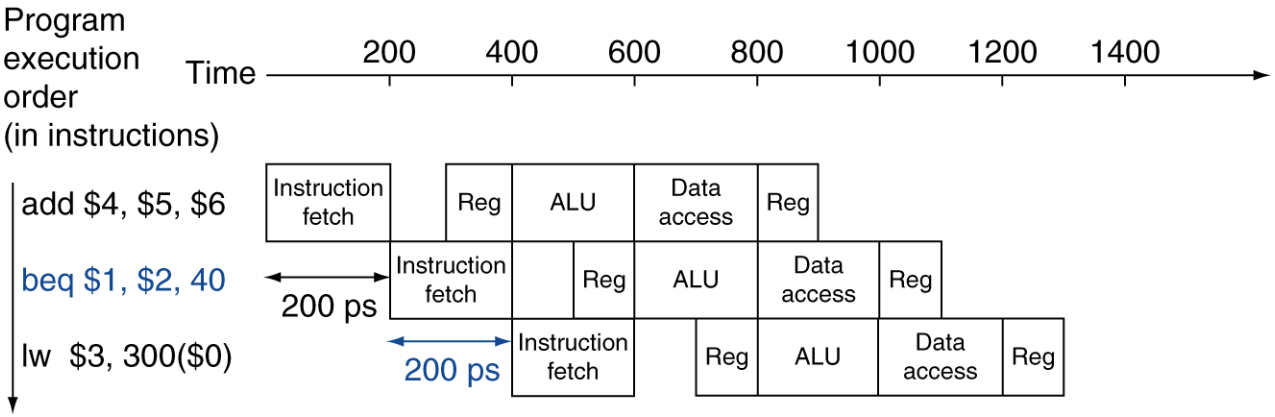
# Branch Prediction

- 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

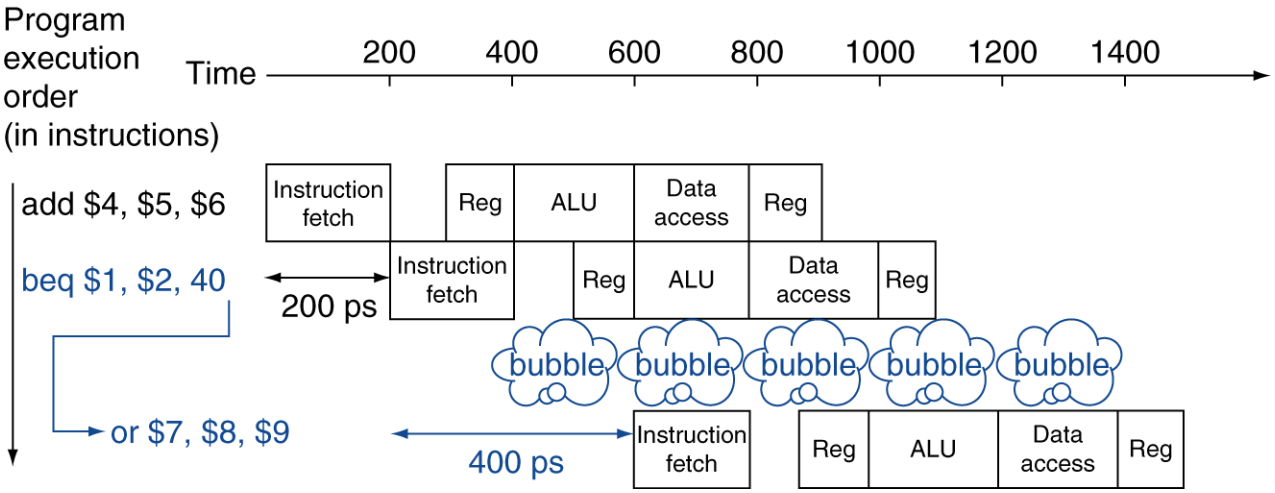
# MIPS with Predict Not Taken



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

# 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

# Acknowledgement

The slides are adopted from Computer

Organization and Design, 5th Edition

by David A. Patterson and John L. Hennessy
2014, published by MK (Elsevier)