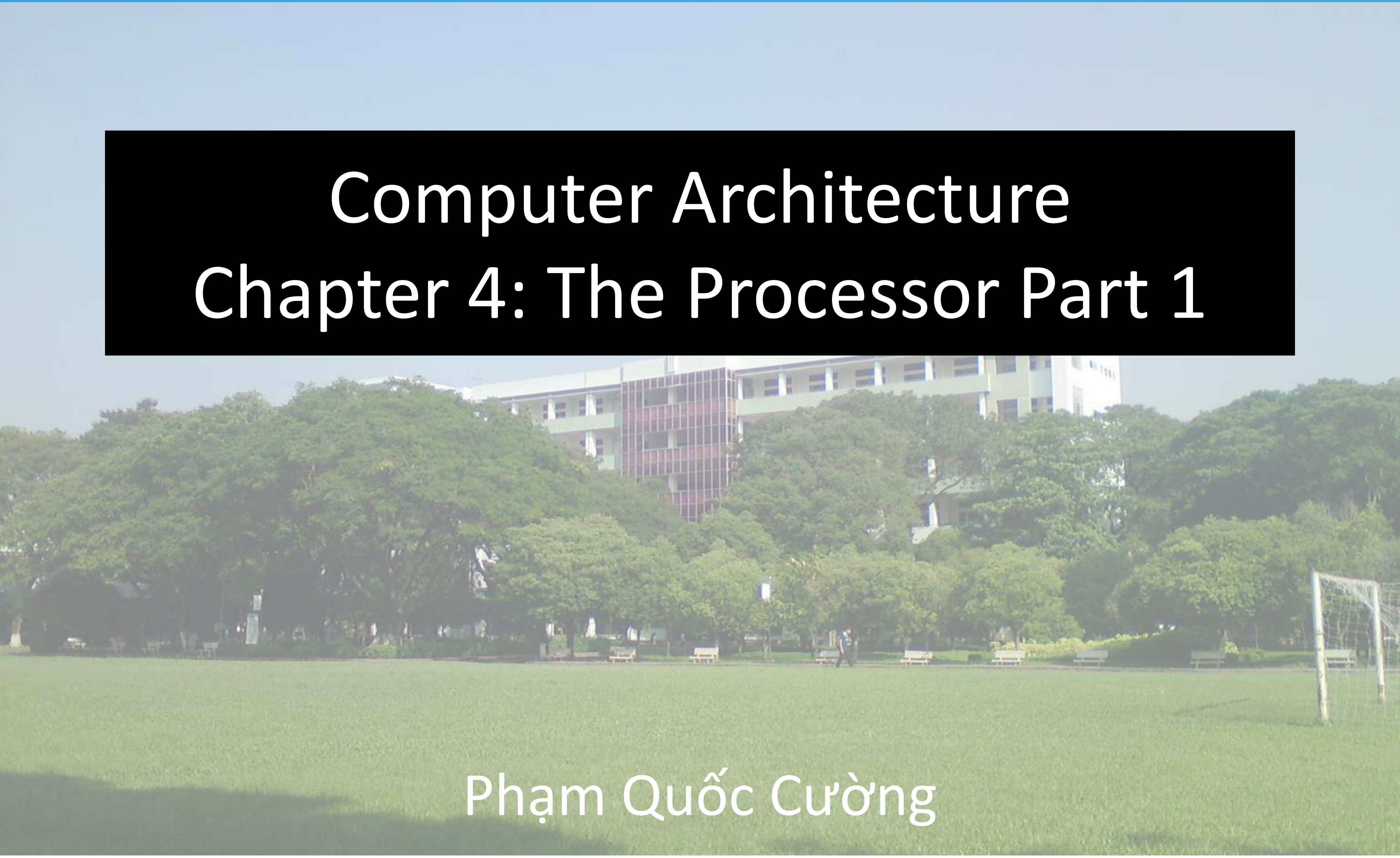


Computer Architecture
Chapter 4: The Processor Part 1

Phạm Quốc Cường

Adapted from Computer Organization the Hardware/Software Interface – 5th

Computer Engineering – CSE – HCMUT

Introduction

- CPU performance factors

CPU Time = IC x CPI x Cycle Time

  - Instruction count (IC)
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

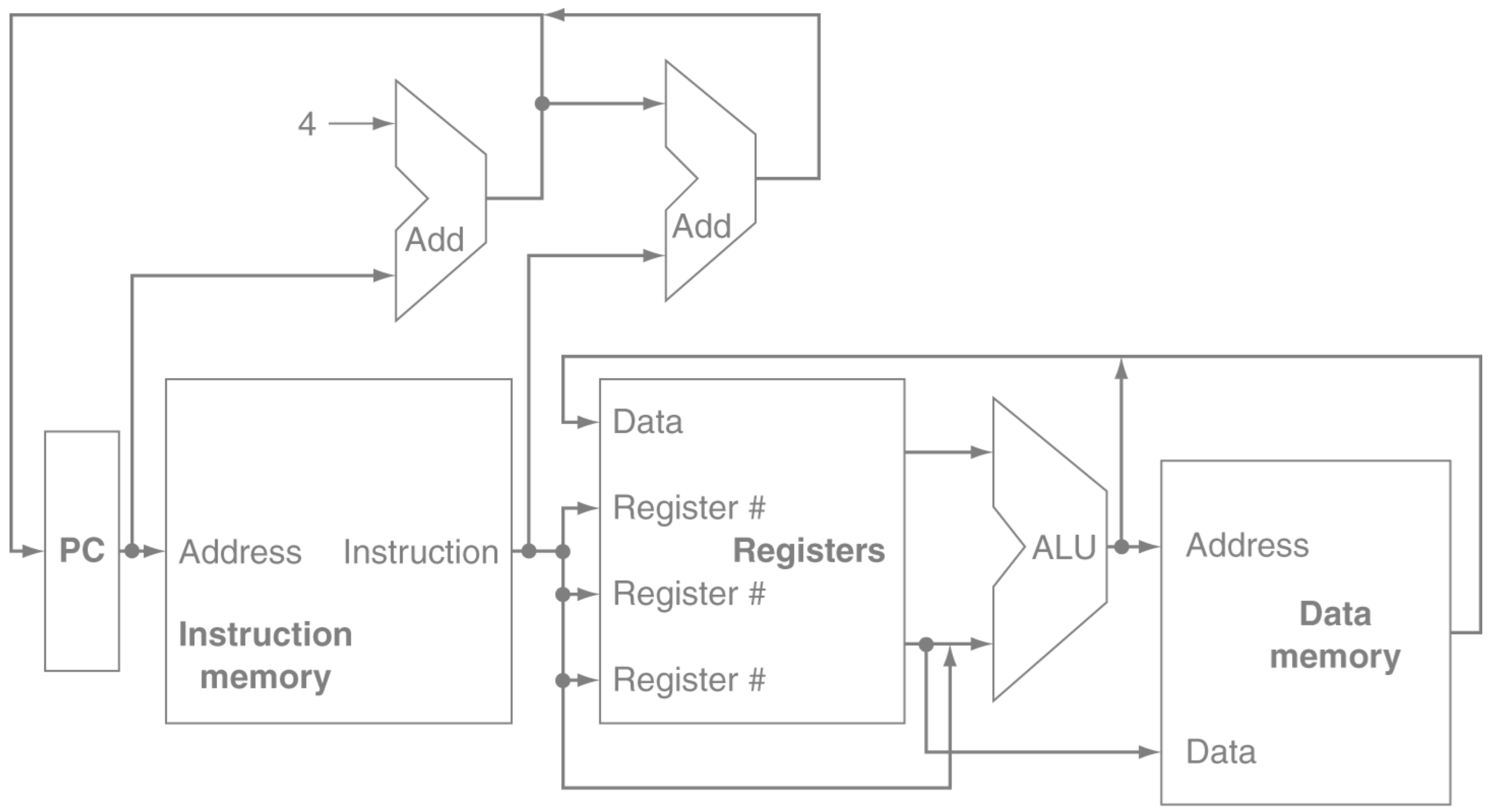
Introduction

- 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`
- How clock rate and CPI could be effected.

Instruction Execution

- Almost instructions
  - PC → instruction memory → fetch instruction
  - Register id → register file (set) → read registers (1 or 2 reads) (Except J type)
- 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 or update Jump target
  - Write back result to register of destination

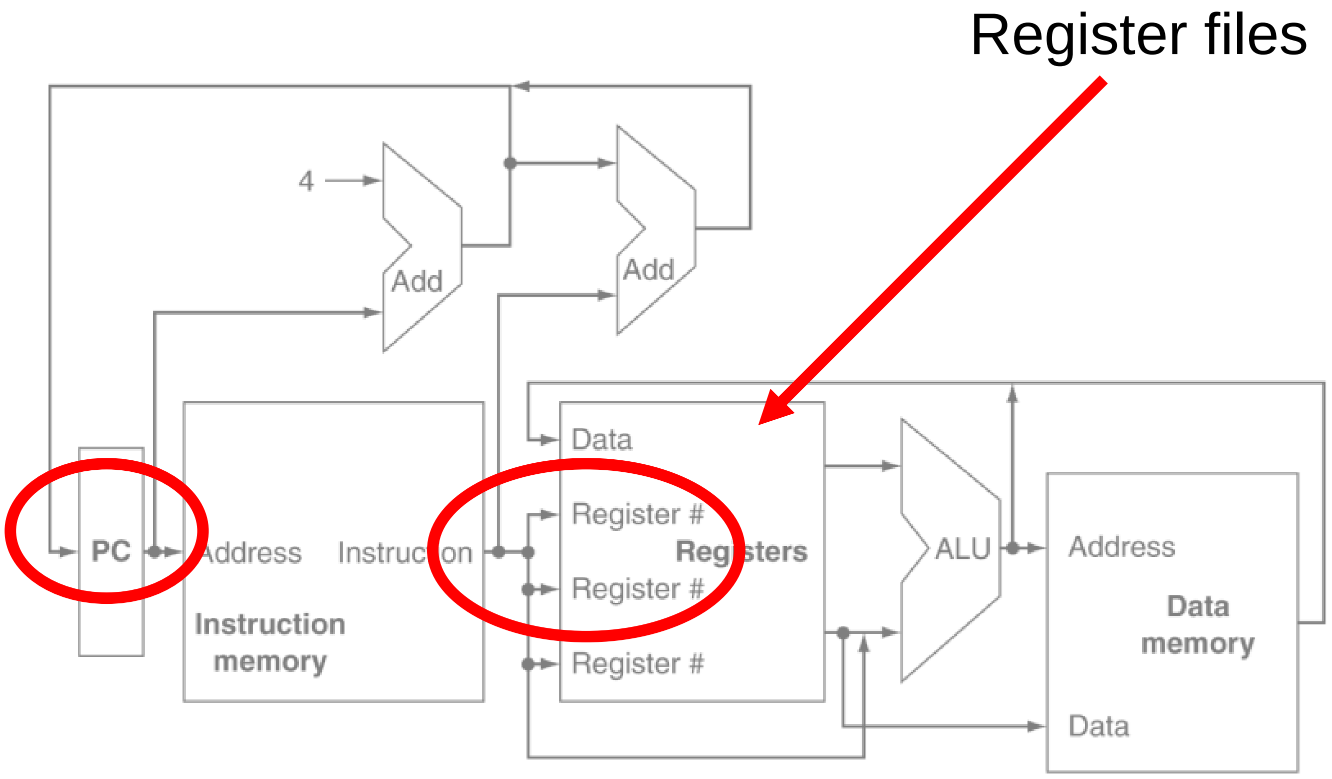
CPU Overview



Instruction Execution

Almost instructions

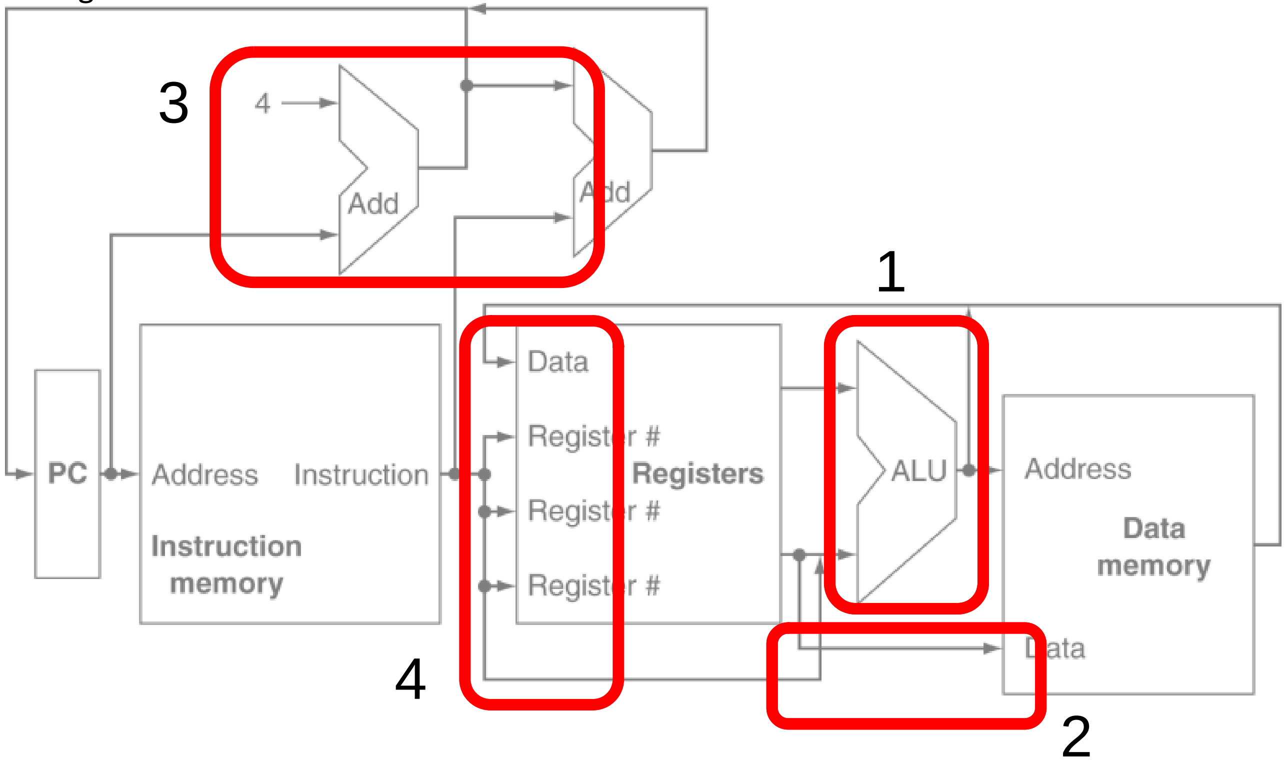
- PC → instruction memory → fetch instruction
- Register id → register file (set) → read registers (1 or 2 reads) (except J type)



Instruction Execution

Depending on instruction class

- (1) Use ALU to calculate
  - Arithmetic result
  - Memory address for load/store
  - Branch target address
- (2) Access data memory for load/store
- (3) PC –> target address or PC + 4 or update Jump target
- (4) Write results to register of destination.

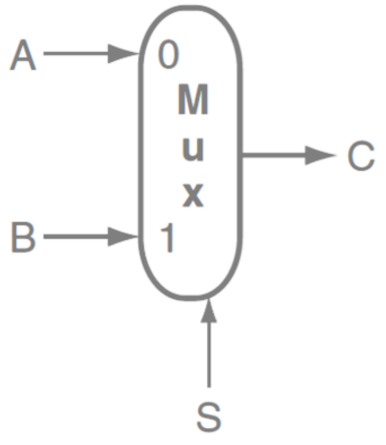


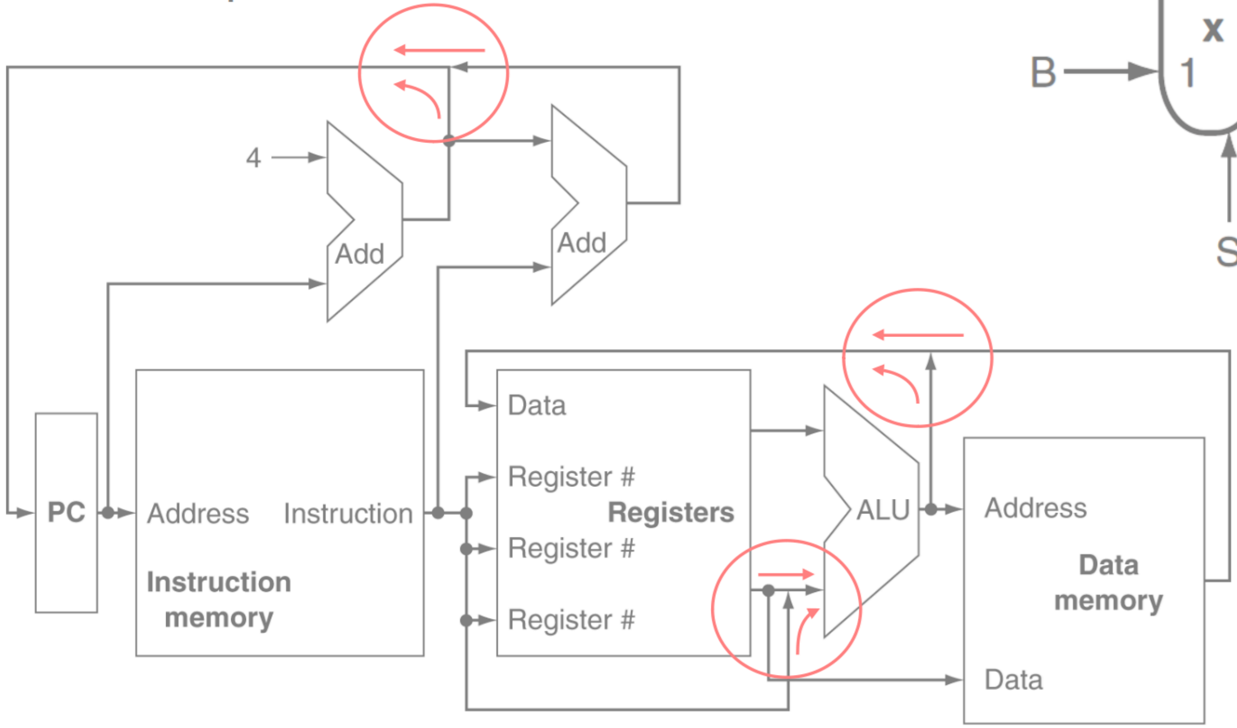
Execution Model

- Instruction fetch: PC -> instruction address
- Instruction decode: register operands → register file
- Instruction execute:
  - Load/store: compute a memory address
  - Arithmetic: compute an arithmetic result
- Write back:
  - Load/store: store a value to a register or a memory location
  - Arithmetic: store a result of register file

Multiplexers

- Wire (connections) cannot joint together → Use Multiplexers (or data selector)





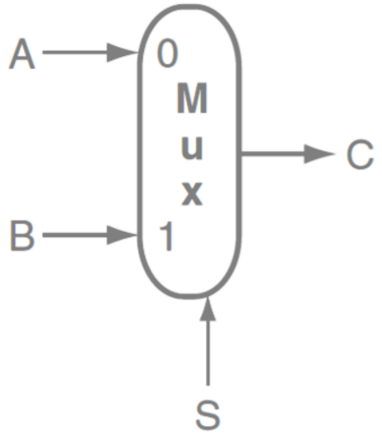
Control vs Data signals

- How to select appropriate wire?
- When Registers need to be read/write?
- When memory need to be read/write?

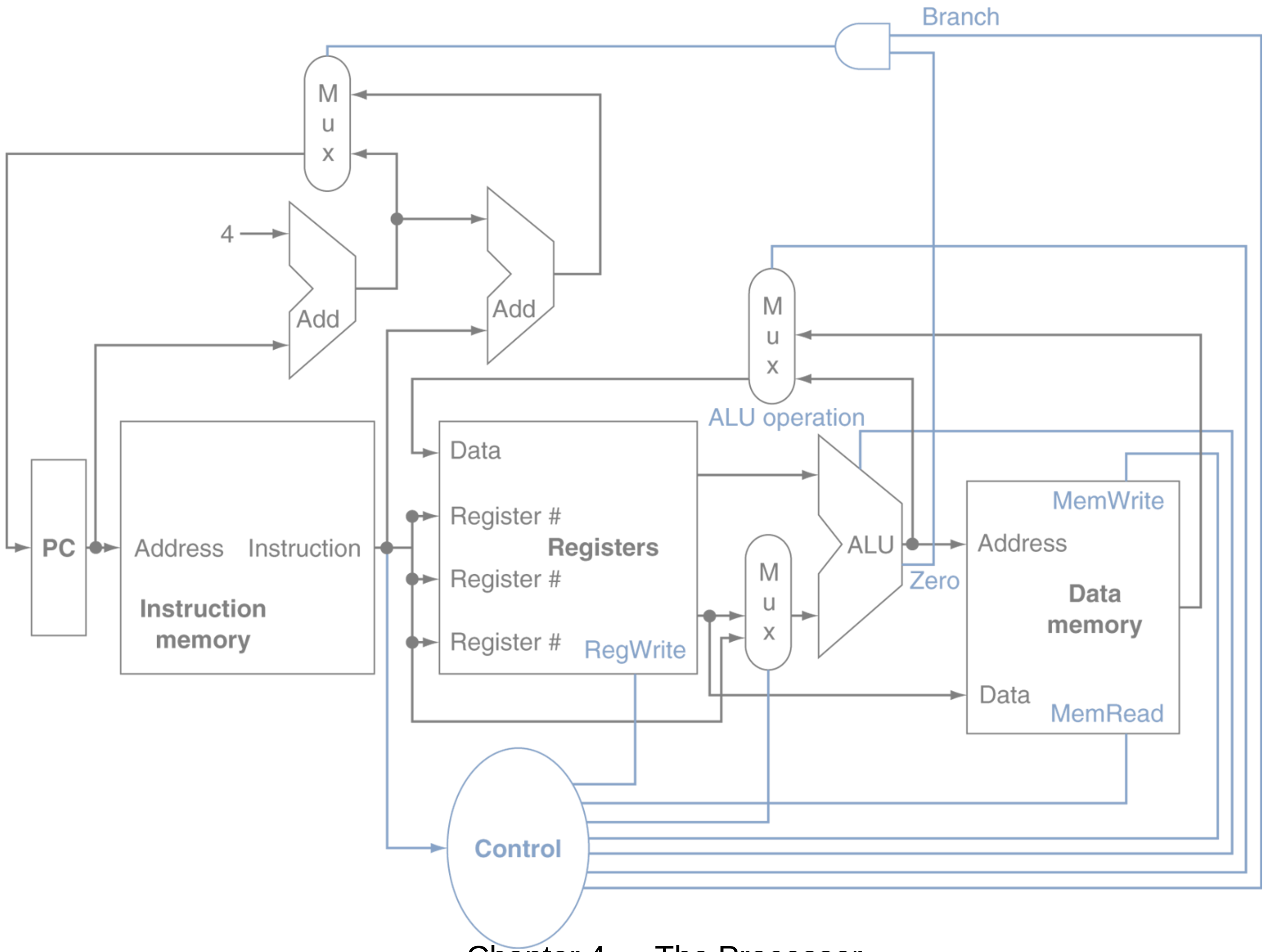
Control vs. Data signals

- Control signal (S): used for multiplexer selection or for directing the operation of a functional unit

- Data signal (A, B, ...): contains information that is operated on by a functional unit



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

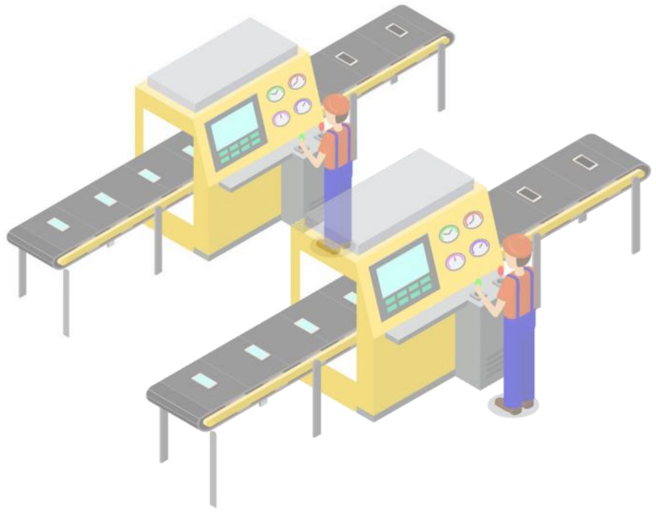
Logic Design Basics

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

Combinational vs State element

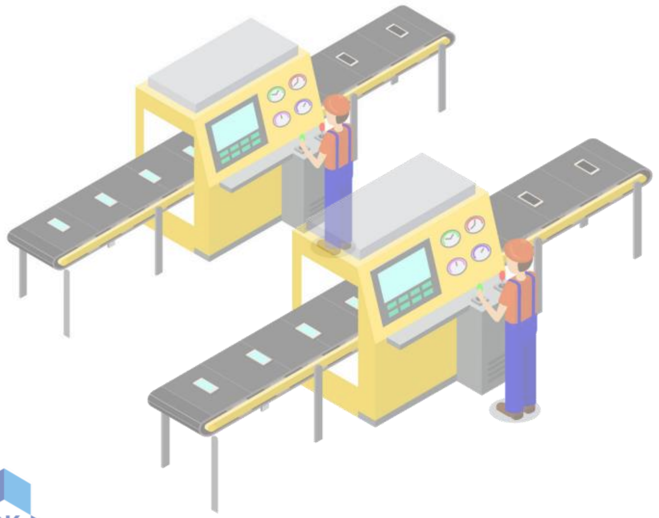
- Combination
- → producer

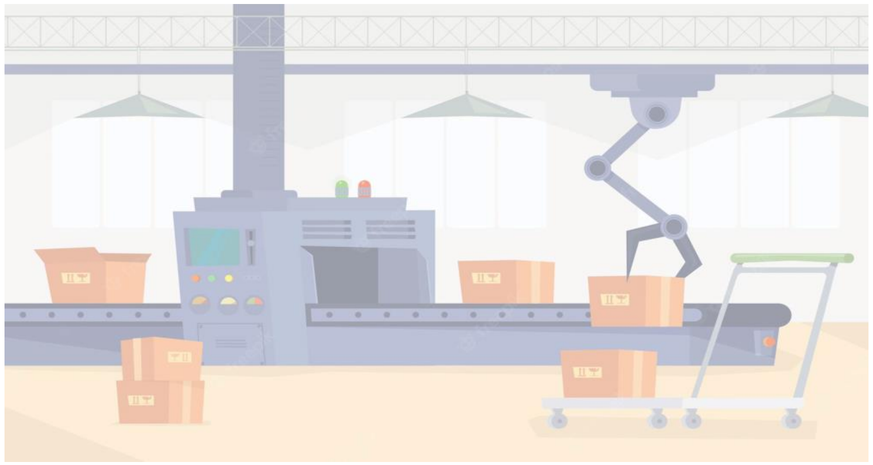
- State element → pick up and store



Controller

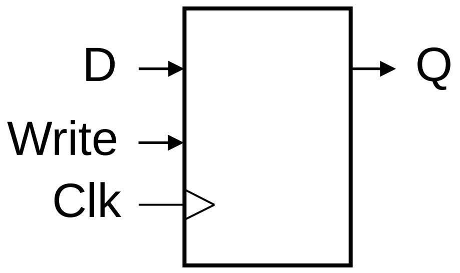
- What will be produced?
- Where the the production come from? → Data selector
- What and when the production is stored?

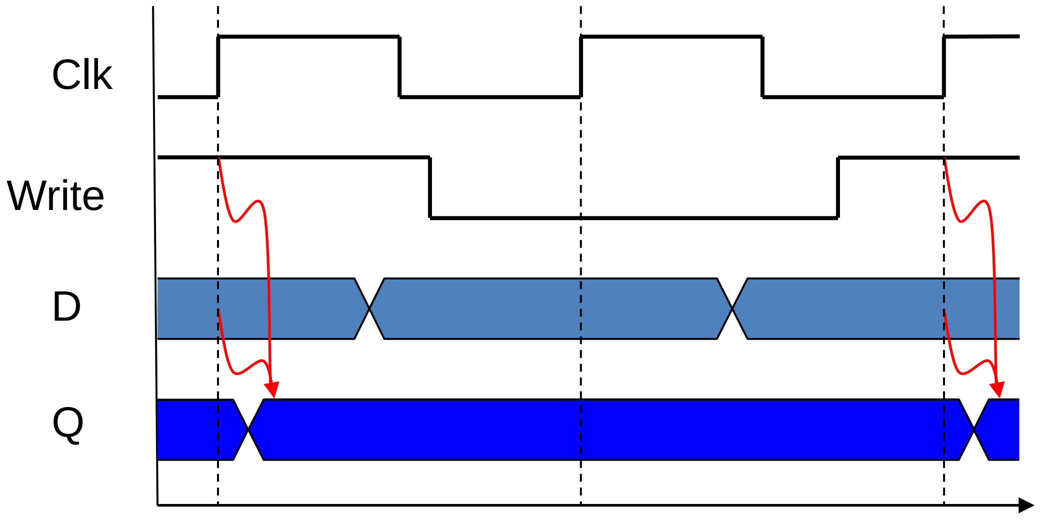




Sequential Elements

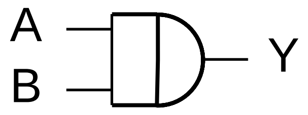
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



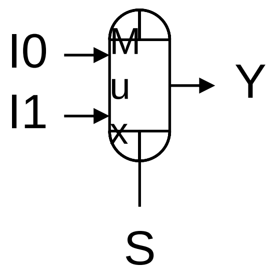


Combinational Elements

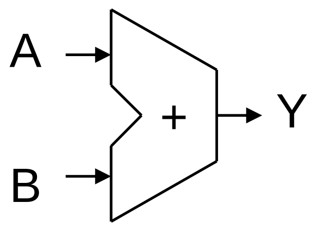
- AND-gate
  - Y = A & B



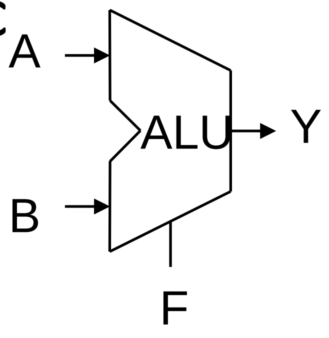
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

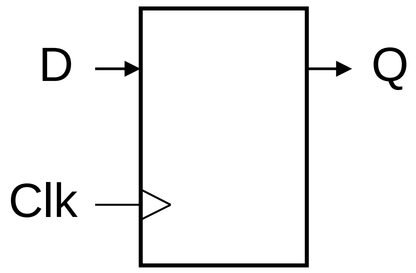


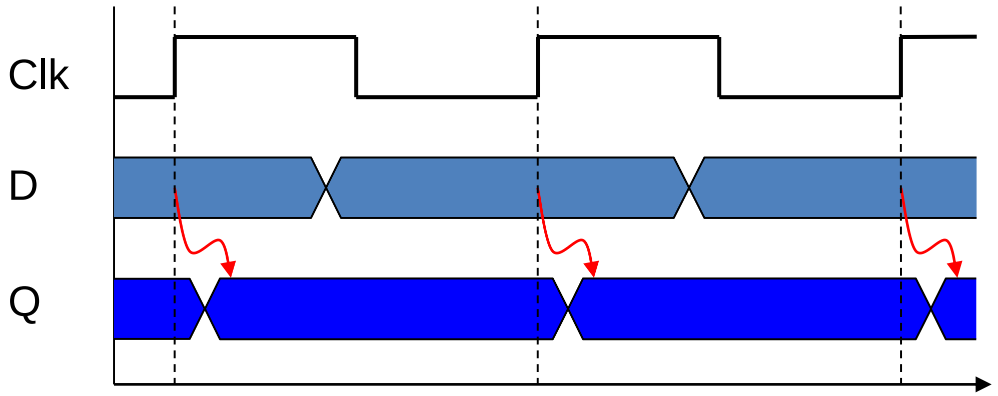
- Arithmetic/Logic Unit
  - Y = F(A,B)



Sequential Elements

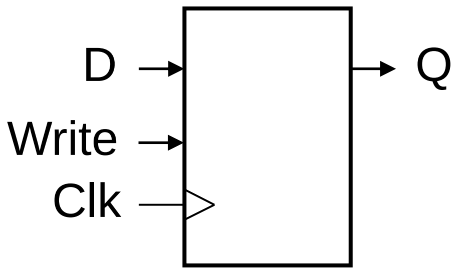
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

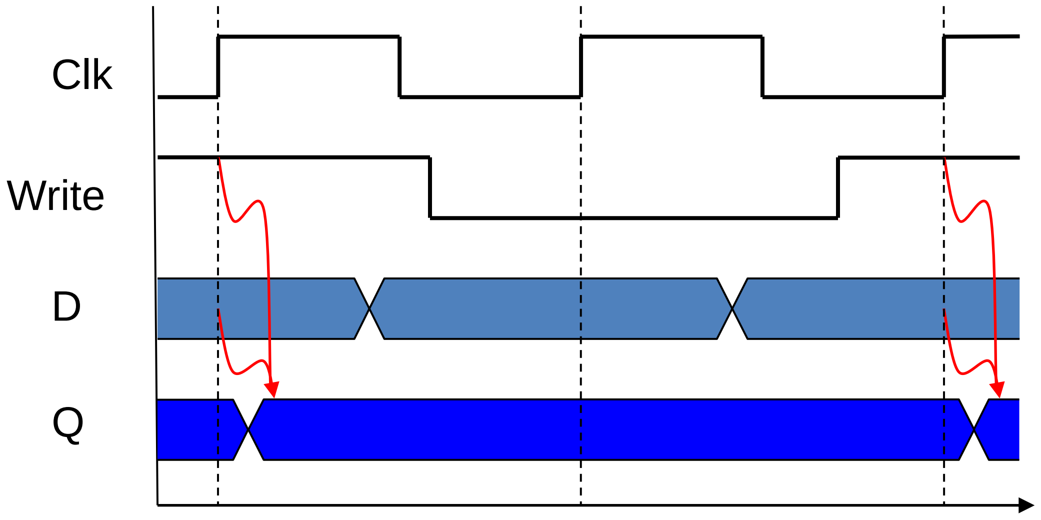




Sequential Elements

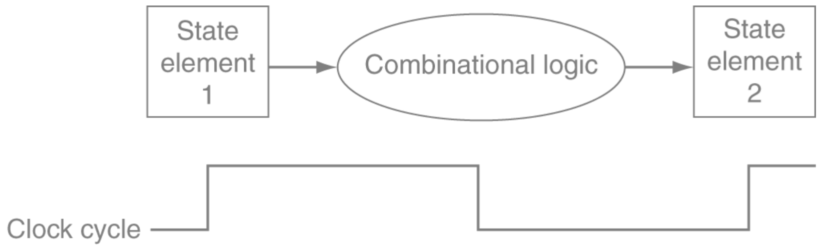
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

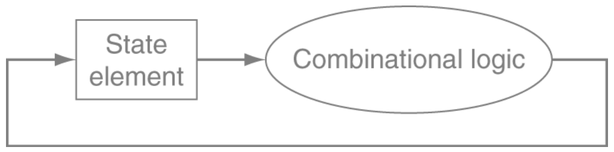




Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements (a memory or a register), output to state element
  - Longest delay determines clock period





Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Longest delay determines clock period

  → Pick up only when the production is available

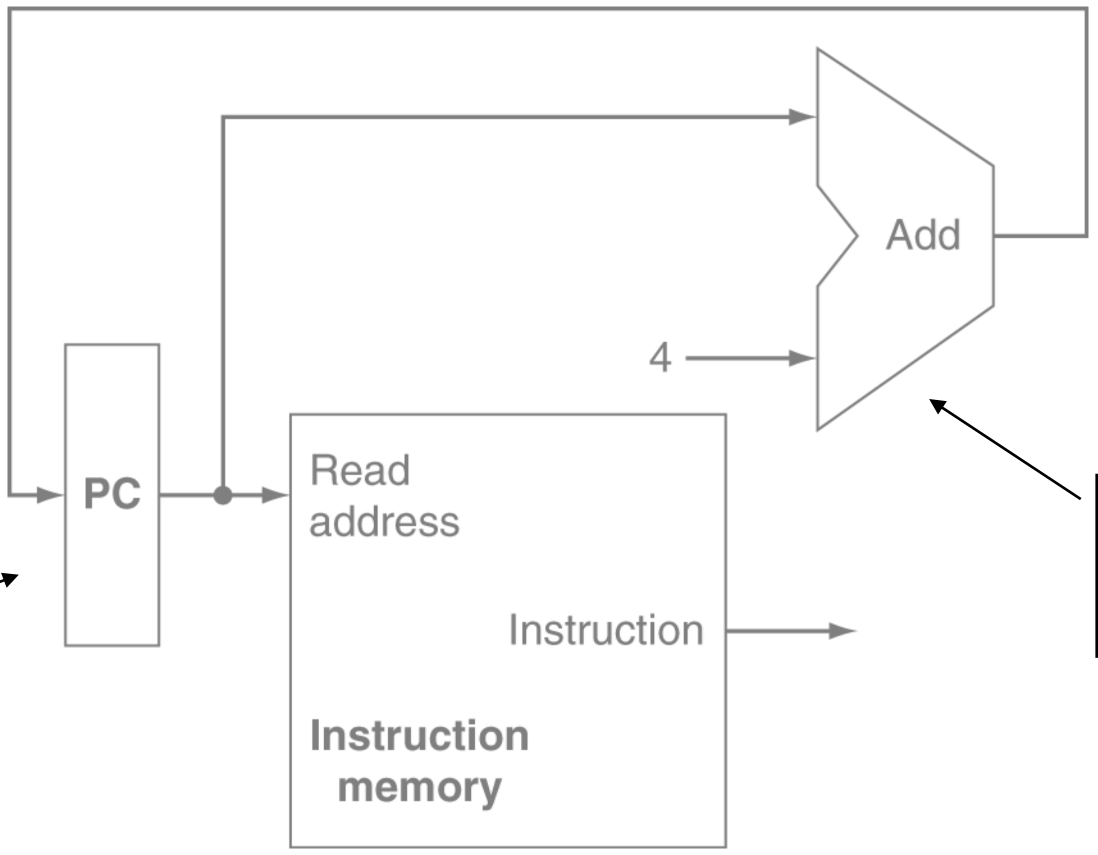
Building a Datapath

- 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

Building a Datapath

- Instruction fetch: PC -> instruction address
- Instruction decode: register operands → register file
- Instruction execute:
  - Load/store: compute a memory address
  - Arithmetic: compute an arithmetic result
- Write back:
  - Load/store: store a value to a register or a memory location
  - Arithmetic: store a result of register file

Instruction Fetch

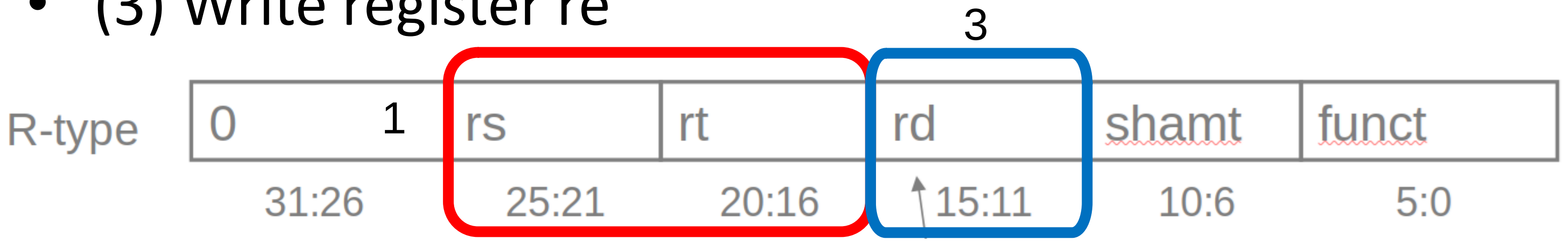


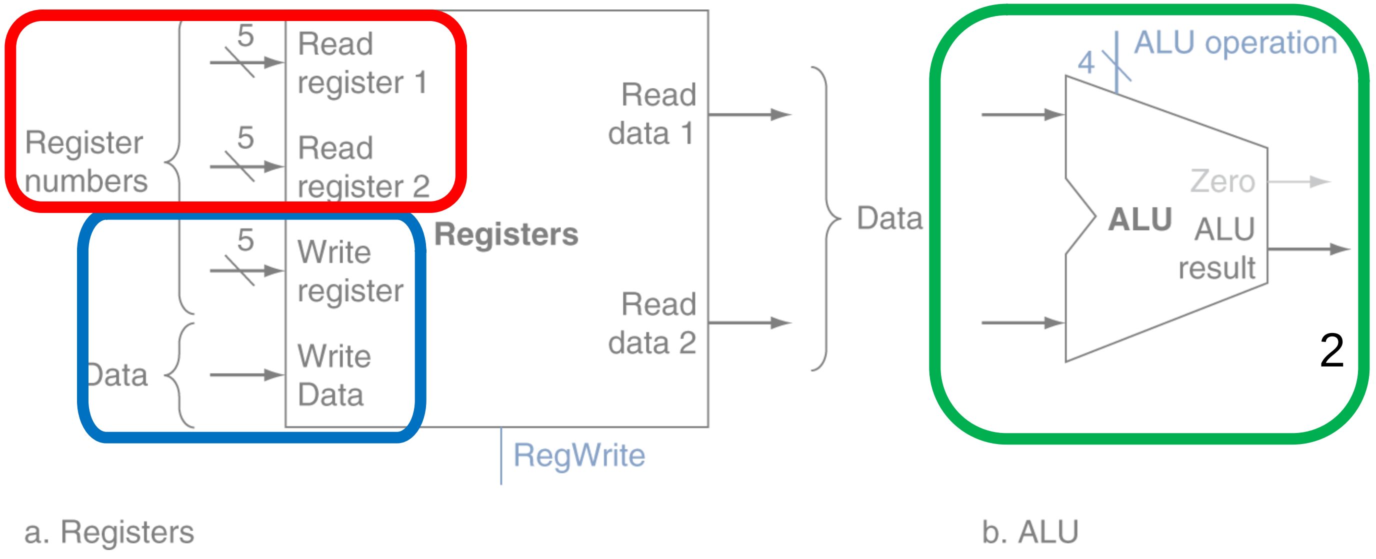
32-bit register

Increment by 4 for next instruction

R-Format Instructions

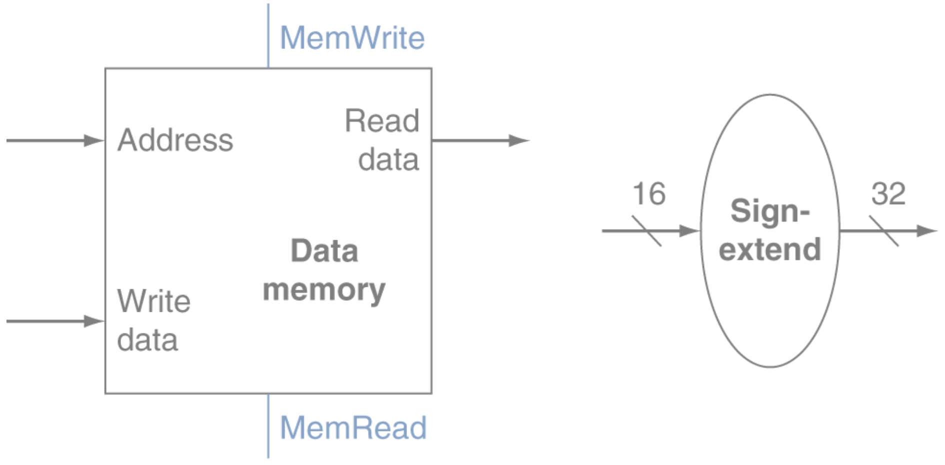
- (1) Read two register operands
- (2) Perform arithmetic/logical operation
- (3) Write register re





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

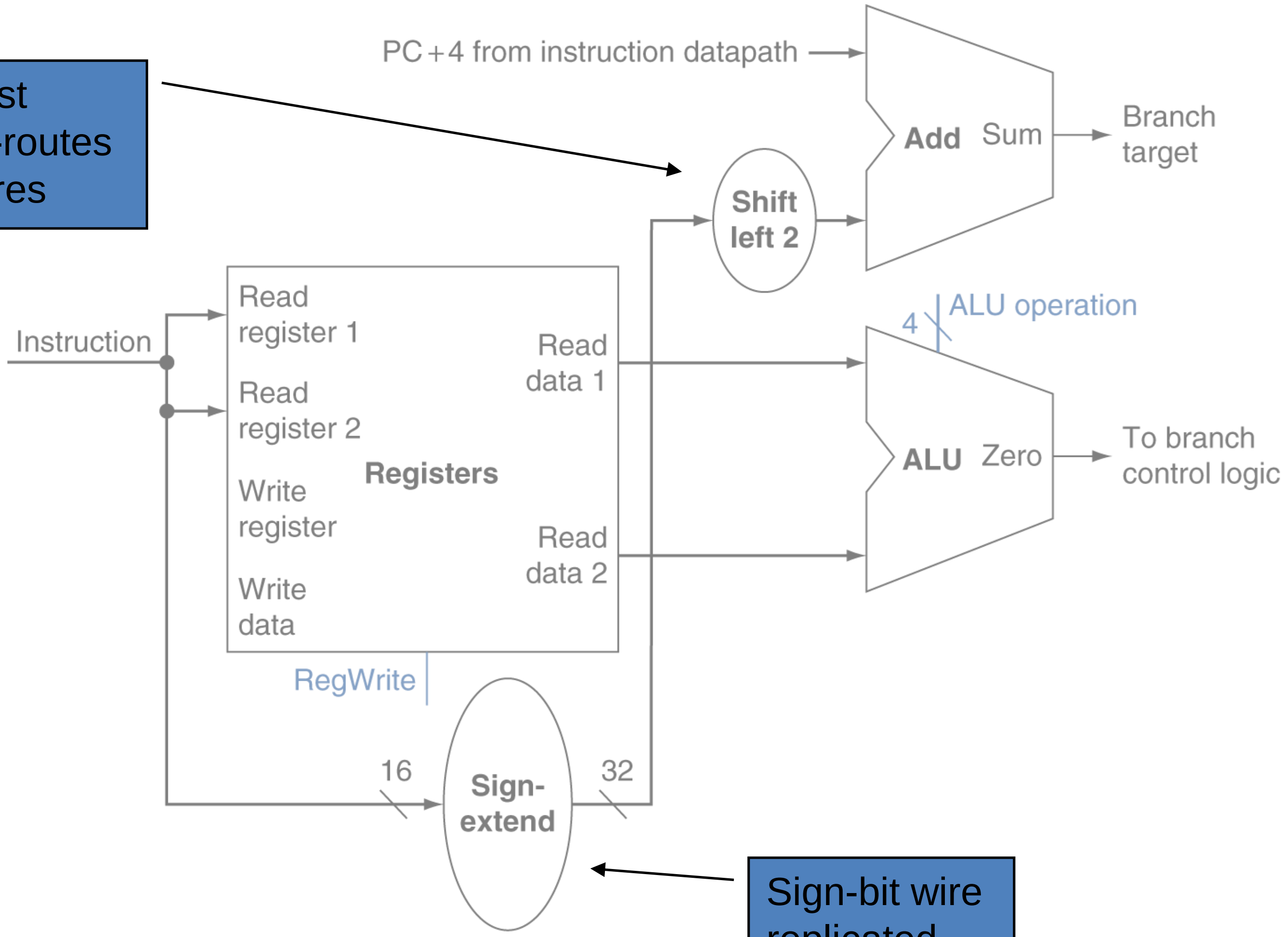
Branch Instructions

- Brach taken: condition is satisfied and the Program Counter (PC) register becomes the branch target
- Branch not taken: PC becomes the address of the next instruction
- Datapath:
  – Compute the branch target
  – Compare the registers

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



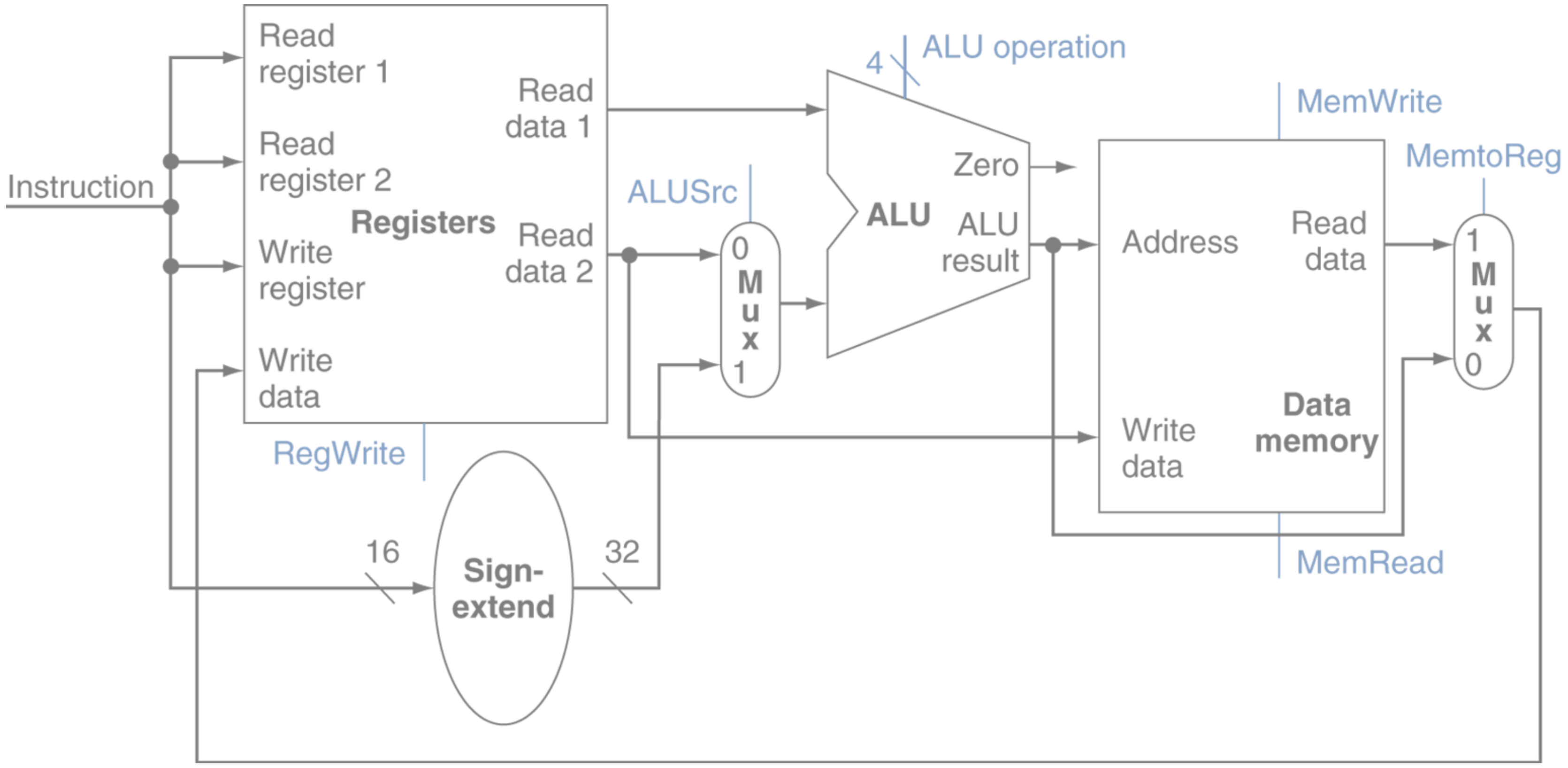
Just re-routes wires

Sign-bit wire replicated

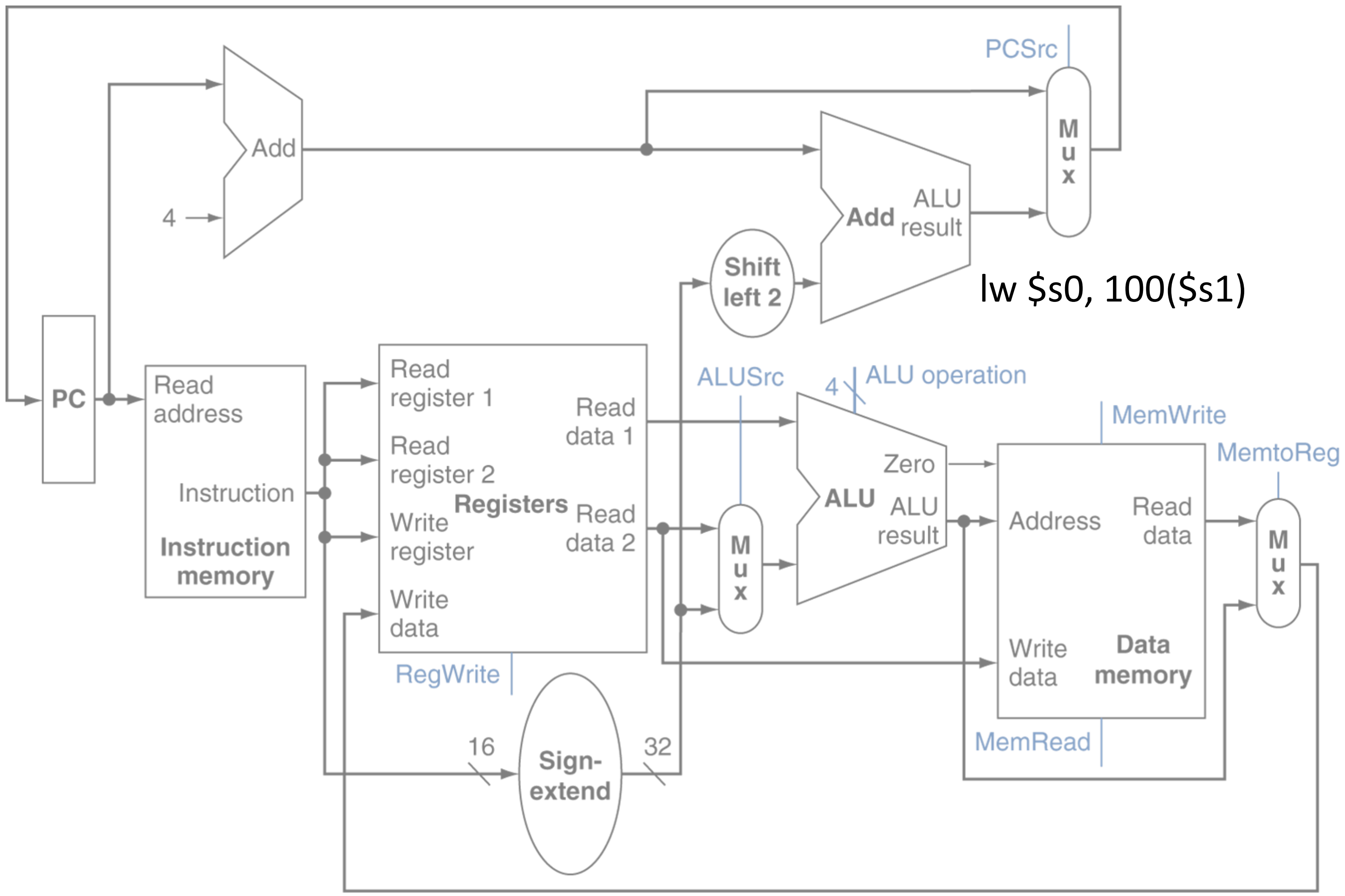
R-Type/Load/Store Datapath

- Instruction execute:
  - Load/store: compute a memory address
  - Arithmetic: compute an arithmetic result
- Write back:
  - Load/store: store a value to a register or a memory location
  - Arithmetic: store a result of register file

R-Type/Load/Store Datapath

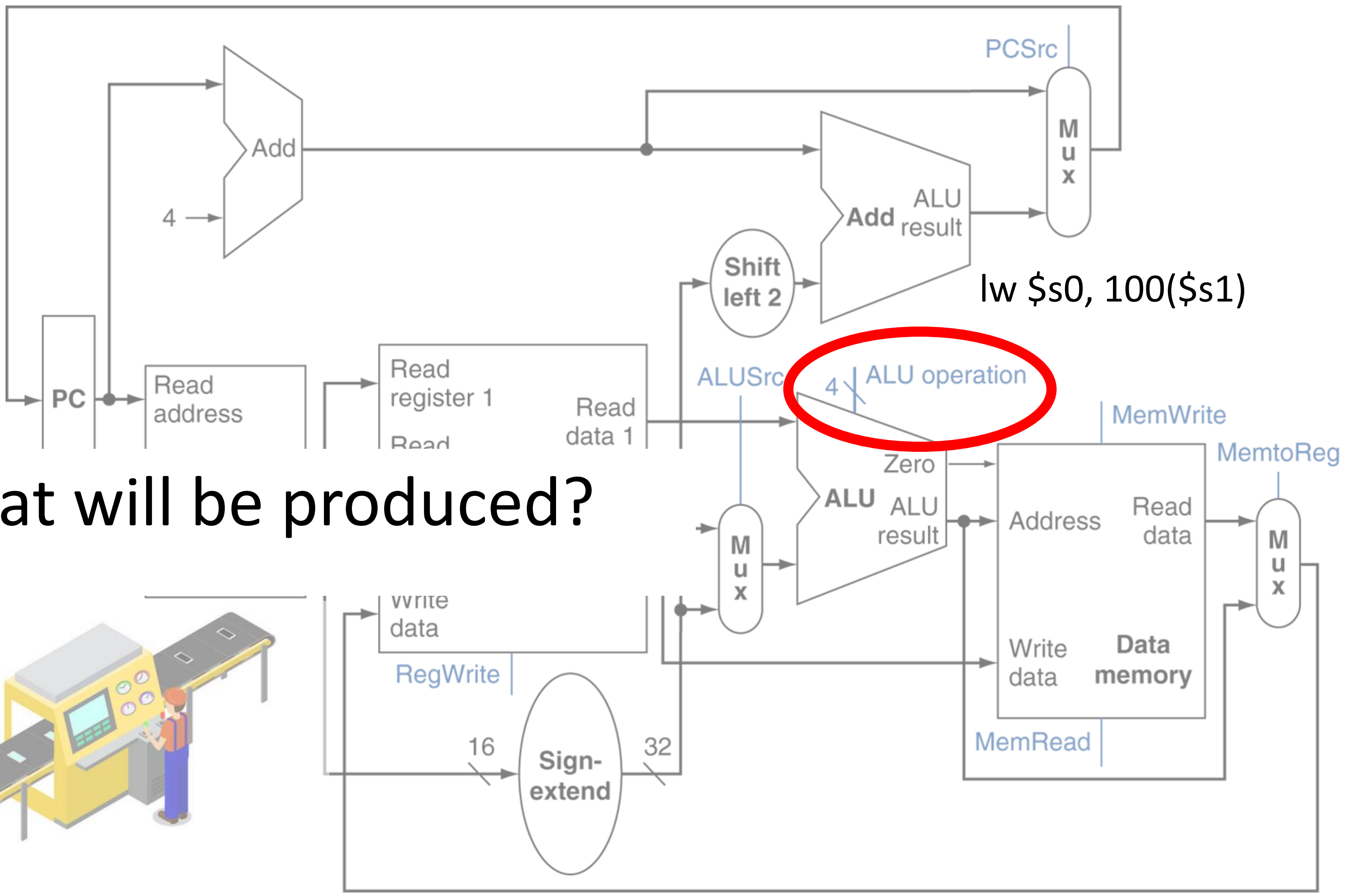


Full Datapath



lw $s0, 100($s1)

Full Datapath



lw $s0, 100($s1)

What will be produced?

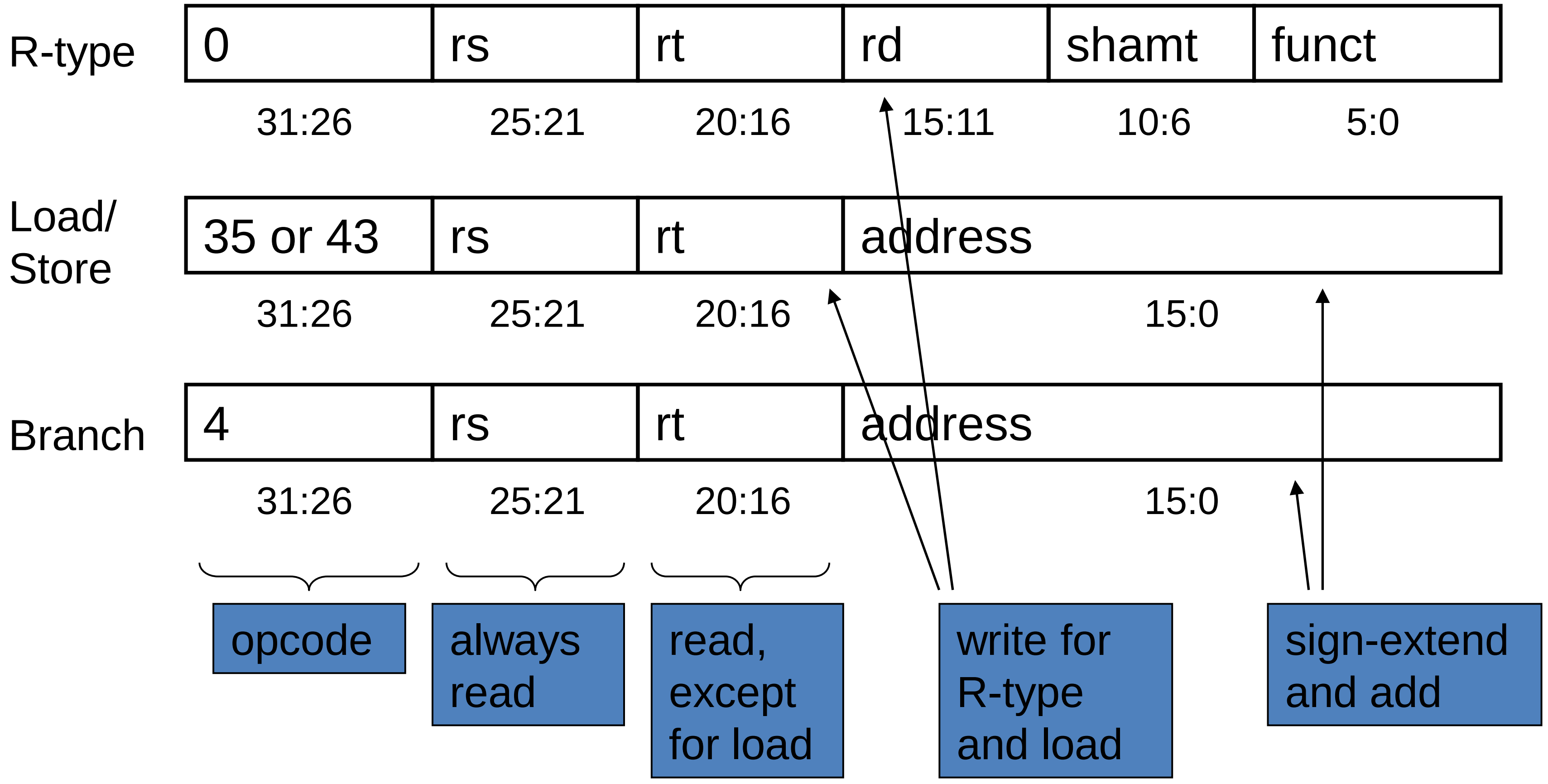
ALU Control

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

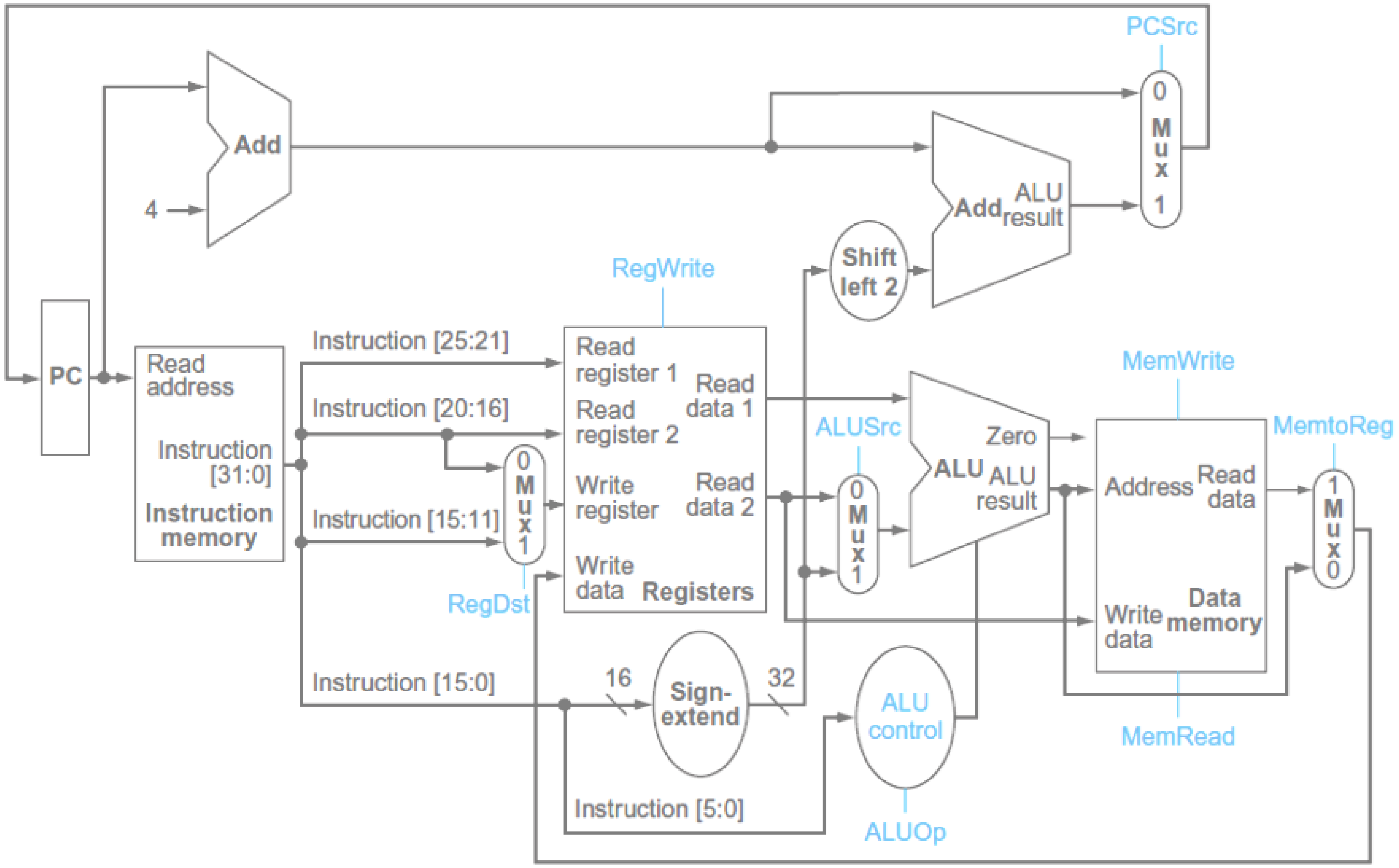
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

The Main Control Unit

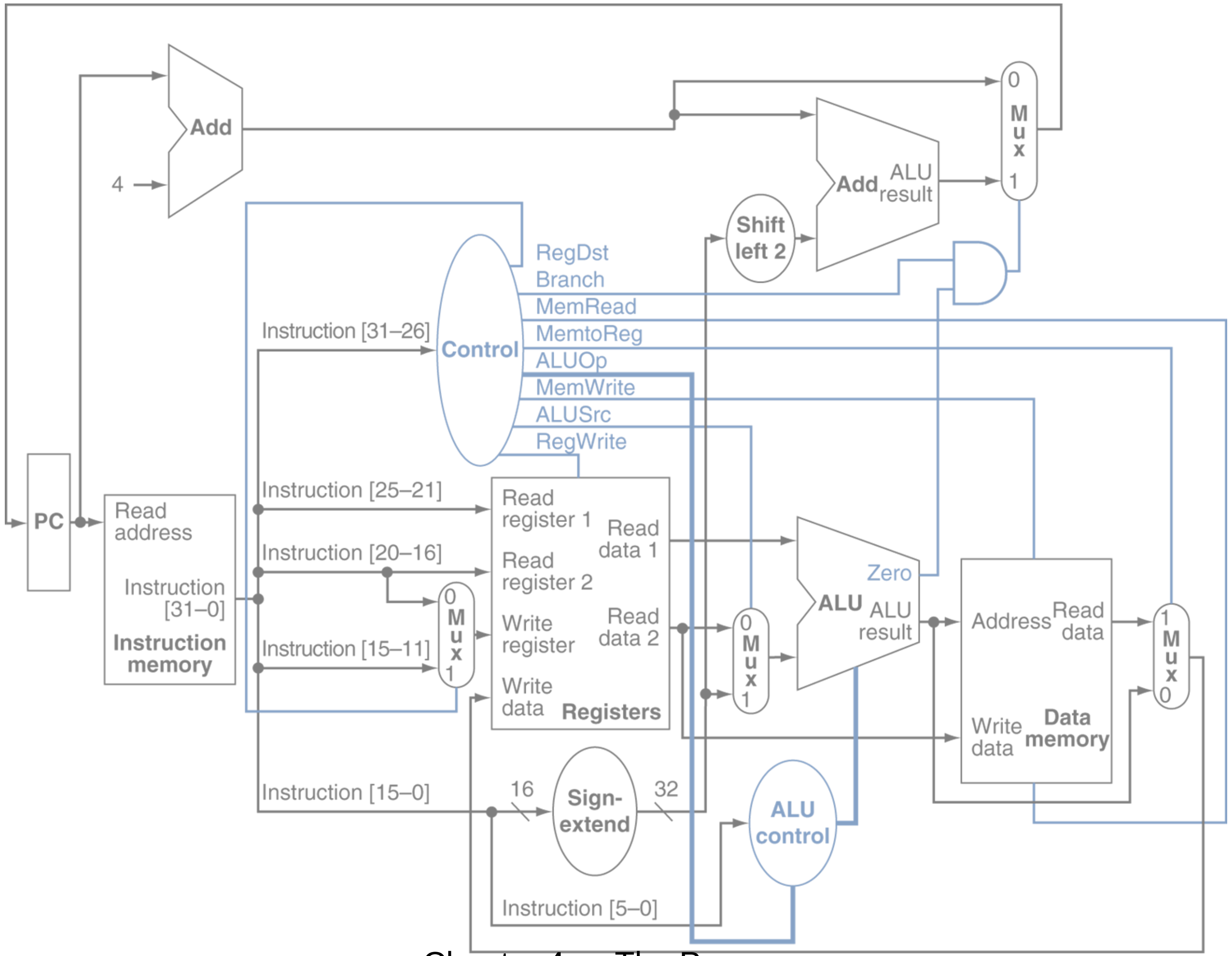
- Control signals derived from instruction



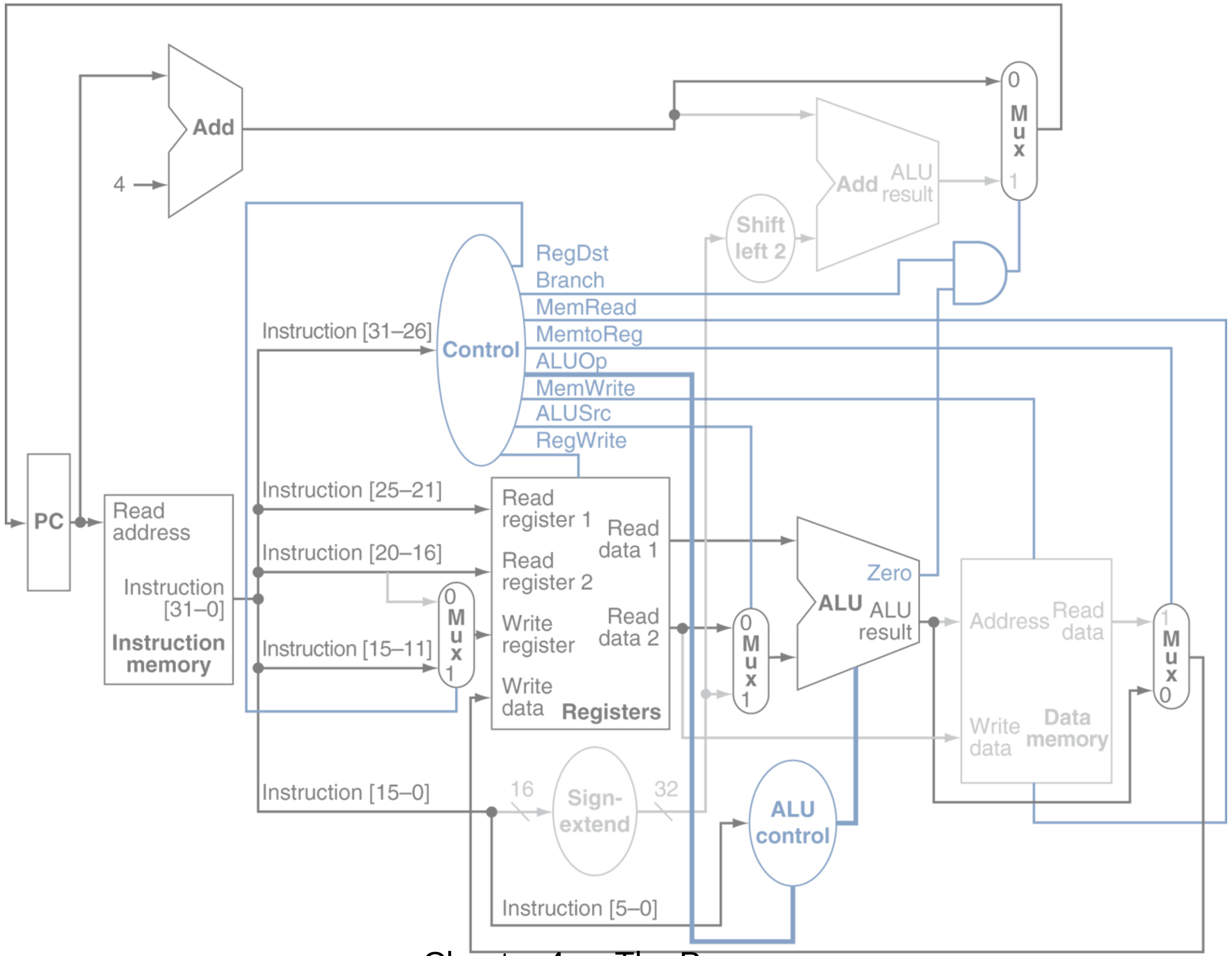
Datapath with Instruction Fields



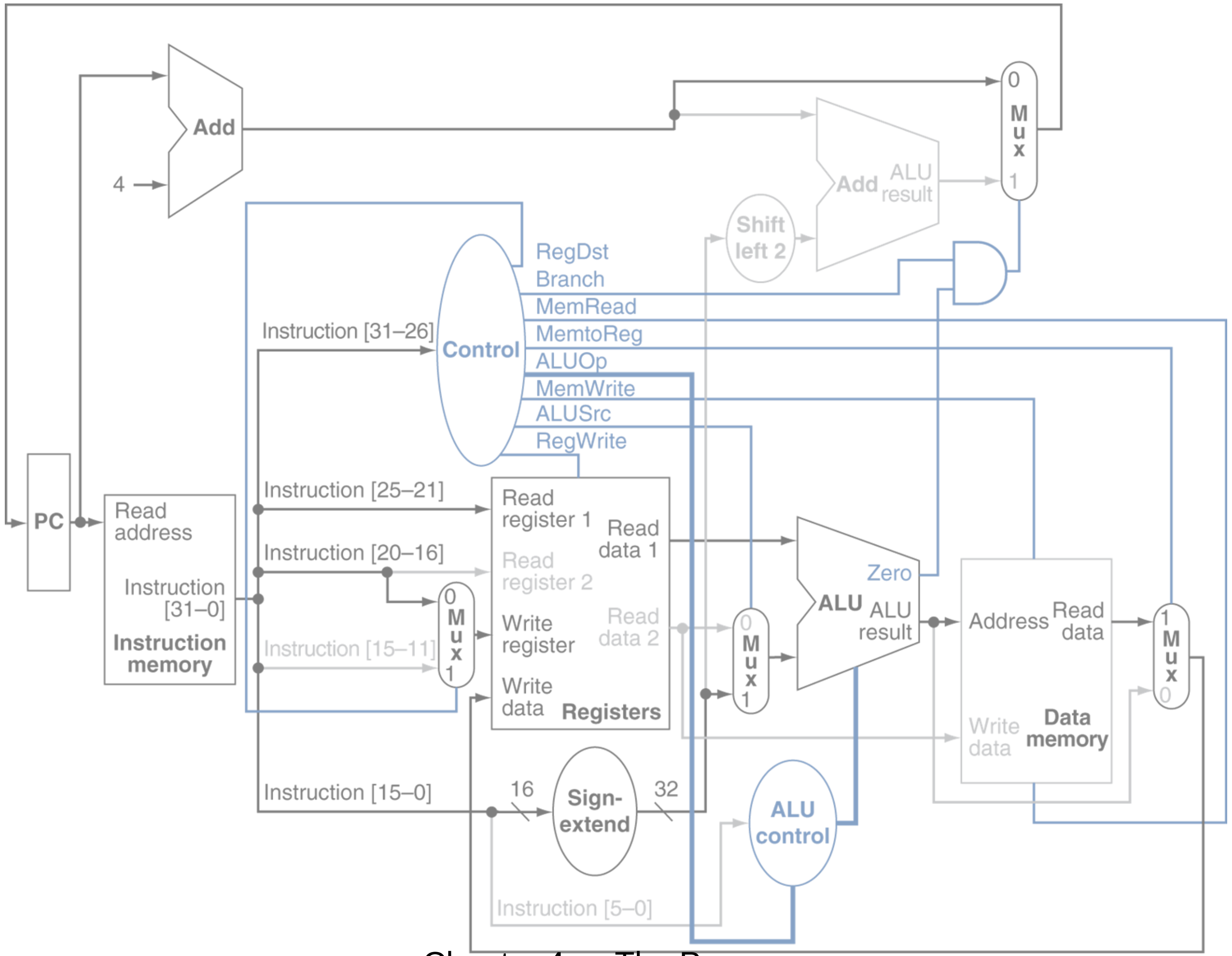
Datapath With Control



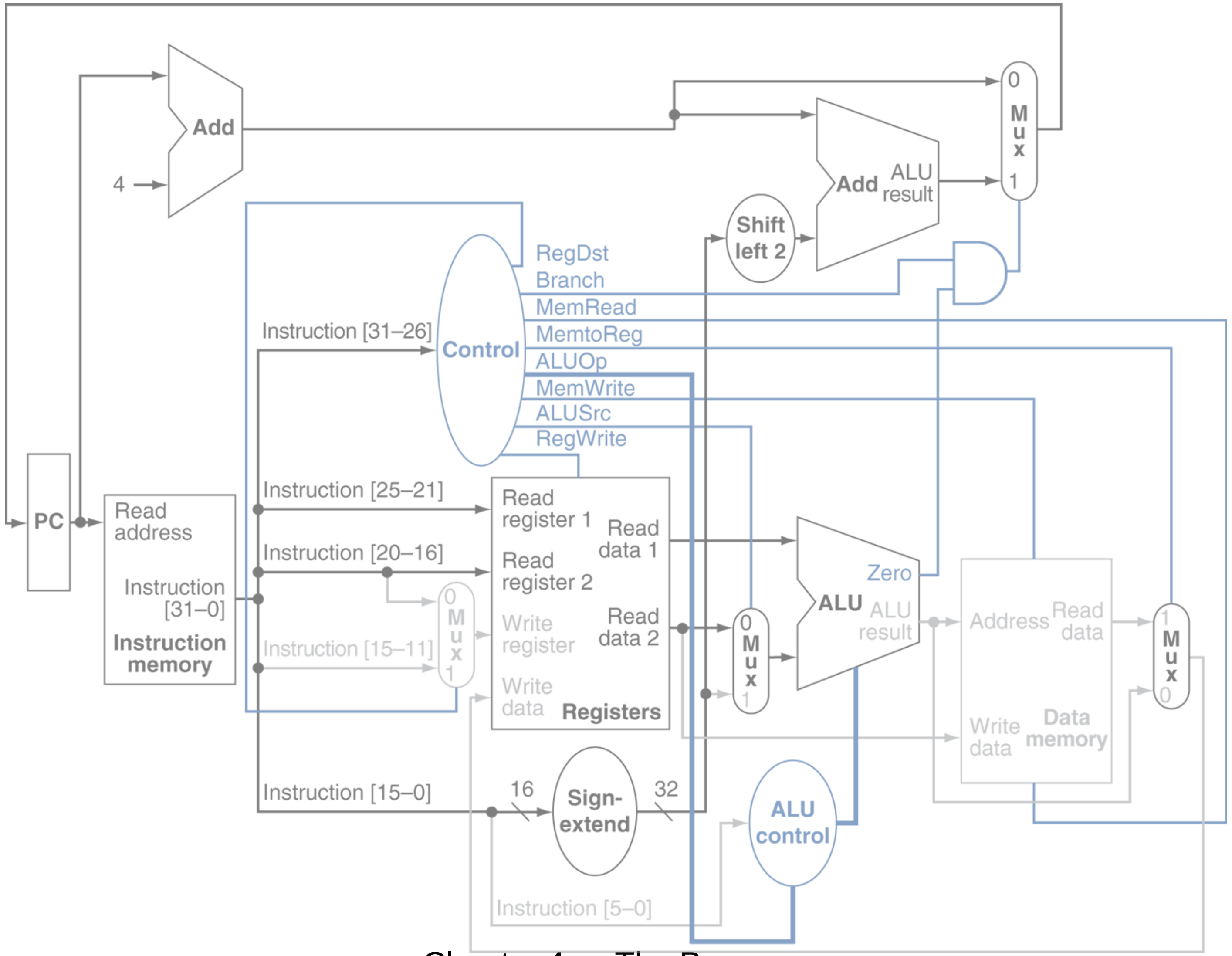
Example: R-Type Instruction



Example: Load Instruction



Example: Branch-on-Equal Instruction

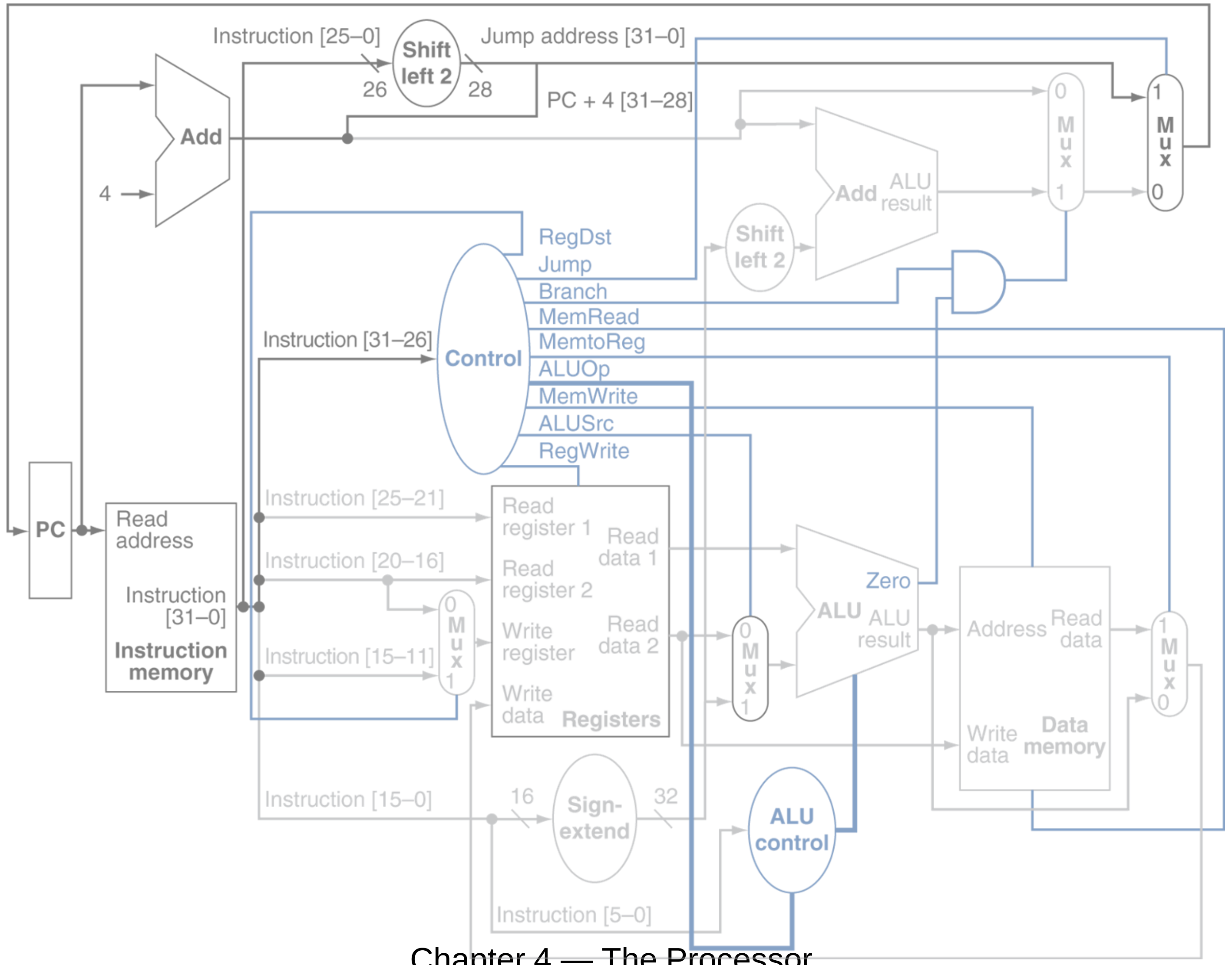


Implementing Jumps

- 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

Jump	2	address
	31:26	25:0

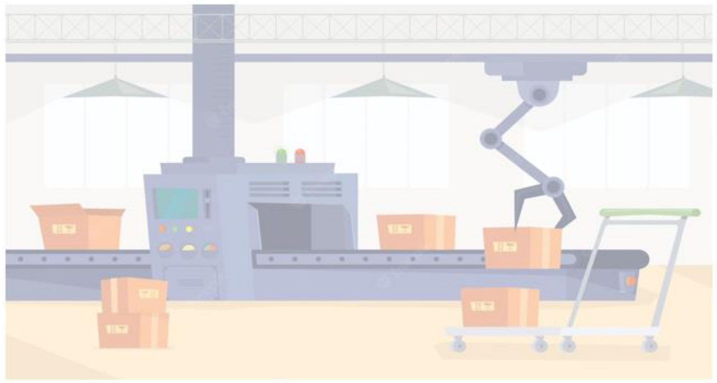
Example: Datapath With Jumps Added



Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file

- What and when the production is stored?



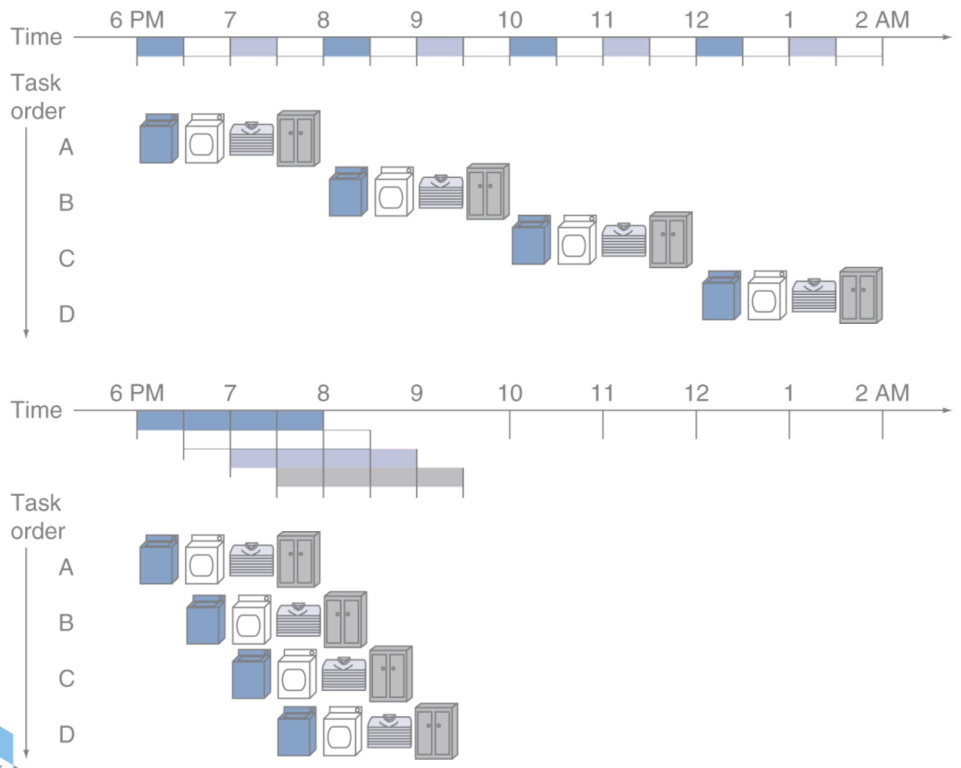
Performance Issues

- 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

$$= \frac{4*n}{4+(n-1)} = \frac{1}{\frac{3+n}{4*n}} \approx 4$$

=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

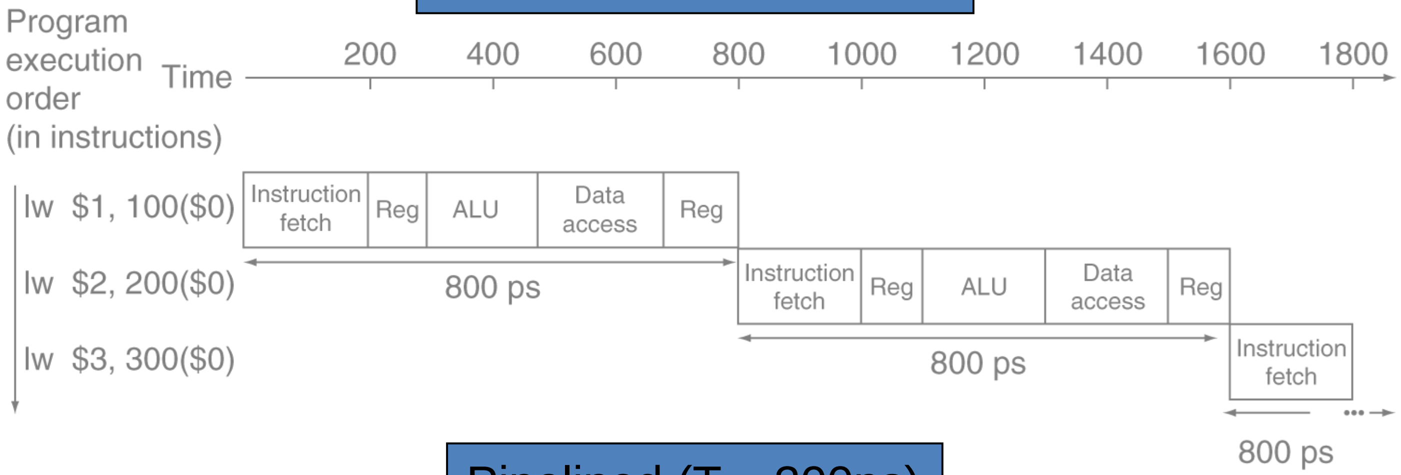
Pipeline Performance

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

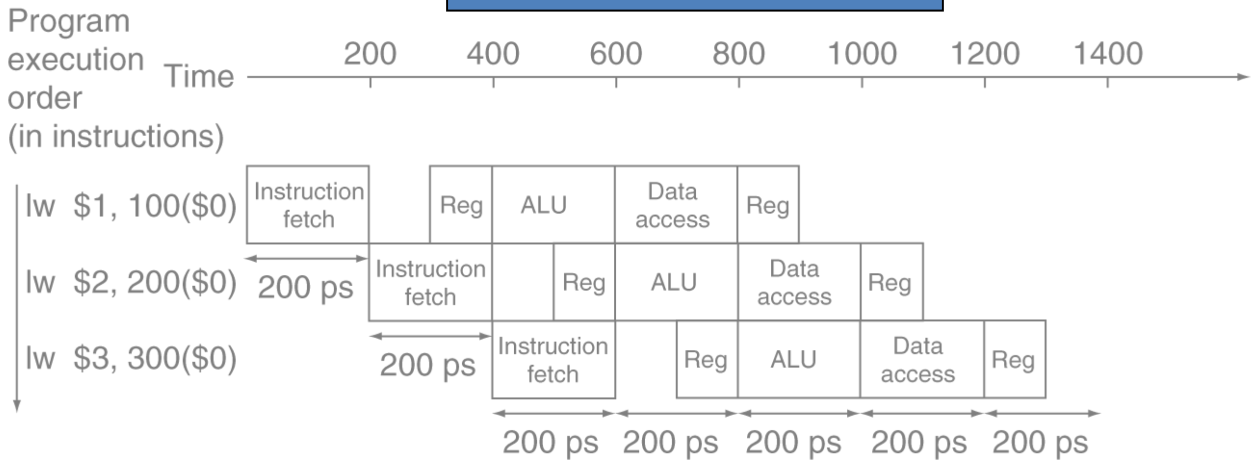
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

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

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

Pipelining and ISA Design

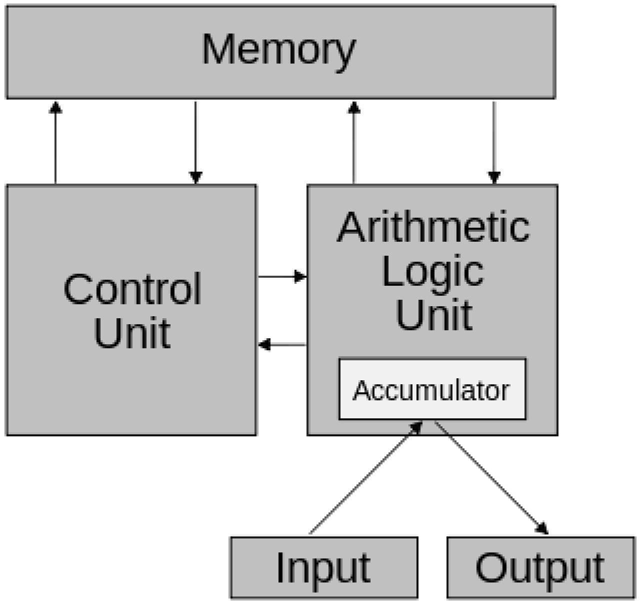
- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in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

Hazards

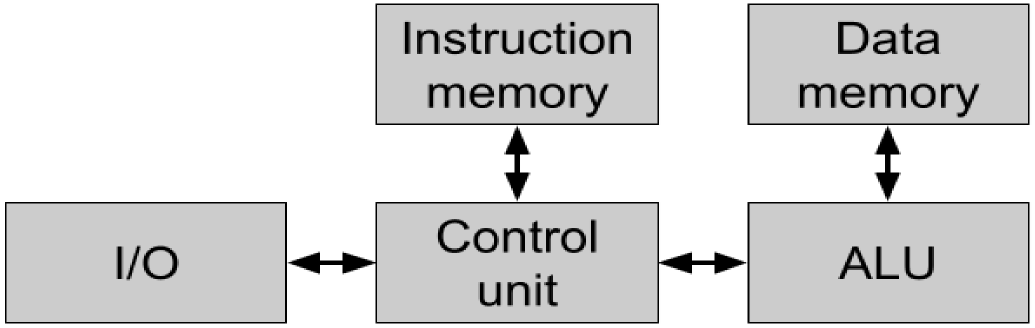
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource



Von Neumann Architecture



Havard architecture

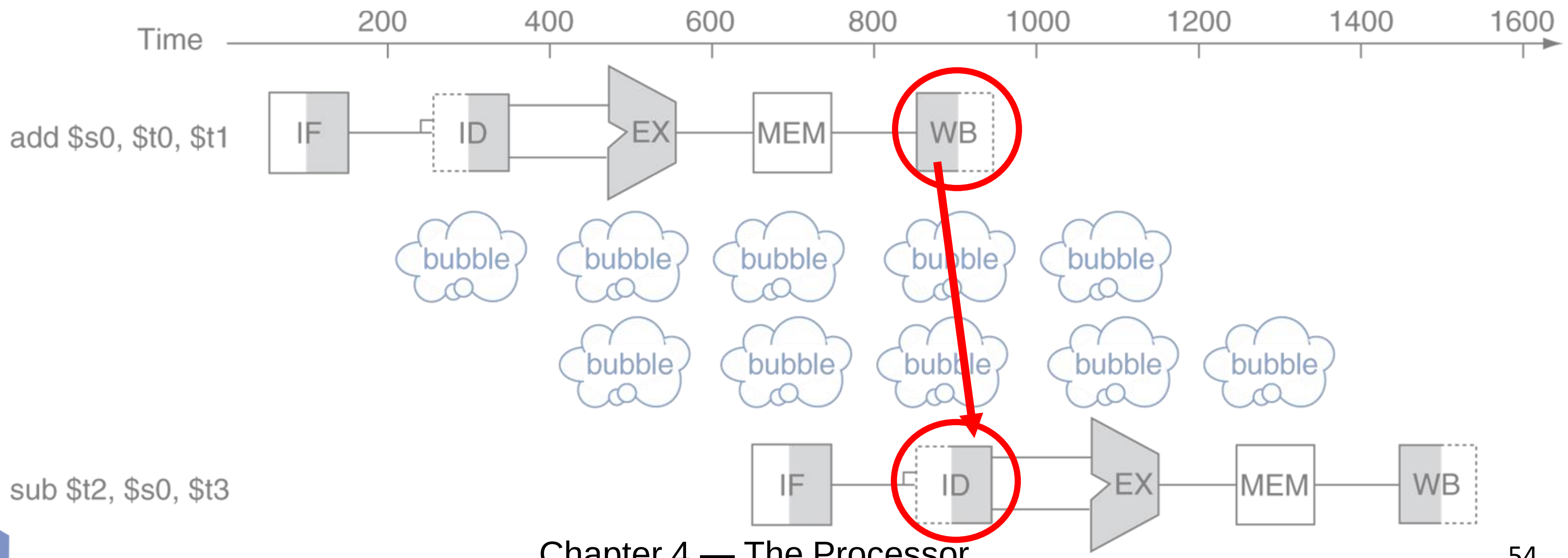
Structure Hazards

- 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

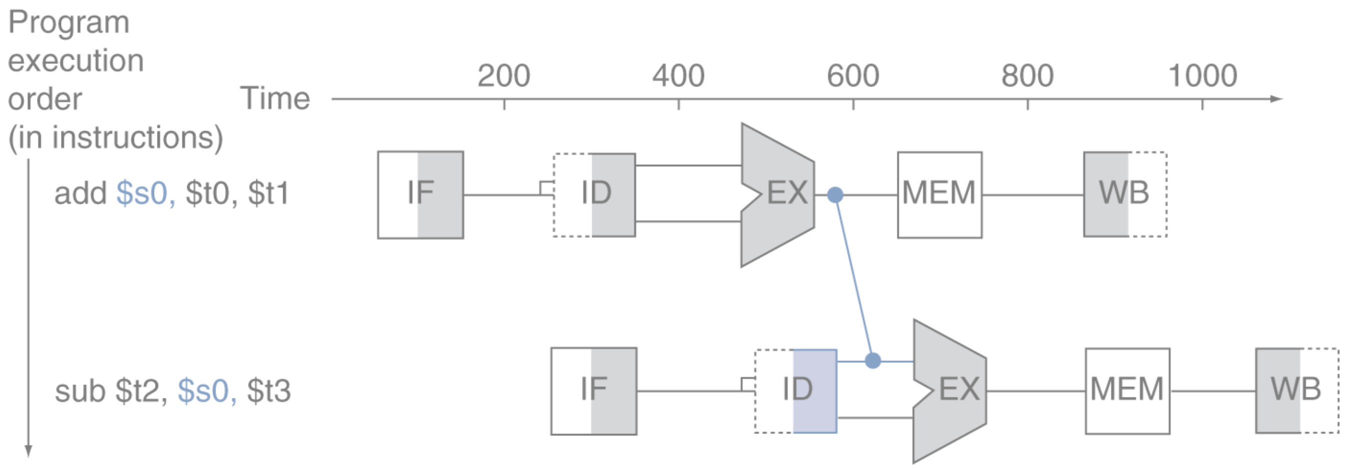
- An instruction depends on completion of data access by a previous instruction

  -
    ```
    add   $s0, $t0, $t1
    sub   $t2, $s0, $t3
    ```



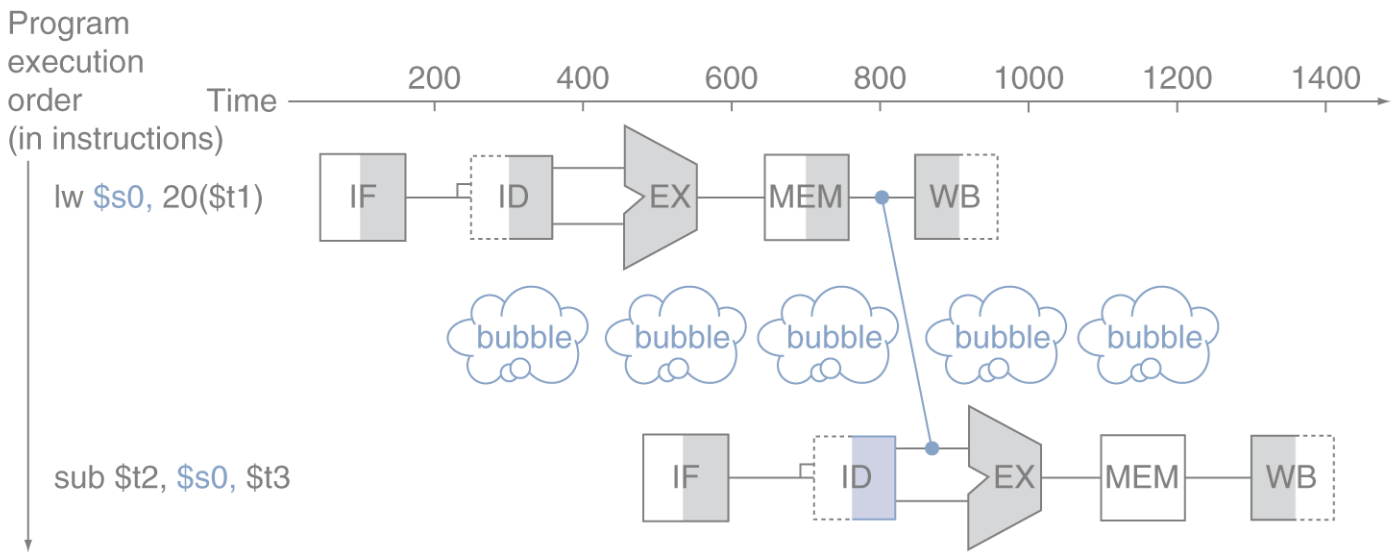
Forwarding (aka Bypassing)

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



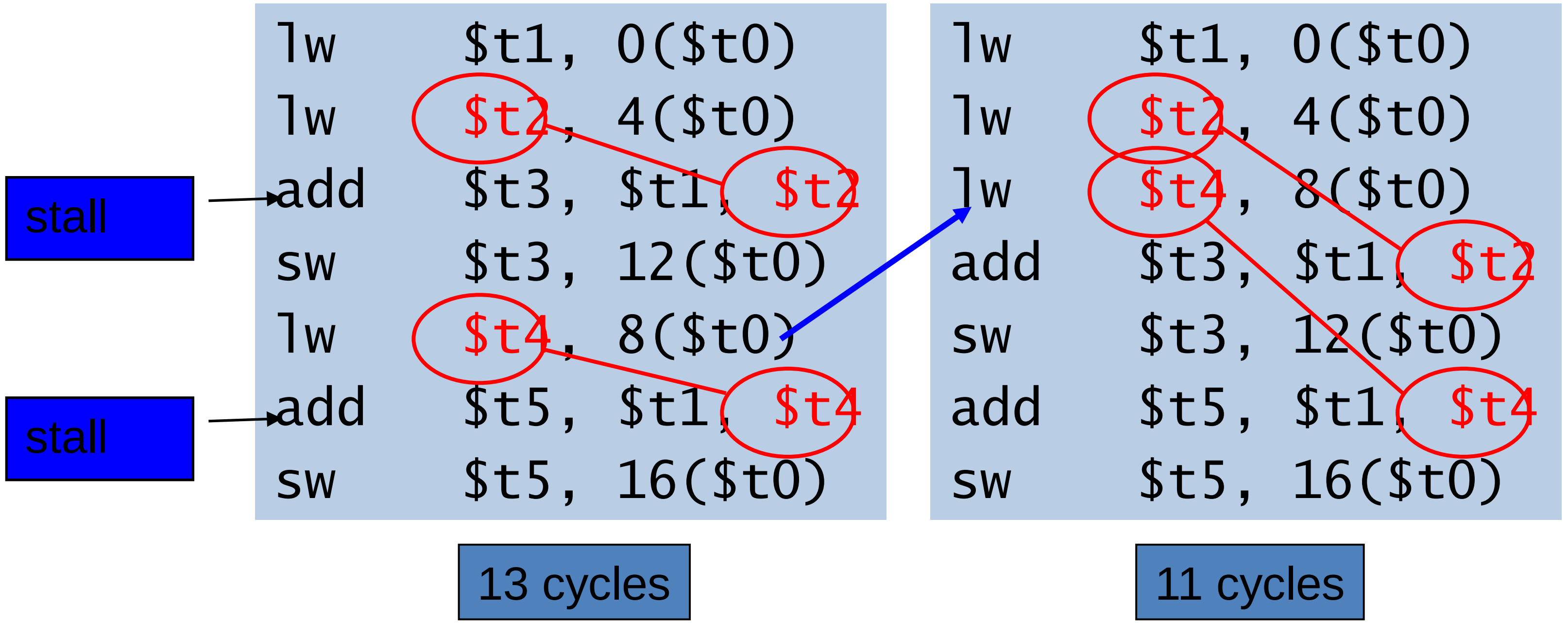
Load-Use Data Hazard

- 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;`

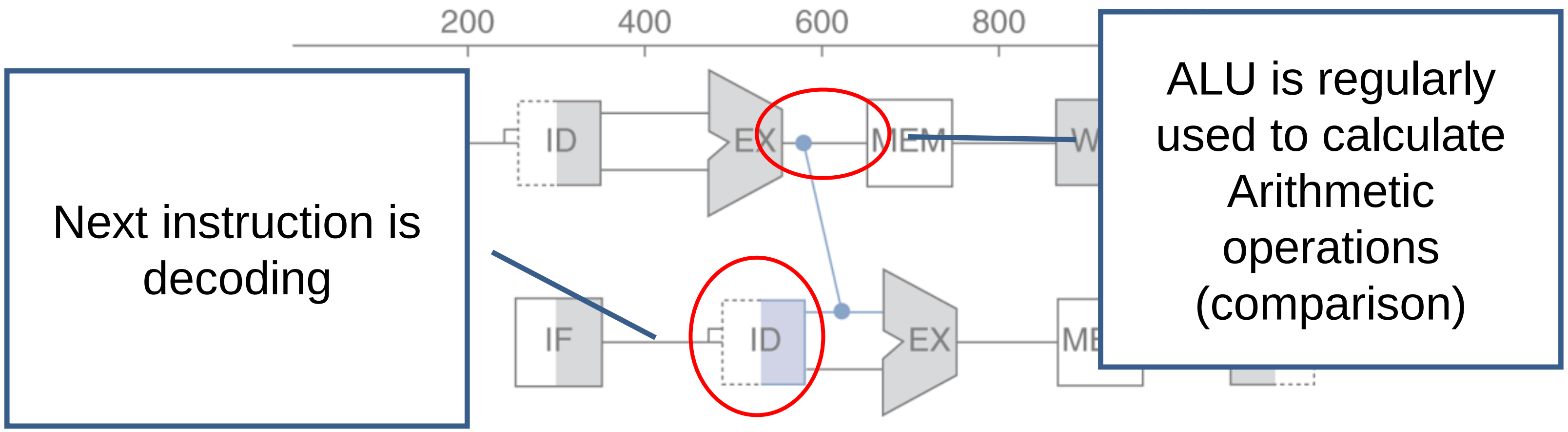


```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1, $t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add     $t5, $t1, $t4
sw      $t5, 16($t0)
```

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
lw      $t4, 8($t0)
add     $t3, $t1, $t2
sw      $t3, 12($t0)
add     $t5, $t1, $t4
sw      $t5, 16($t0)
```

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

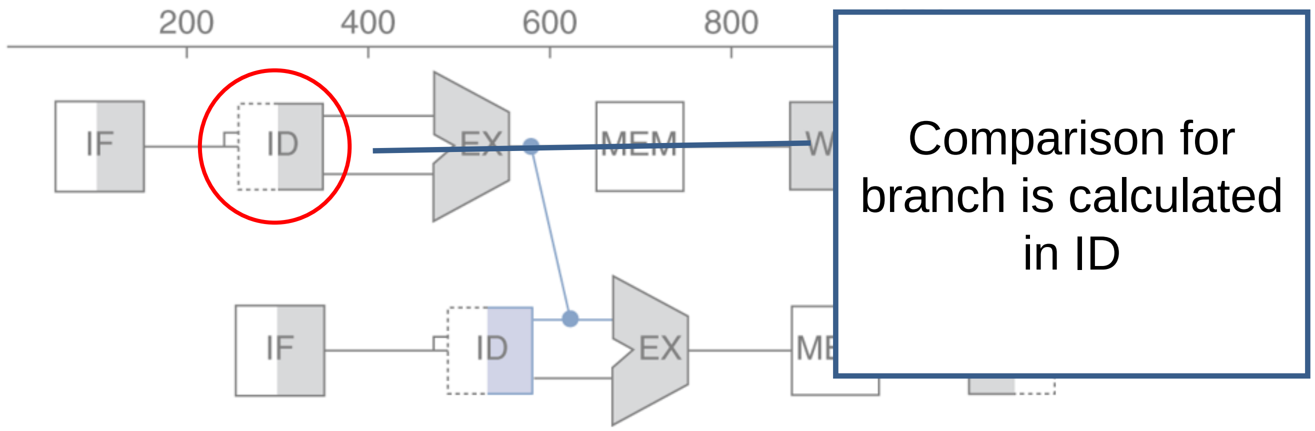


Next instruction is decoding

ALU is regularly used to calculate Arithmetic operations (comparison)

Control Hazards

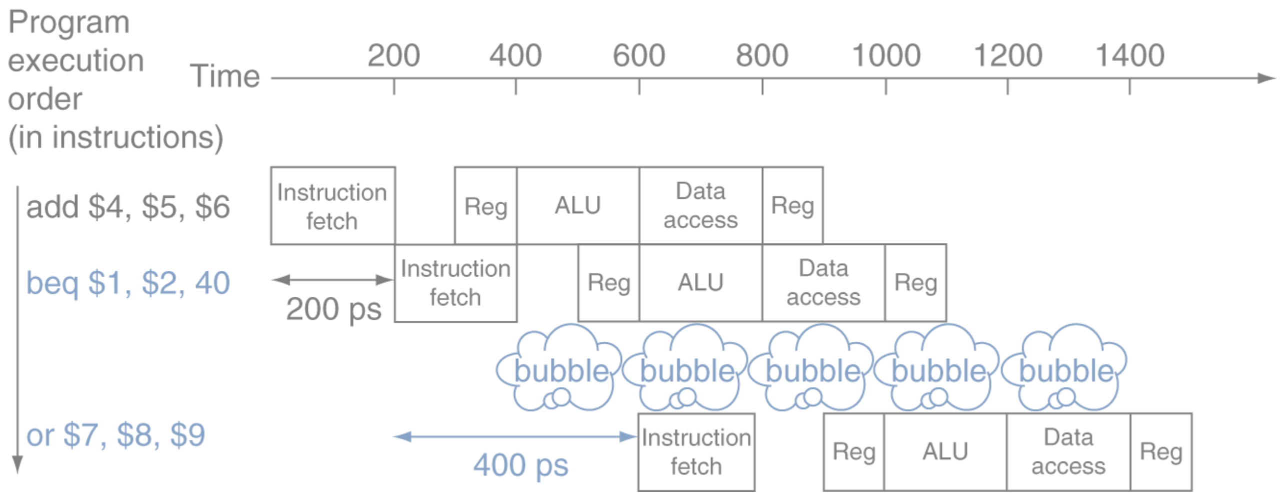
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage



Comparison for branch is calculated in ID

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

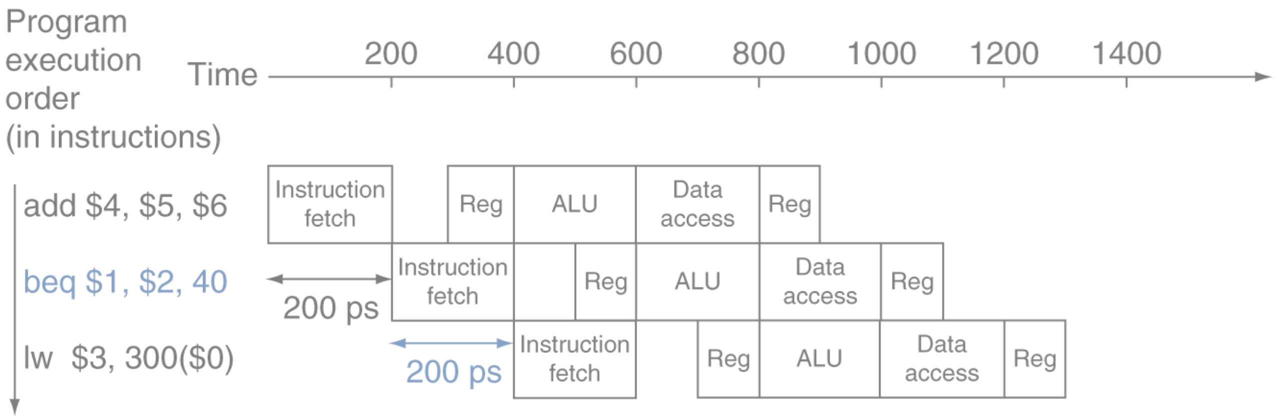


Branch Prediction

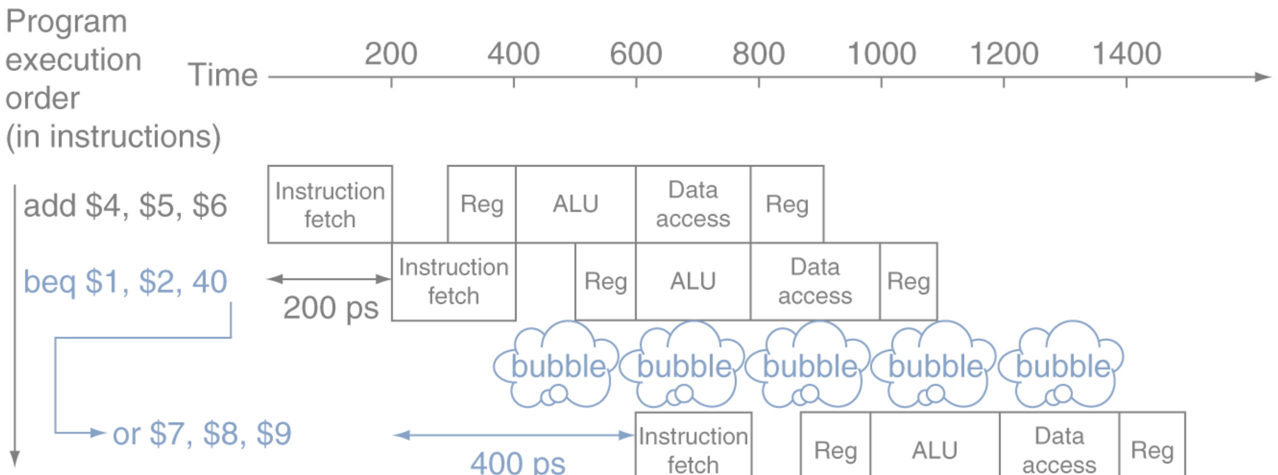
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

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