Chapter 4

The Processor

Introduction

- We will examine two MIPS implementations
  - A simplified version
  - A more realistic *pipelined* version
- Instruction types
  - I-Type: `lw`, `sw, beq, addi, andi,` …
  - R-Type: `add`, `sub`, `and`, `or`, …
  - J-Type: `j, jal`

MIPS Instruction Encoding

R-type — add, sub, and, or, …

opcode 0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

I-Type — lw, sw, beq, addi, andi, …

opcode	rs	rt	Immediate
31:26	25:21	20:16	15:0

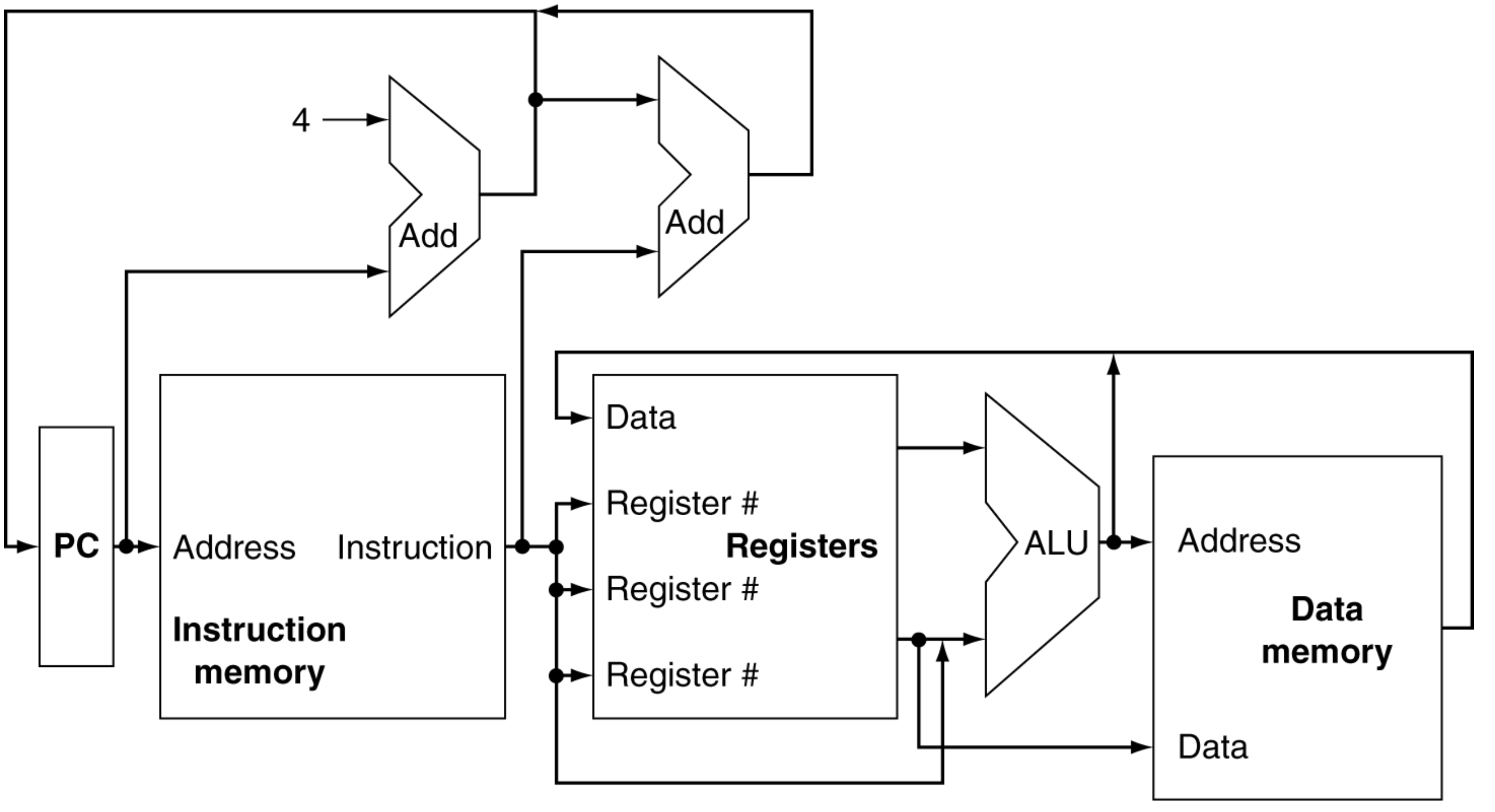
J-Type — J, jal

opcode	address
31:26	25:0

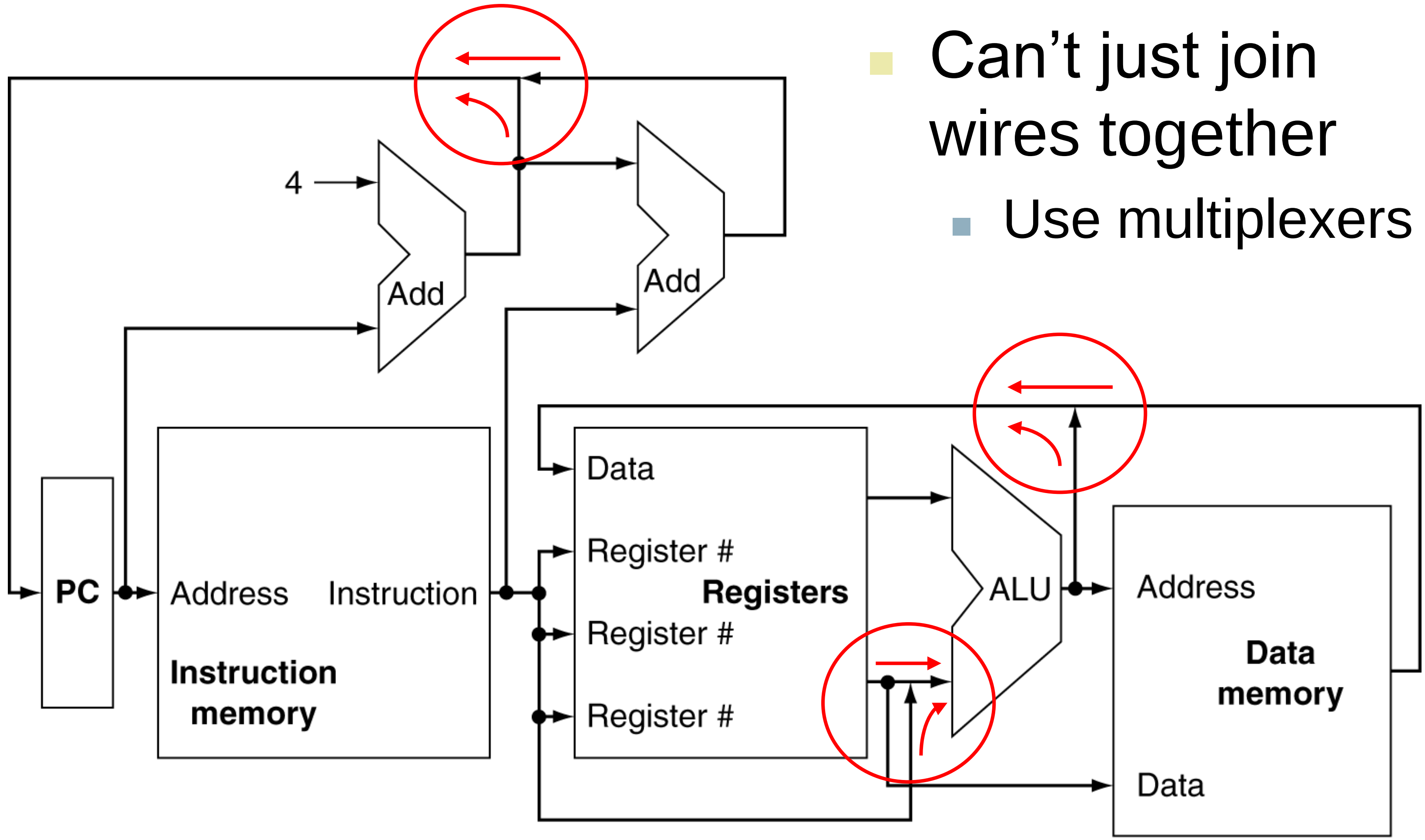
Inside the CPU…

- PC → fetch from instruction memory
- Opcode → identify the format of bits 25:0
  - Register numbers → read from register file (R, I)
  - Immediate → to PC (J, I-branch), to ALU (I-other)
  - Funct & shamt → to ALU (R)
- Depending on the instruction
  - Use ALU to calculate:
    - Arithmetic / logic / shift result
    - Memory address for load / store
  - Access data memory for load/store
  - Write result to register file
  - PC ← Target Address (J, I-branch) or PC + 4

CPU Overview

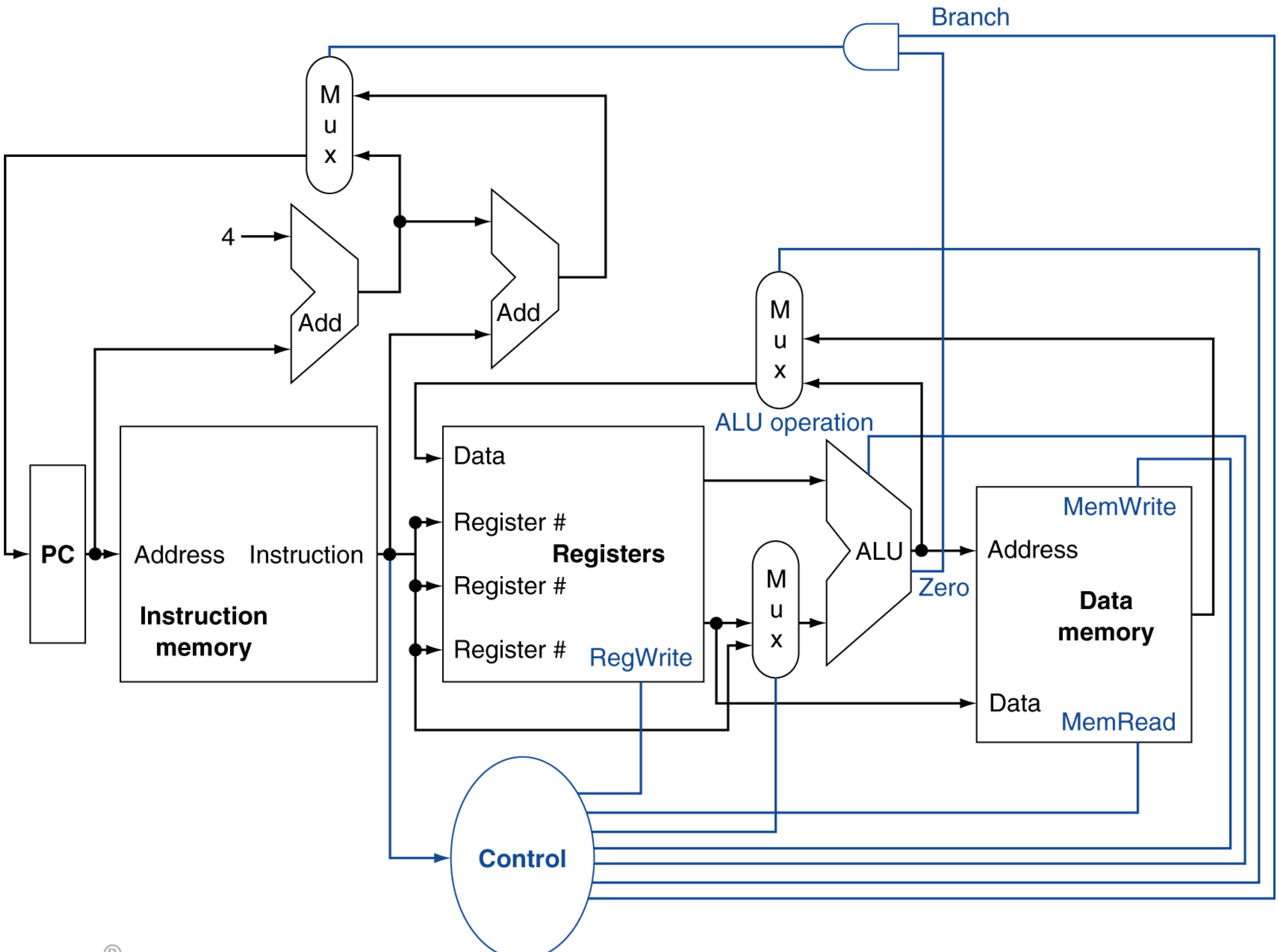


Choices, choices everywhere!



- Can't just join wires together
  - Use multiplexers

Control



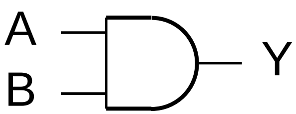
Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
Data
Register #
Registers
Register #
Register #
RegWrite
PC
Address
Instruction
Instruction
memory
ALU
Zero
M
u
x
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

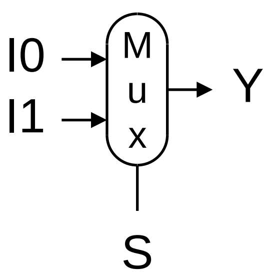
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

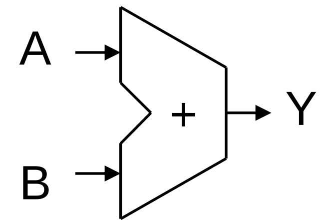
- AND-gate
  - Y = A & B



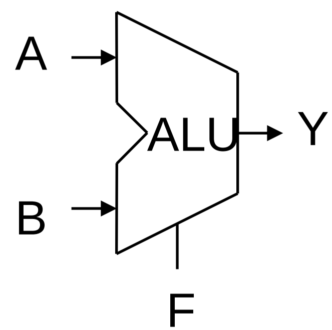
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

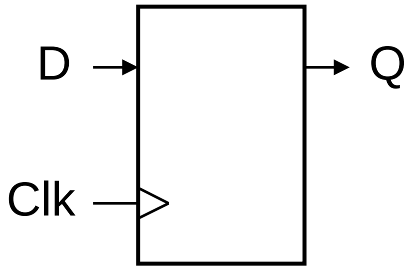


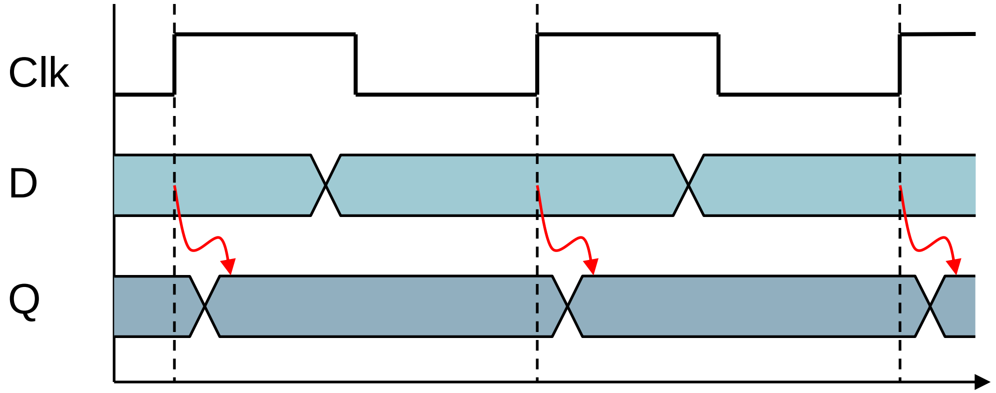
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

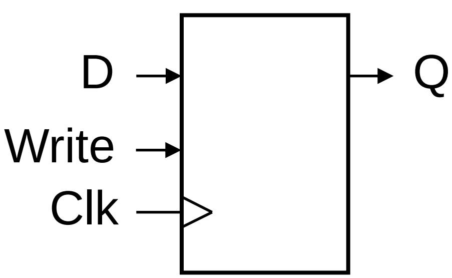
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1 *(or from 1 to 0)*

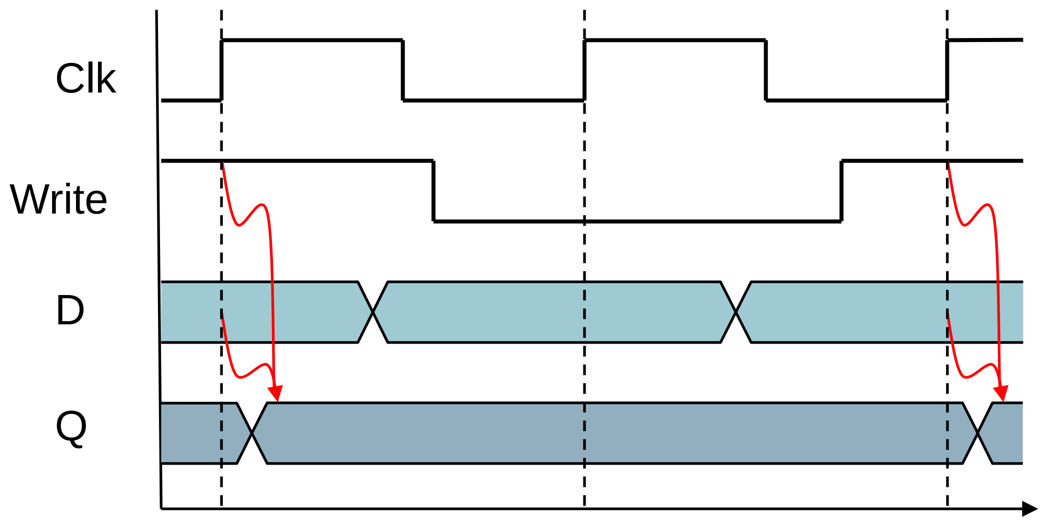




Sequential Elements

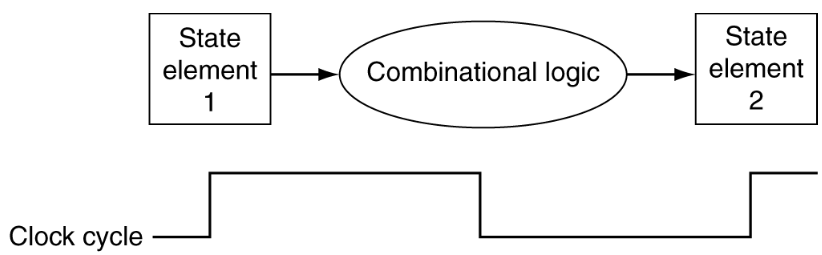
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

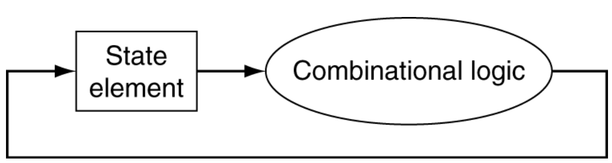




Clocking Methodology

- 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
  - Collection of elements that process data and addresses in the CPU
    - Registers, ALU, multiplexers, …
- We will build a MIPS datapath step by step
  - Refining the overview design
  - Adding control logic as required

Instruction Execution

1 PC → fetch from instruction memory

2 Opcode → identify the format of bits 25:0

- Register numbers → read from register file (R, I)
- Immediate → to PC (J, I-branch), to ALU (I-other)
- Funct & shamt → to ALU (R)

- Depending on the instruction

  3 Use ALU to calculate:

  - Arithmetic / logic / shift result
  - Memory address for load / store

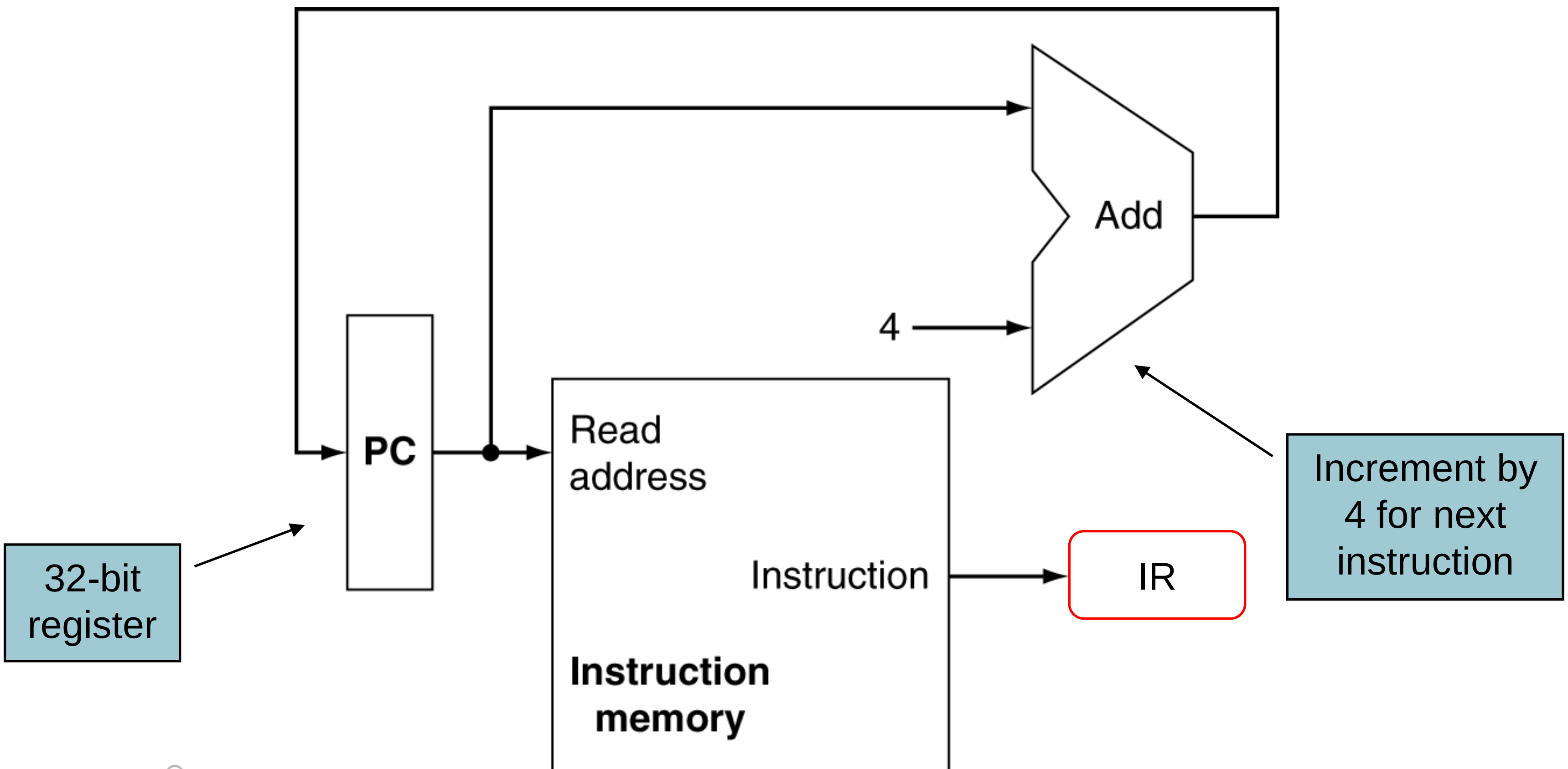
  4 Access data memory for load/store

  5 Write result to register file

  6 PC ← Target Address (J, I-branch) or PC + 4

Instruction Fetch

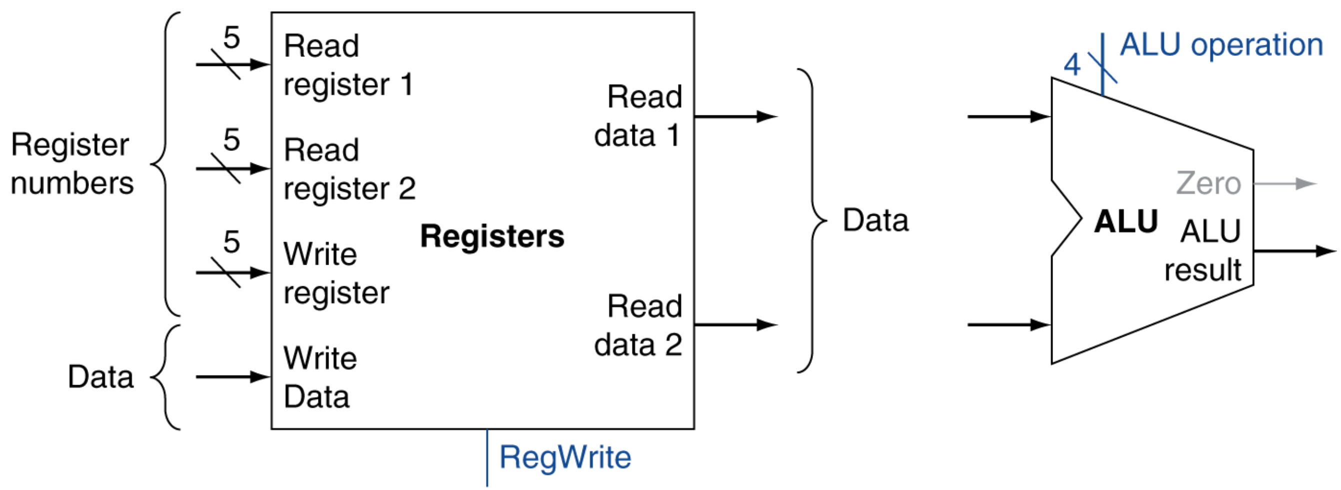
What are the components needed?



R-Type Instructions

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

	opcode 0	rs	rt	rd	shamt	funct
R-type	31:26	25:21	20:16	15:11	10:6	5:0



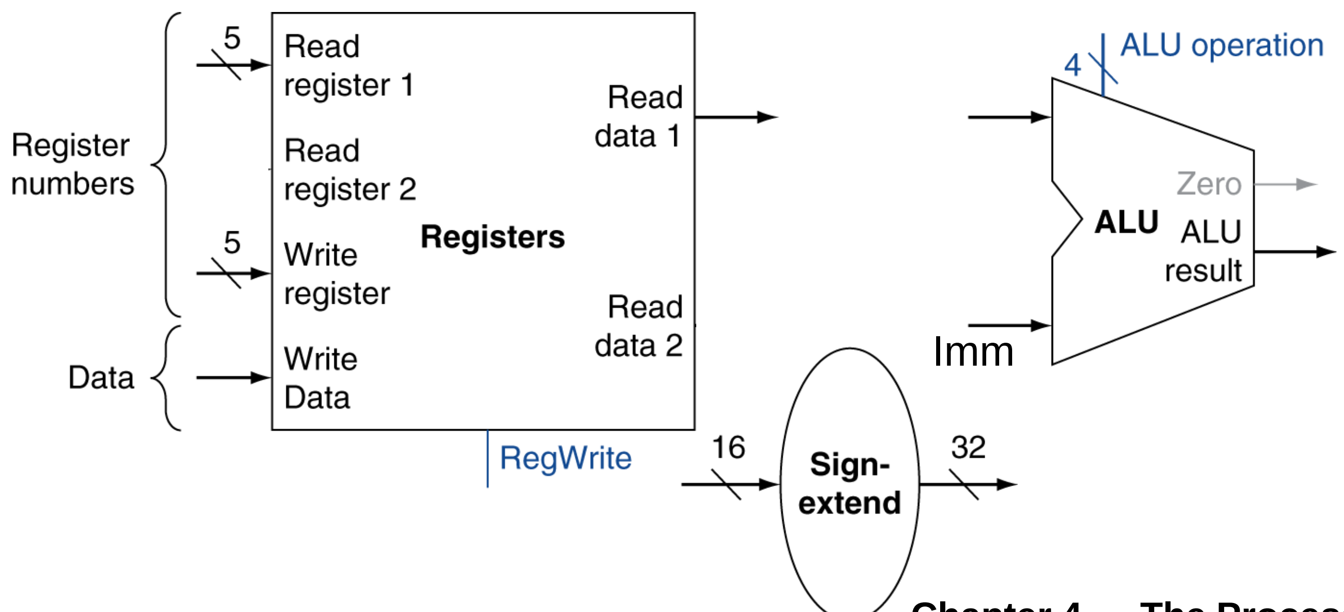
a. Registers

b. ALU

I-Type arith/logic Instructions

- Read **one** register operand
- Perform arithmetic/logical operation
- Write result to register

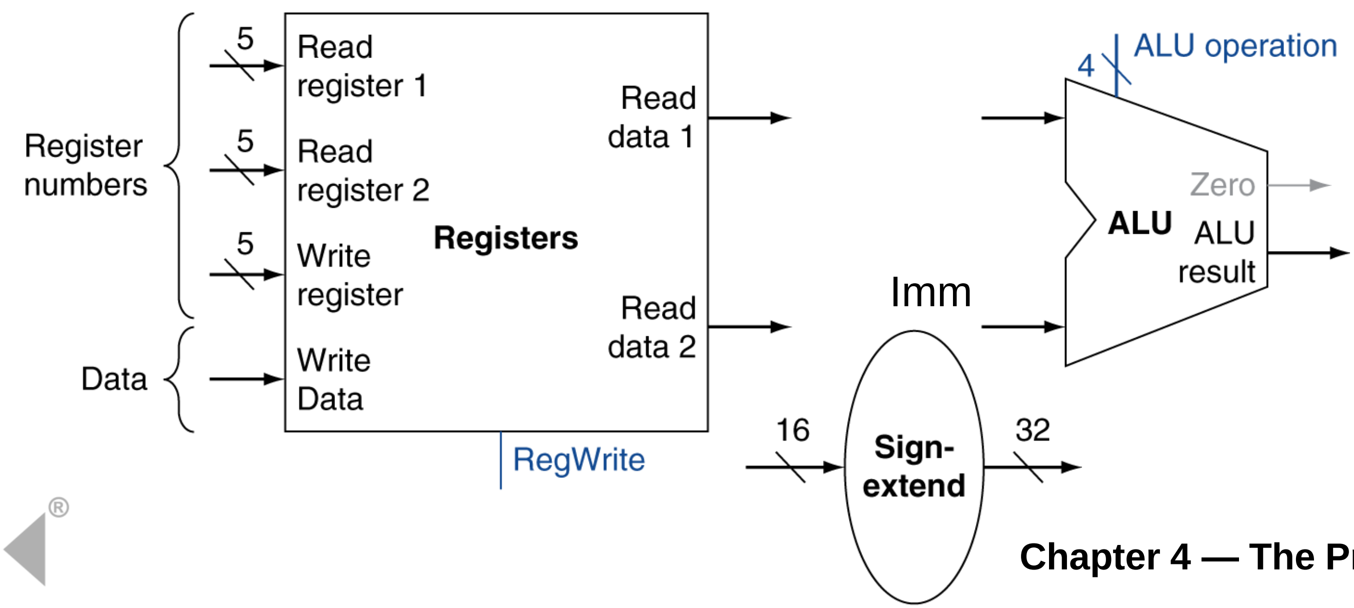
I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0



I-Type Load/Store Instructions

- Read register operand (base register)
- Calculate address using 16-bit offset
  - Use ALU, but need to sign-extend offset
- Load: Read from memory and update register
- Store: Write from register to memory

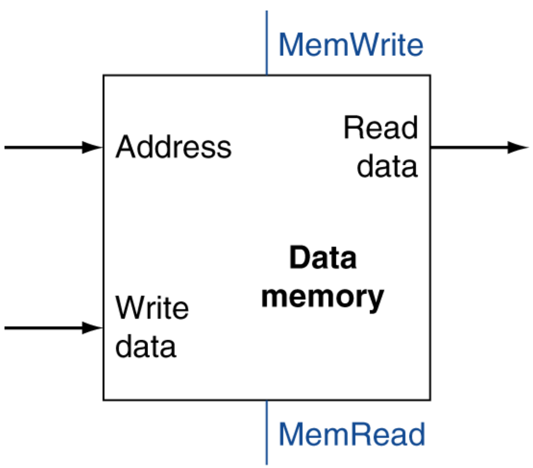
I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0

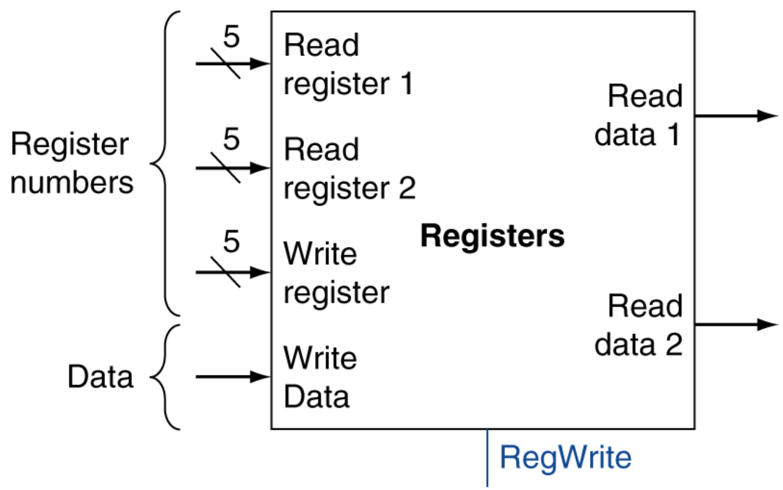


I-Type Load/Store Instructions

- Read register operand (base register)
- Calculate address using 16-bit offset
  - Use ALU, but need to sign-extend offset
- Load: Read from memory and update register
- Store: Write from register to memory

I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0



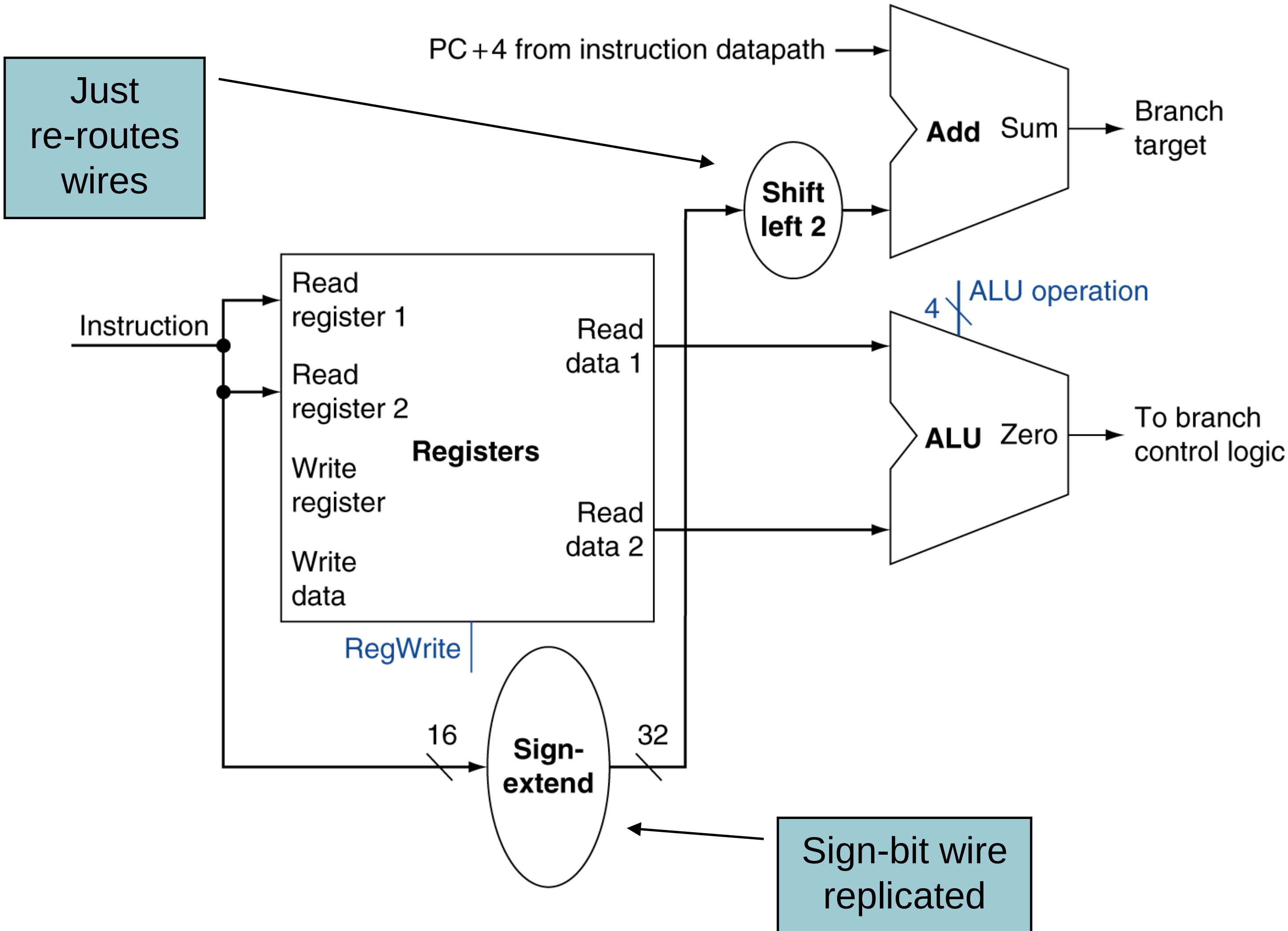


I-Type Branch Instructions

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

I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0

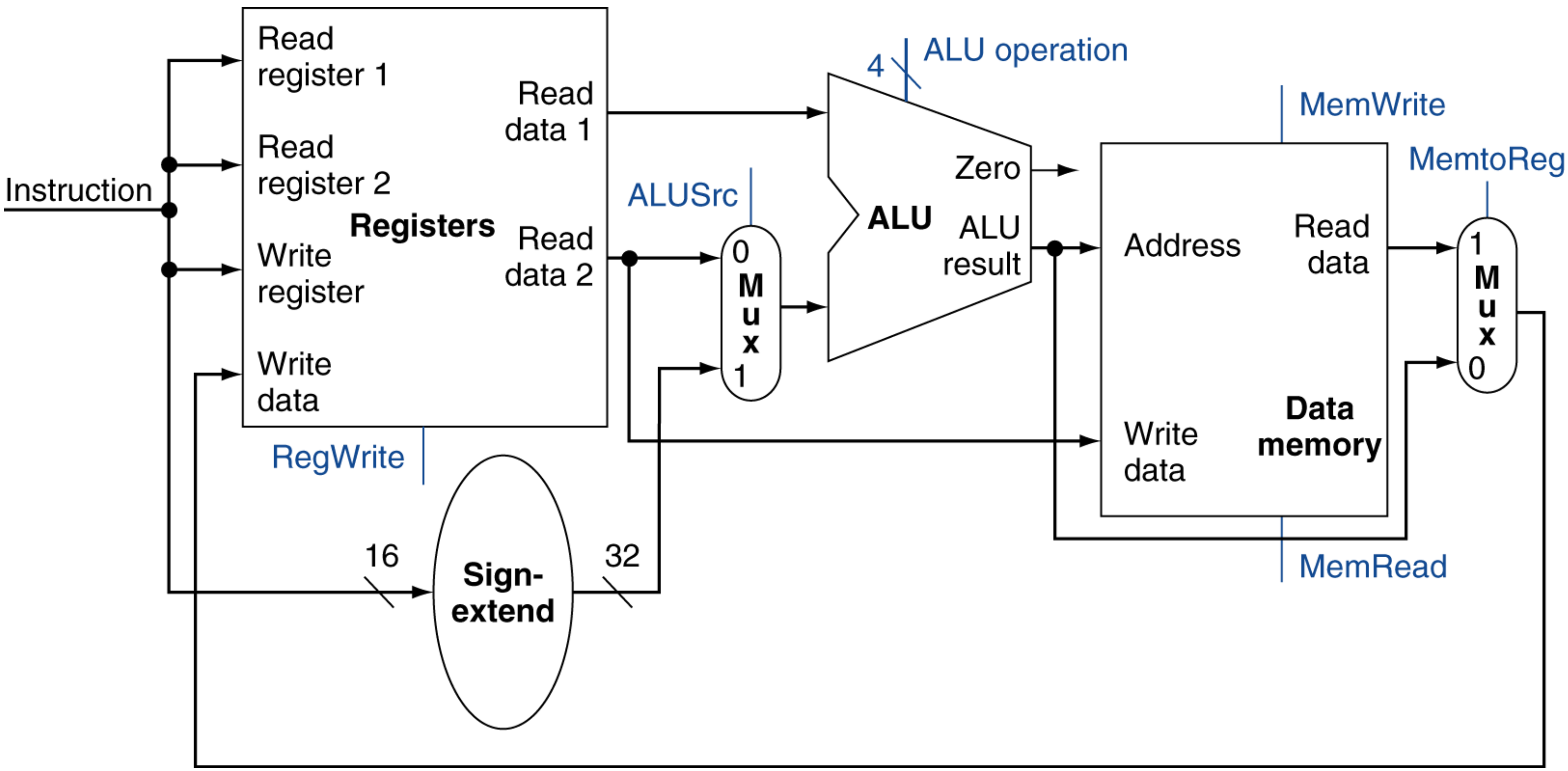
I-Type Branch Instructions



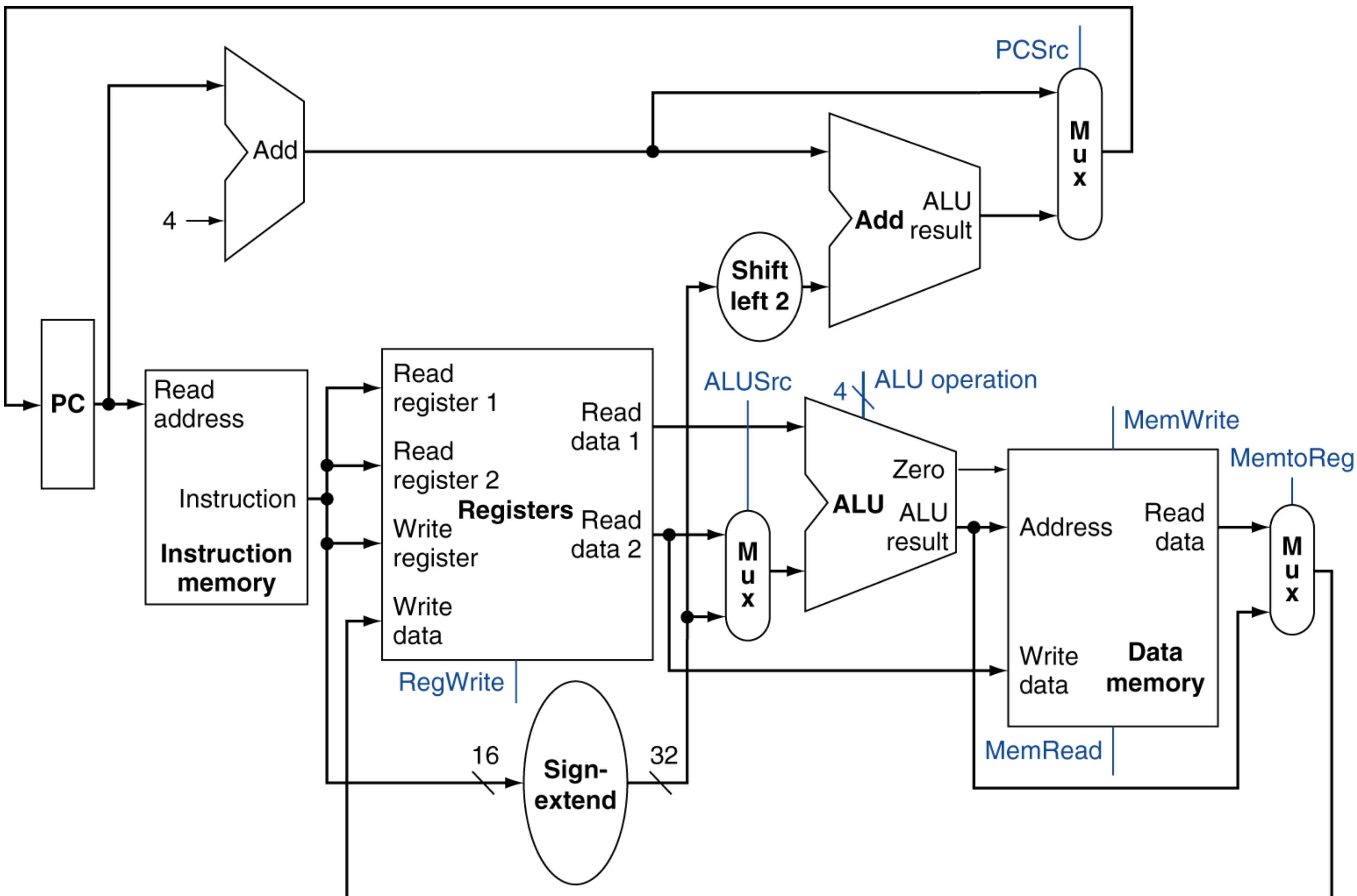
Composing the Elements

- 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

Arith/Logic/Load/Store Datapath



Full Datapath



ALU Control (lw/sw/beq/R-Type)

- ALU used for
  - lw / sw: Function = add
  - beq: Function = subtract
  - R-type: Function depends on “funct” field

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

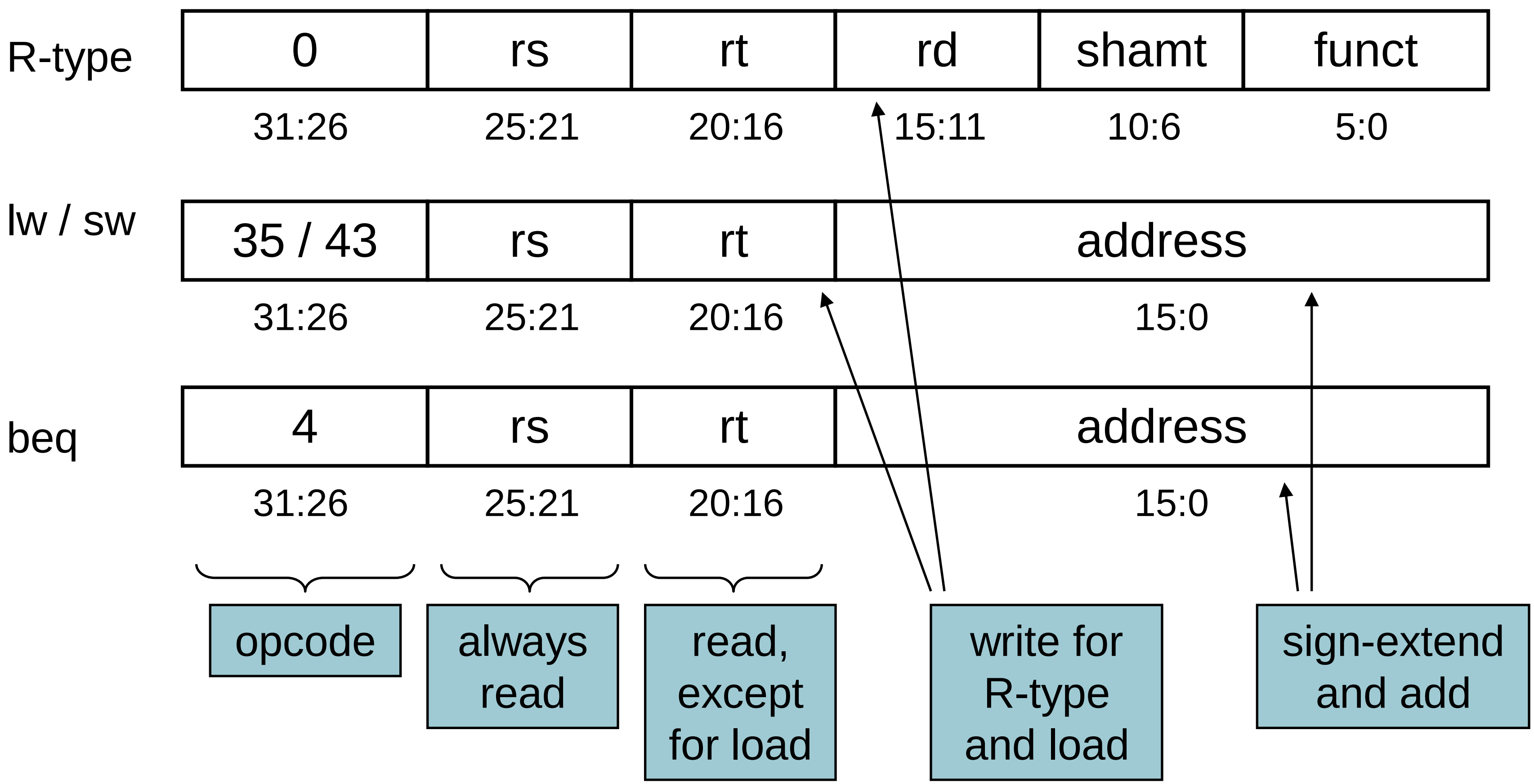
ALU Control

- 2-bit *ALUOp* signal derived from *opcode*
  - ALU control derived from *ALUOp* and *funct*
  - Try to find combinational logic functions for *ALUOp* and *funct* signals, for the instructions given below:

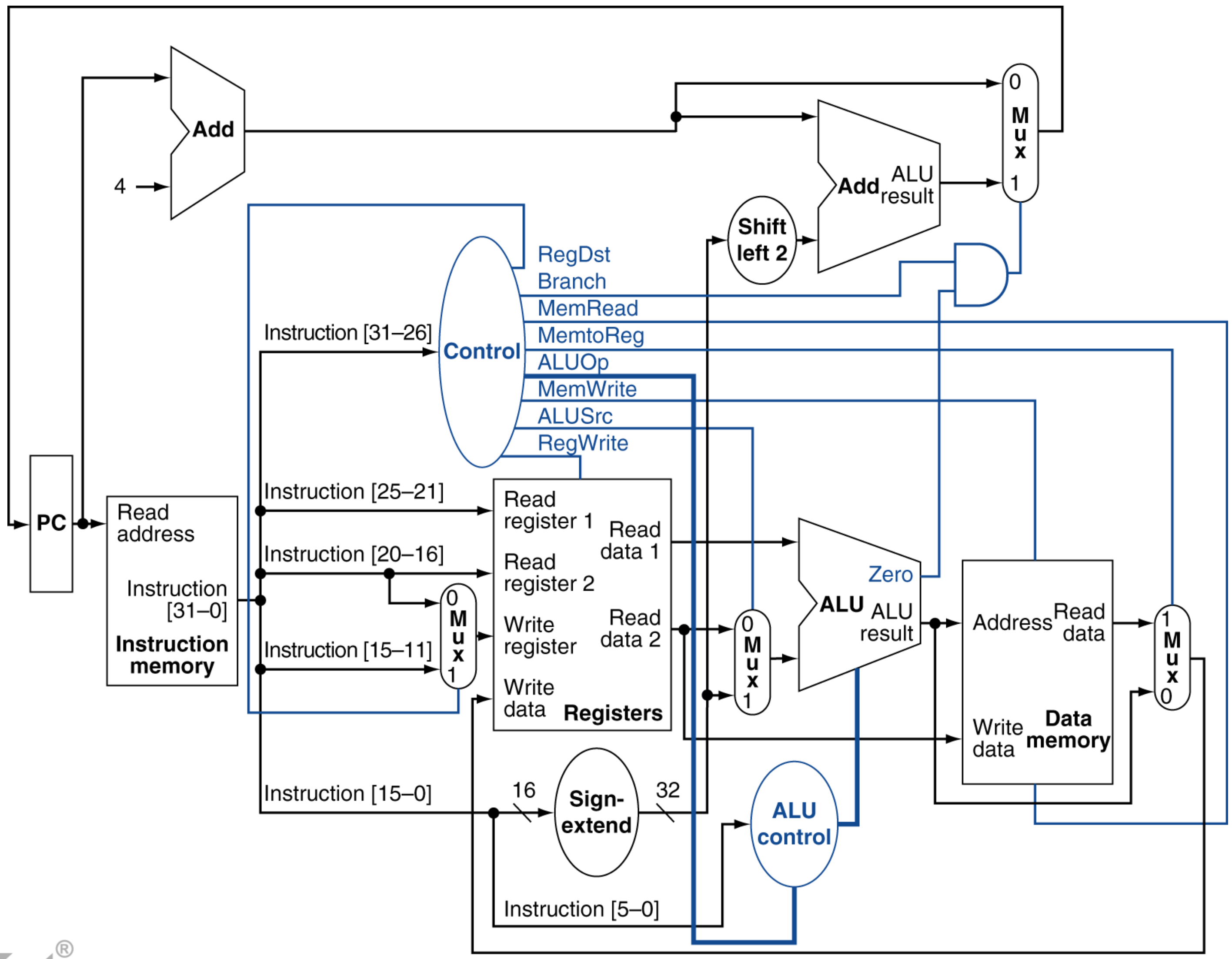
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw 100011	00	load word	XXXXXX	add	0010
sw 101011	00	store word	XXXXXX	add	0010
beq 000100	01	branch equal	XXXXXX	subtract	0110
R-type 000000	10 OR 1x	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction

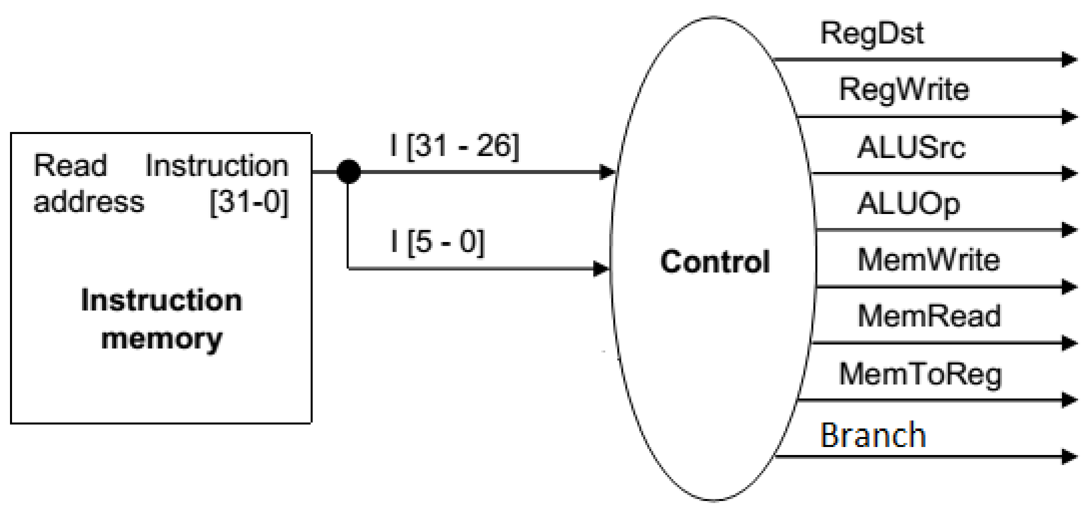


Datapath With Control



Generating Control Signals

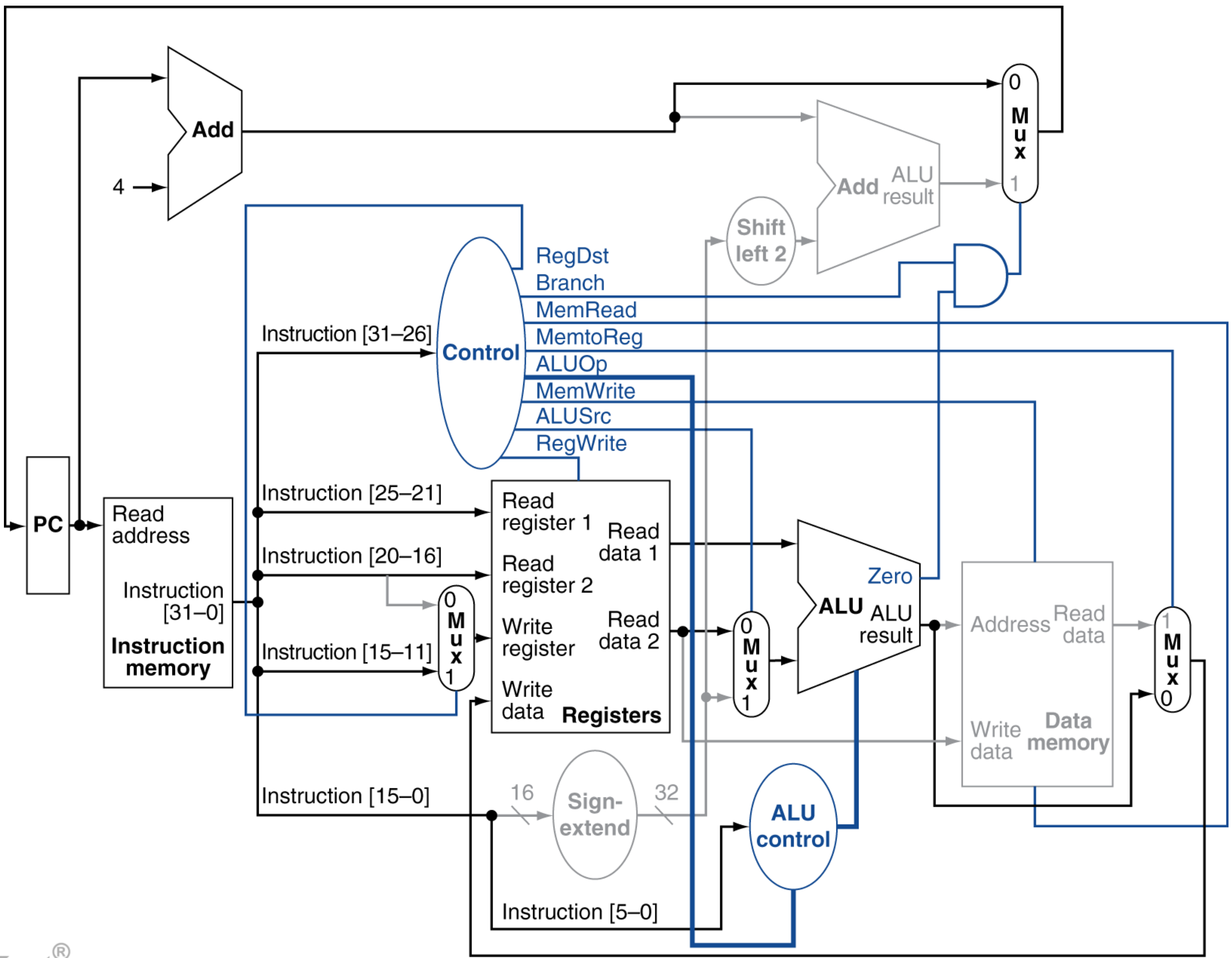
- The control unit needs 12 bits of inputs.
  - Six bits make up the instruction's **opcode**.
  - Six bits come from the instruction's **func** field.
- The control unit generates 9 bits of output, corresponding to the blue control signals mentioned on the previous slide
- You can build the actual circuit by using Boolean algebra



Generating Control Signals

- The control unit is responsible for setting all the control signals so that each instruction is executed properly.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.
- To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.

R-Type Instructions



R-Type Instructions

add, sub, and, or, …

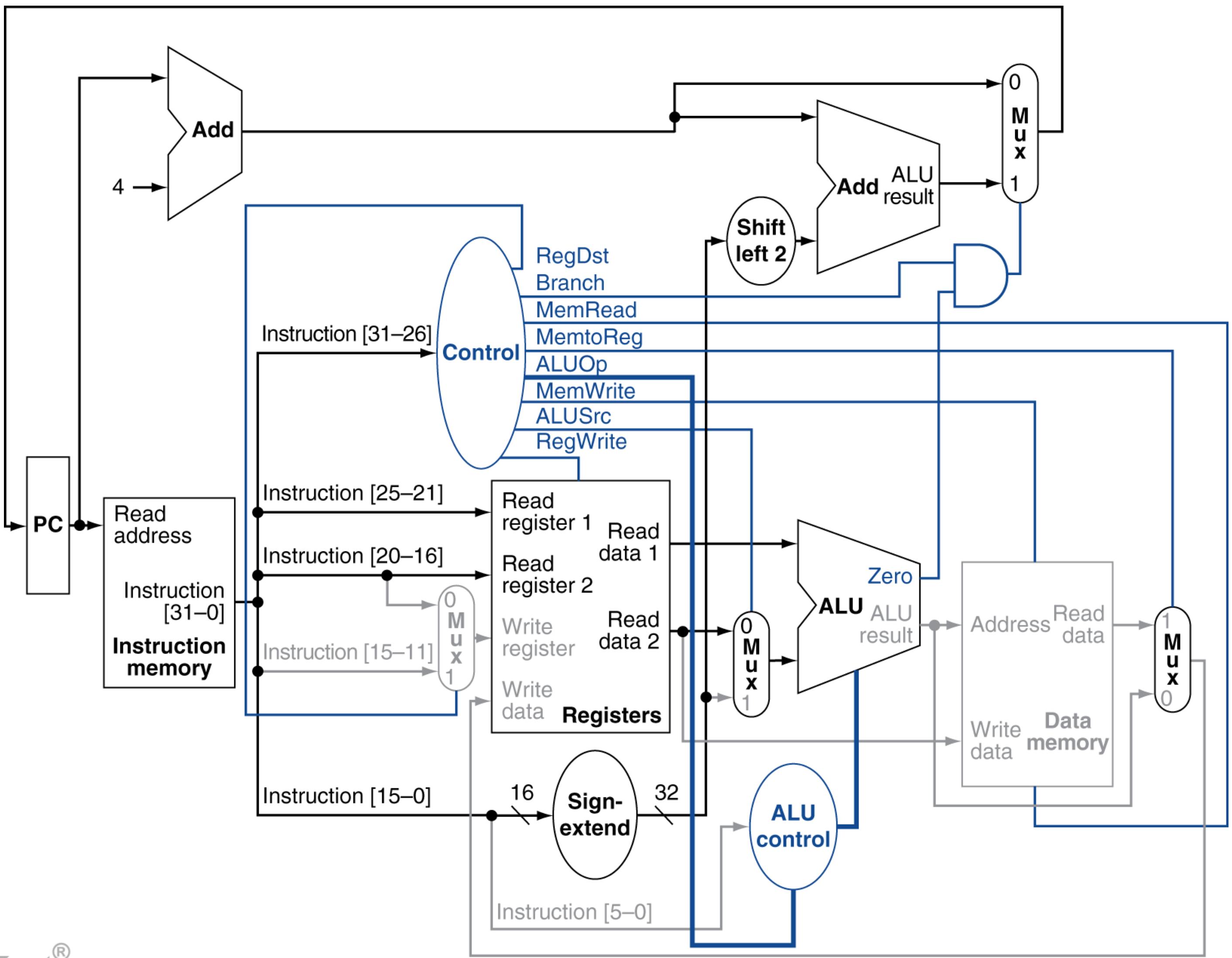
R-type	opcode 0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0

add $1, $2, $3

add	000000	00010	00011	00001	00000	100000

RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Branch-if-Equal Instruction



Branch-if-Equal Instruction

`lw, sw, beq, addi, andi, …`

I-Type

opcode	rs	rt	Immediate
31:26	25:21	20:16	15:0

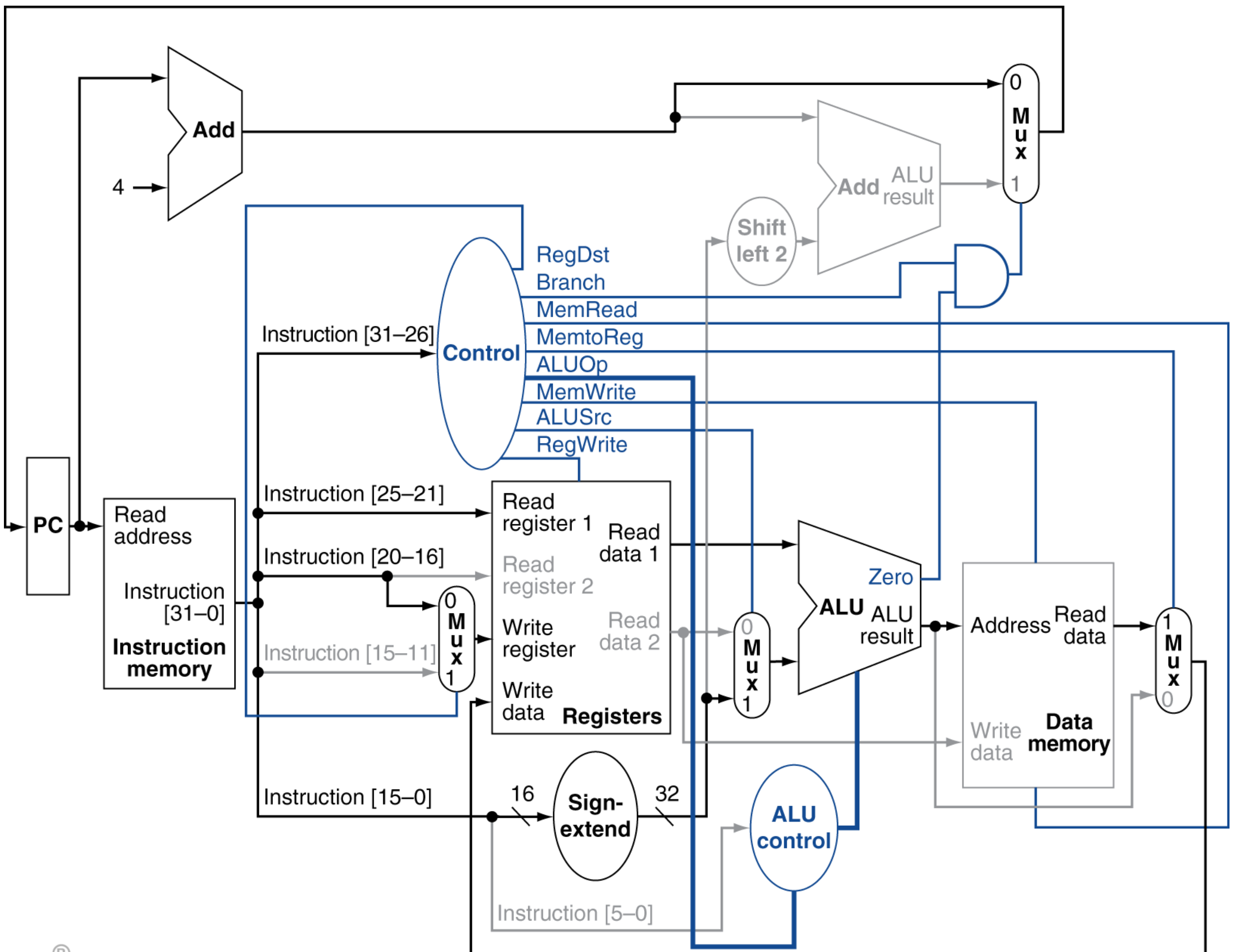
`beq $1, $2, 100`

beq

000100	00001	00010	0000000001100100

RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Load Word Instruction



Load Word Instruction

`lw`, `sw`, `beq`, `addi`, `andi`, …

I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0

`lw $8, 32($9)`

lw	100011	01001	01000	0000000000100000

RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Store Word Instruction

Draw the relevant data path elements and control signals which are used by a Store Word instruction (sw)

Store Word Instruction

lw, sw, beq, addi, andi, …

I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0

sw $8, 32($9)

sw	100111	01001	00100	0000000001100100

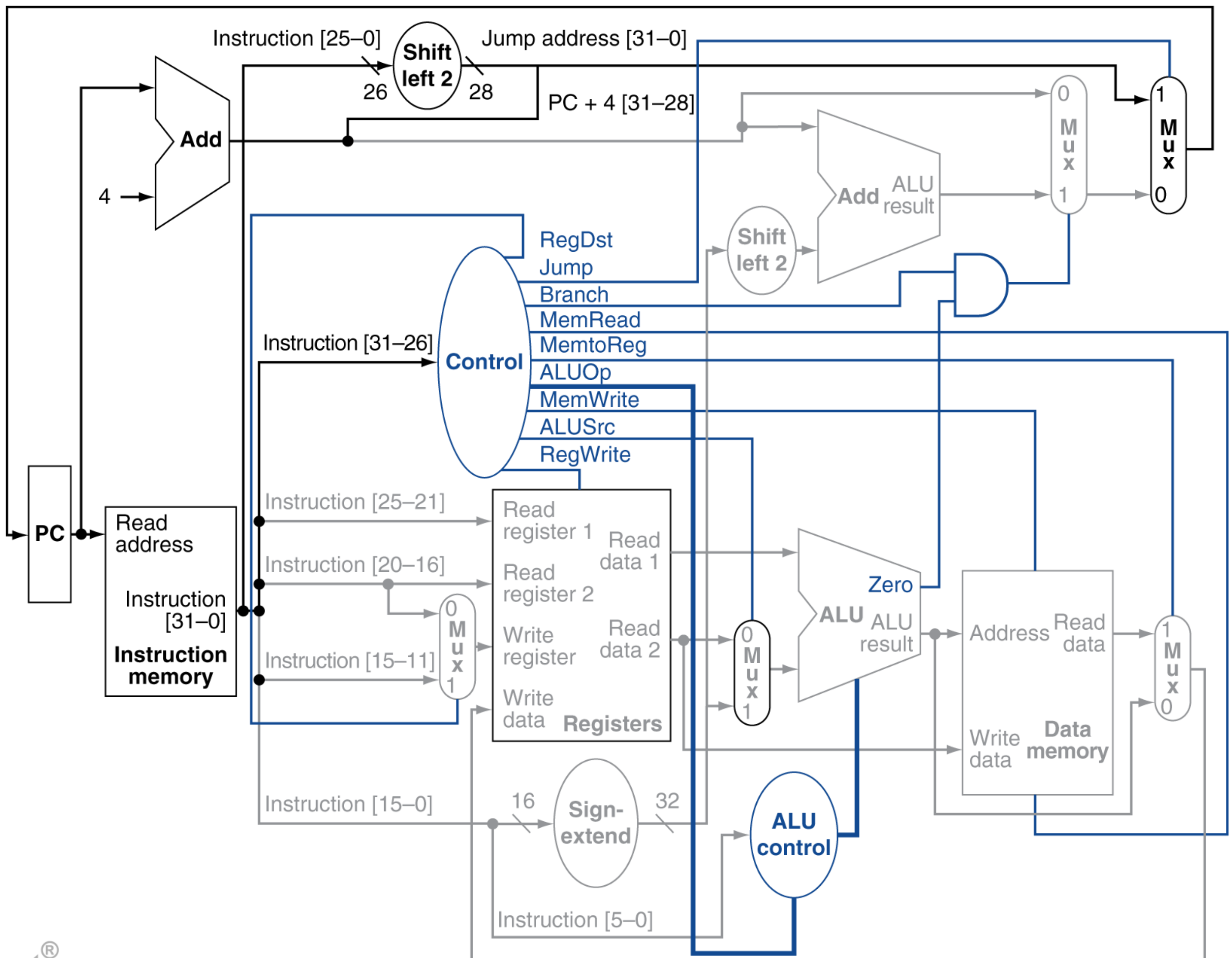
RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Implementing Jumps

Jump (j)	2	address
	31:26	25:0

- Jump uses word address (size 26-bits)
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal generated using opcode

Datapath With Jumps Added



Single-Cycle Implementation

- A datapath contains all the functional units and connections necessary to implement an instruction set architecture.
  - For our single-cycle implementation, we use two separate memories, an ALU, some extra adders, and lots of multiplexers.
  - MIPS is a 32-bit machine, so most of the buses are 32-bits wide.
- The control unit tells the datapath what to do, based on the instruction that's currently being executed.
  - Our processor has 9 control signals that regulate the datapath.
  - The control signals can be generated by a combinational circuit with the instruction's 32-bit binary encoding as input.

The Datapath & the Clock (1)

1. On a positive clock edge, the PC is updated with a new address.
2. A new instruction can then be loaded from memory. The control unit sets the datapath signals appropriately so that
   - registers are read,
   - ALU output is generated,
   - data memory is read, and
   - branch target addresses are computed.

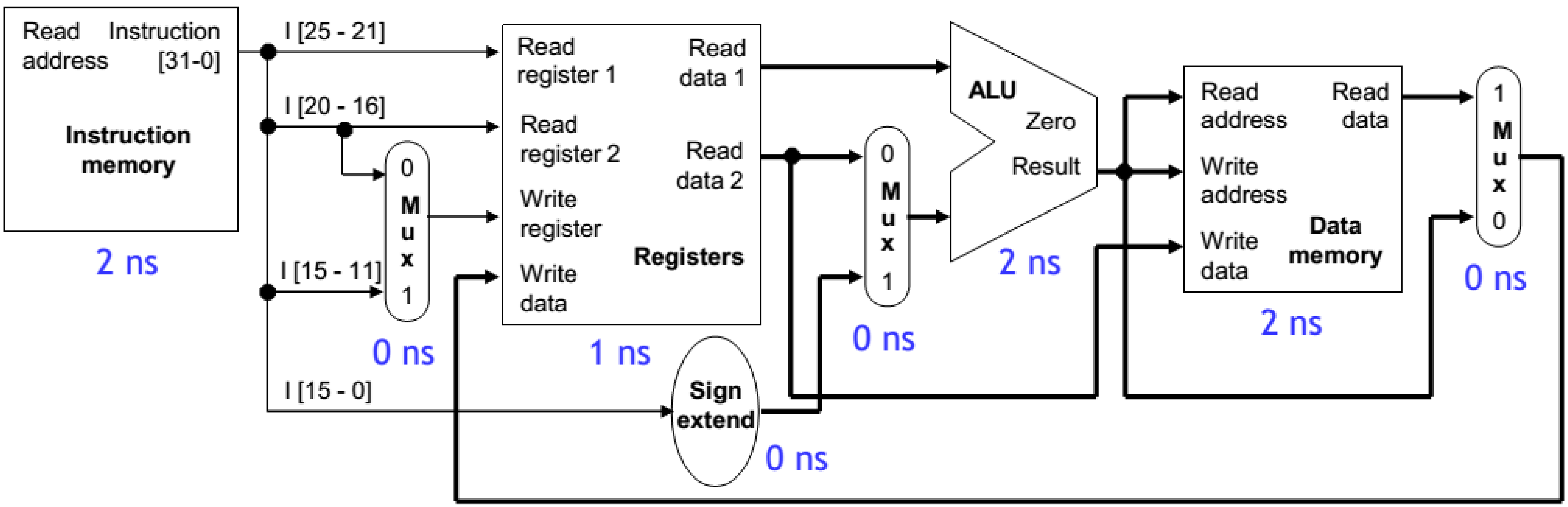
The Datapath & the Clock (2)

3. Several things happen on the next positive clock edge.
   - The register file is updated for arithmetic or lw instructions.
   - Data memory is written for a sw instruction.
   - The PC is updated to point to the next instruction.

In a single-cycle datapath everything in Step 2 must complete within one clock cycle, before the next positive clock edge.

Computing the longest (critical) path

- Calculate the instruction latencies for all the instructions we have, assuming the circuit latencies given below:



lw	sw	add	sub	and	or	beq	j

Average Instruction Latency

- Let's consider the gcc instruction mix as in the following table. Calculate the average instruction latency.

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

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, in a single-cycle CPU
- Violates design principle
  - Making the common case fast
- We can improve performance with a multi-cycle implementation

Multi-Cycle CPU

- Different instructions consume different number of clock cycles
  - Arrange the datapath into “stages”
  - Fetch -> Reg Read -> ALU -> Mem Access -> Reg Write
  - Each stage to complete within one (smaller) clock cycle
  - Not all instructions will use all stages
- How to decide clock period?
  - Slowest stage determines the clock period
  - Define stages such that their latencies are equal (roughly)
- Can we improve performance even more?

Yes! Using a technique called “Pipelining”