Chapter 4

The Processor

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 ISA subset, covering most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

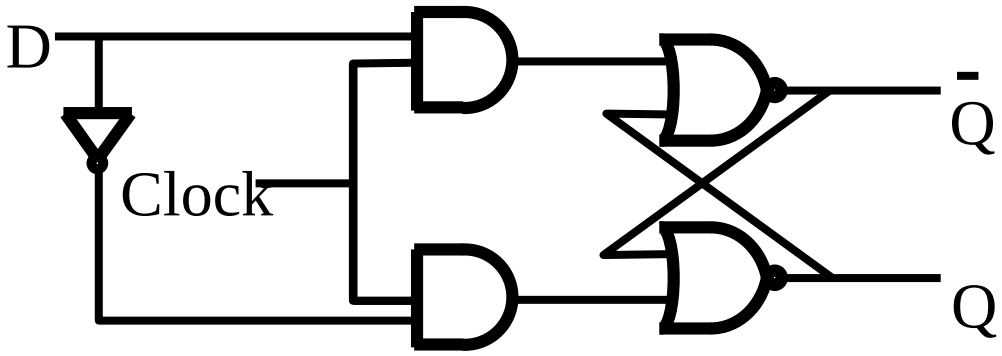
Datapath

- The *datapath* is the interconnection of the components that make up the processor
  - Includes elements that process data and addresses within the CPU
- The datapath must provide connections for moving bits from and to the memory, registers, and ALU

Control

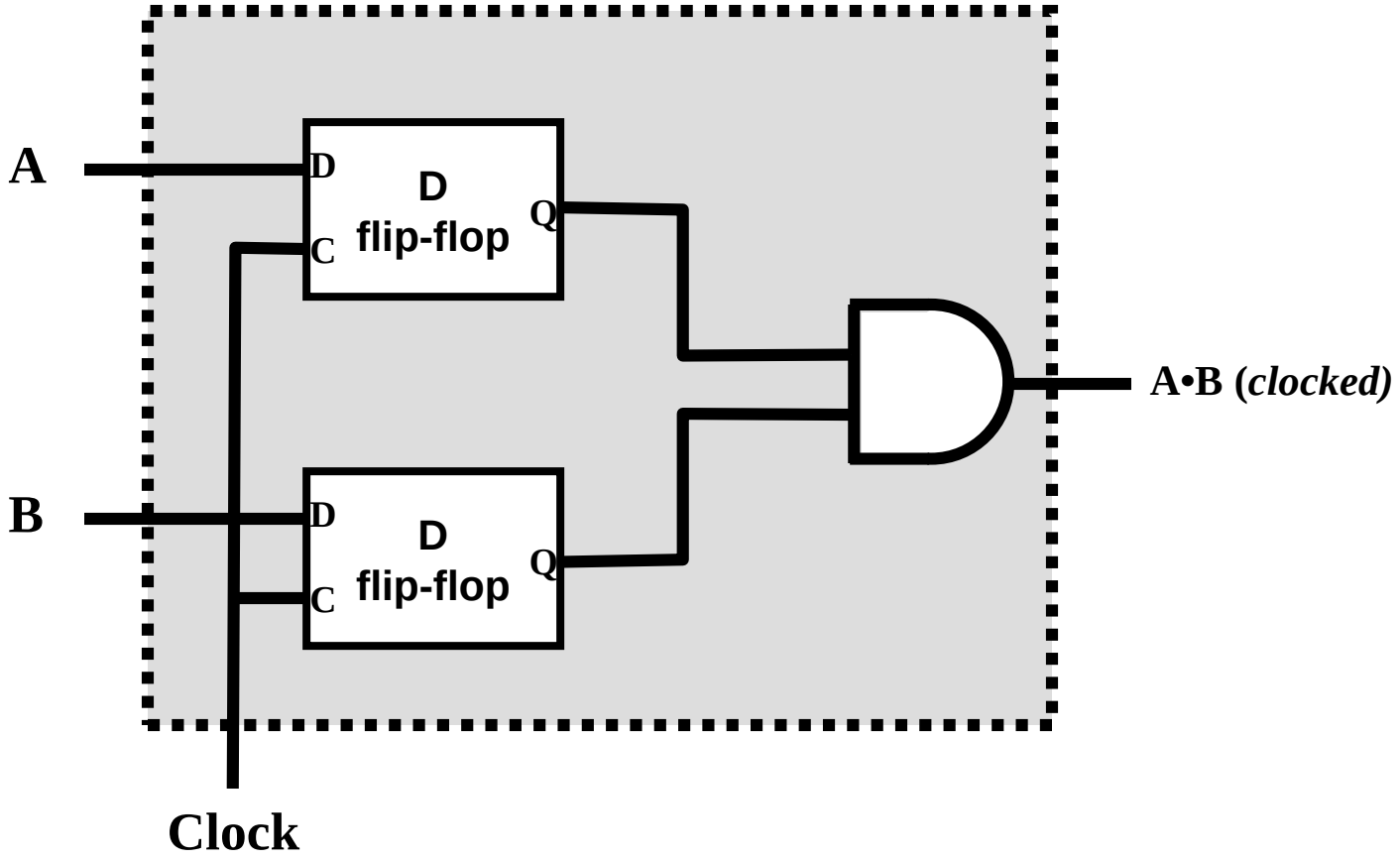
- The *control* is a collection of signals that enable and disable the inputs and outputs of the various components
- You can think of the control as the brain, whereas the datapath is the body
  - The datapath only does what the control instructs it to do

D Flip-Flop



The output (Q) changes to reflect D only when the Clock is asserted (i.e., on the clock transition from 0 to 1)

Clocked AND gate



Edge-triggered Clocking

- On their own, combinational elements do not use or need a clock
  - that's why we added the D flip-flops to our AND gate on the previous slide to create a clocked AND gate
- In general, values to be saved are updated only on a clock edge
  - When the clock switches from 0 to 1, *state elements* accept input signals

State Elements

- Any component that *stores* one or more values is called a *state element*
  - The entire processor can be viewed as a circuit that moves from one state (collection of all individual state elements) to another state
  - At time *t*, a component uses values generated at time *t-1*

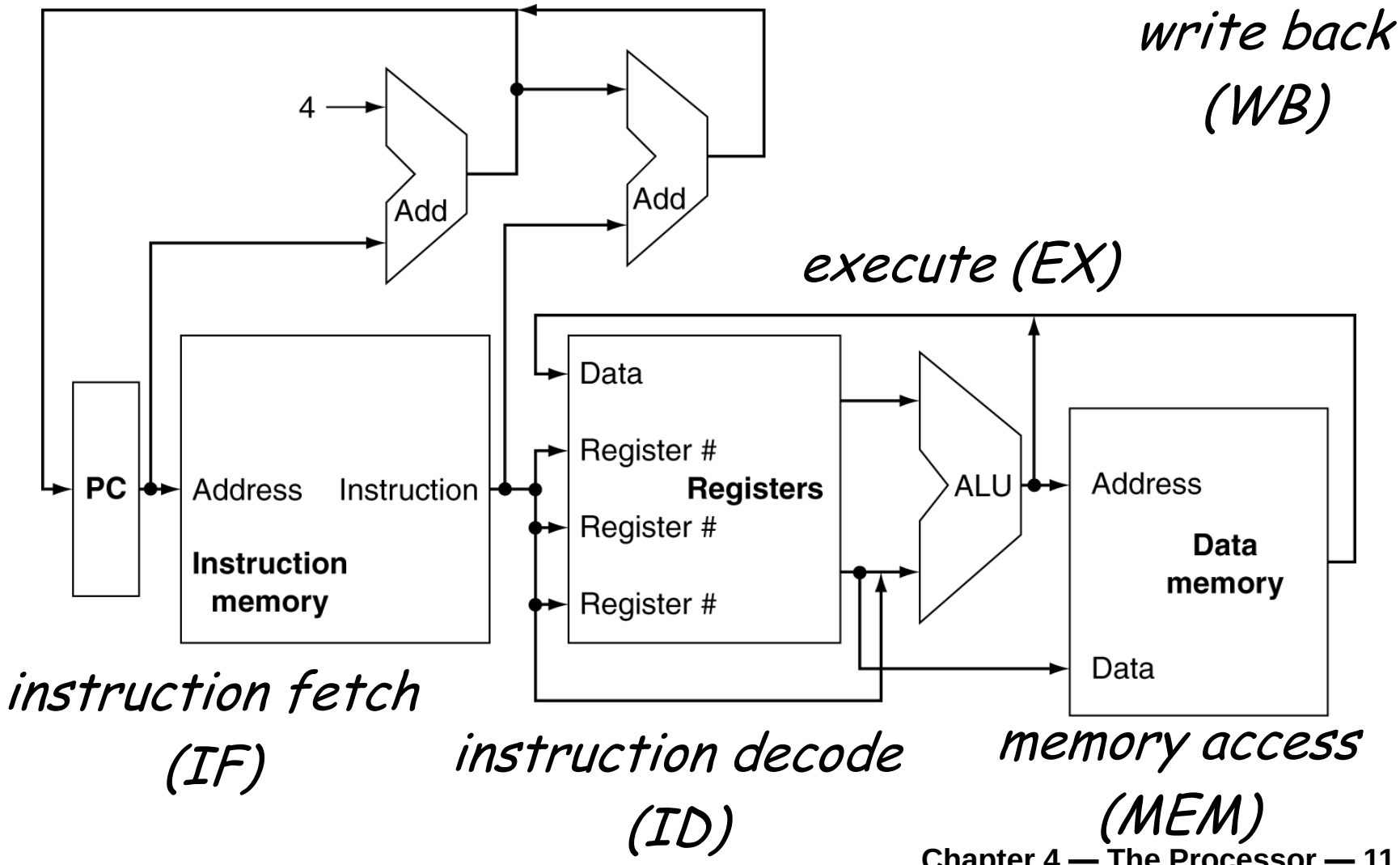
Instruction Memory vs Data Memory

- In processor design, it is useful to treat the memory that holds instructions as a separate *instruction memory* component
  - Instruction memory is *temporary*
  - Instruction memory is *read-only*
- Typically there is really one such memory that holds instructions and data
  - As we will see when we talk more about memory, the processor often has two interfaces to memory, one for instructions and one for data!

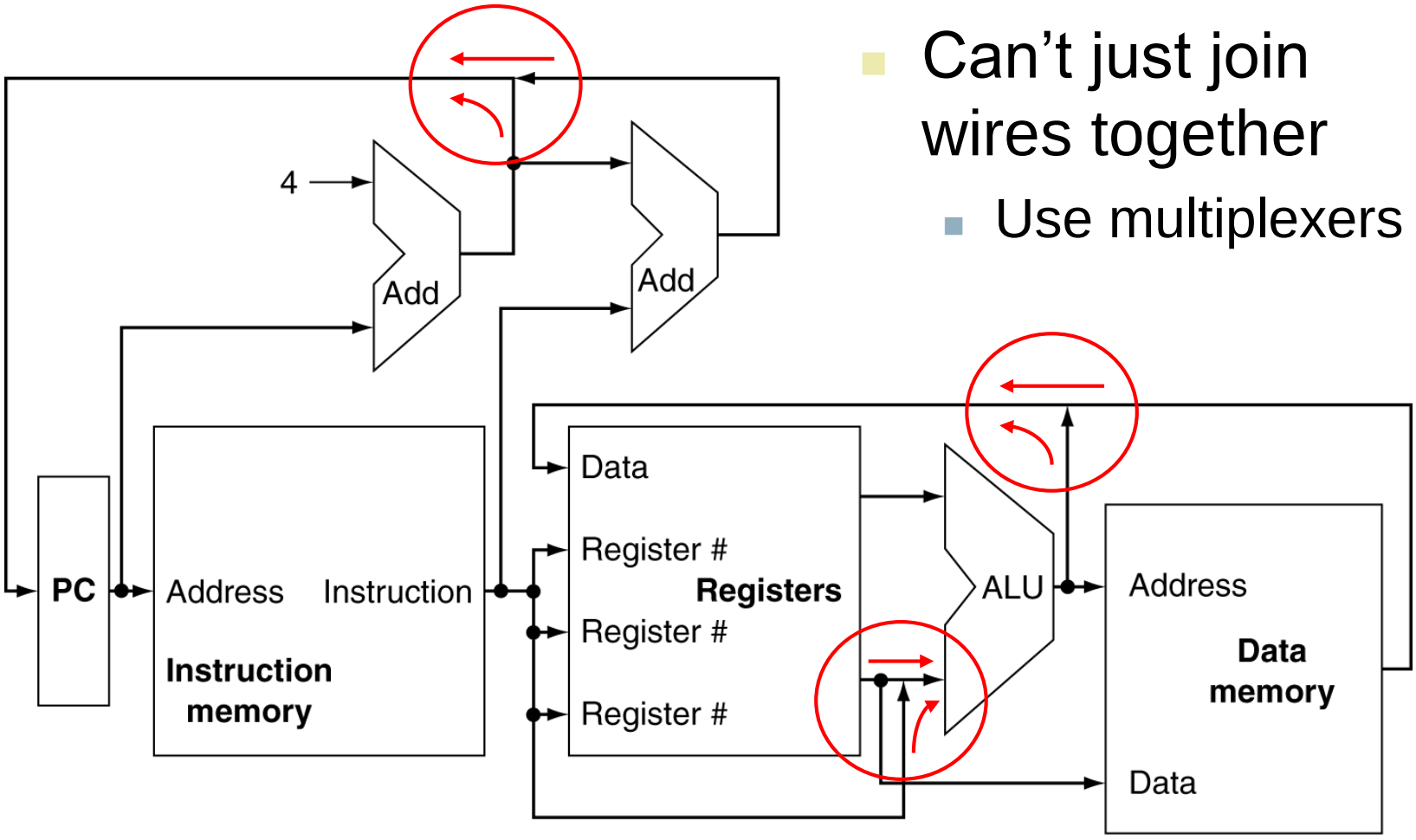
Functions for MIPS Instructions

- We can generalize the functions we need to realize all MIPS instructions as follows:
  - Using the program counter (PC) register as the address, read the next instruction from memory
  - Read one or two register values (depending on the specific instruction)
  - Perform an ALU operation, then maybe perform a memory read or write
  - Possibly change the value of a register

CPU Overview

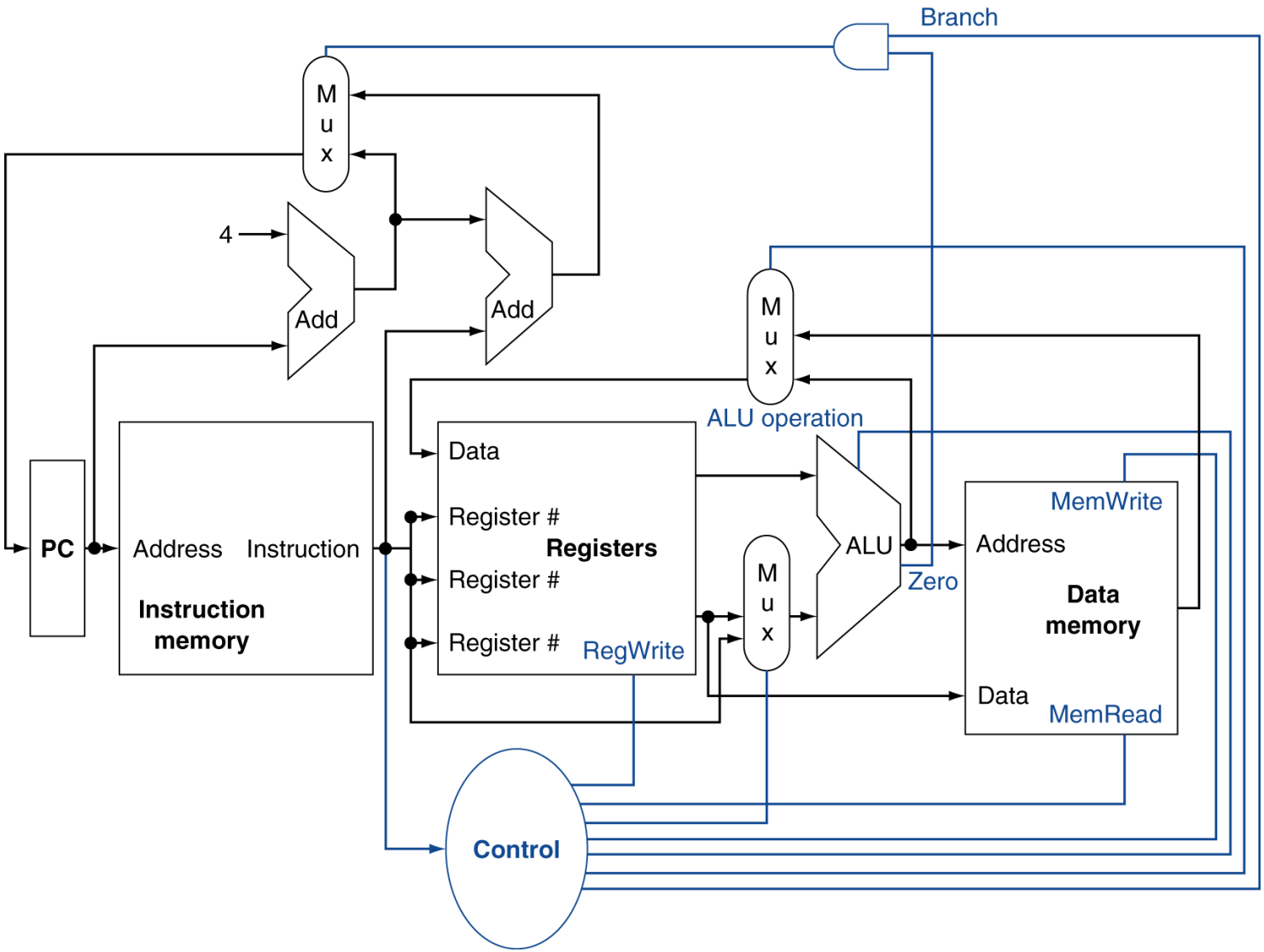


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

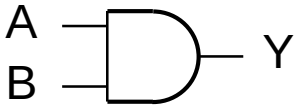

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

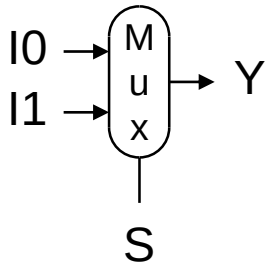
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

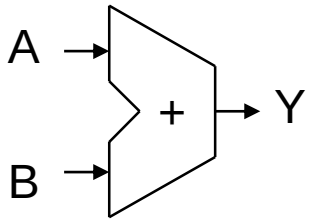
- AND-gate
  - Y = A & B



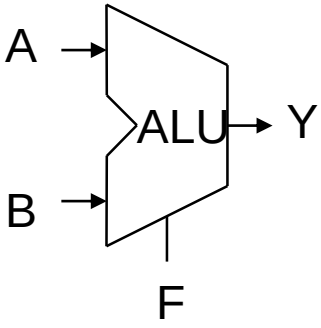
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

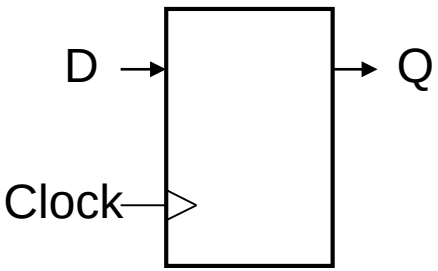


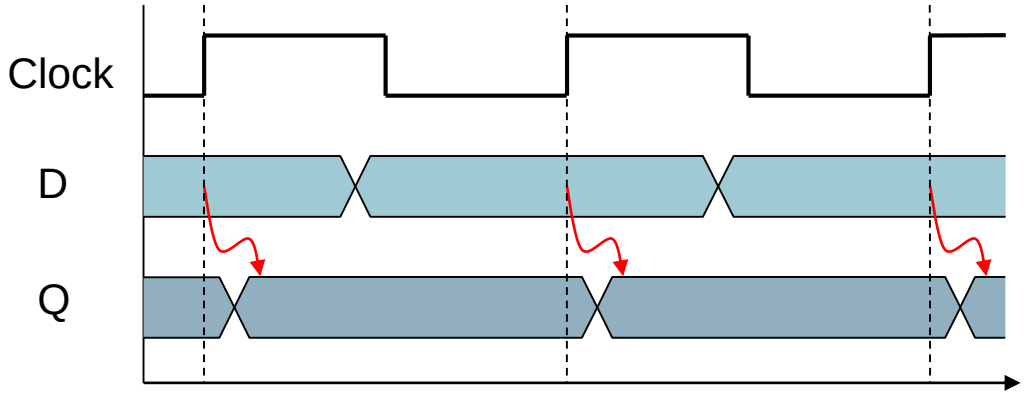
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

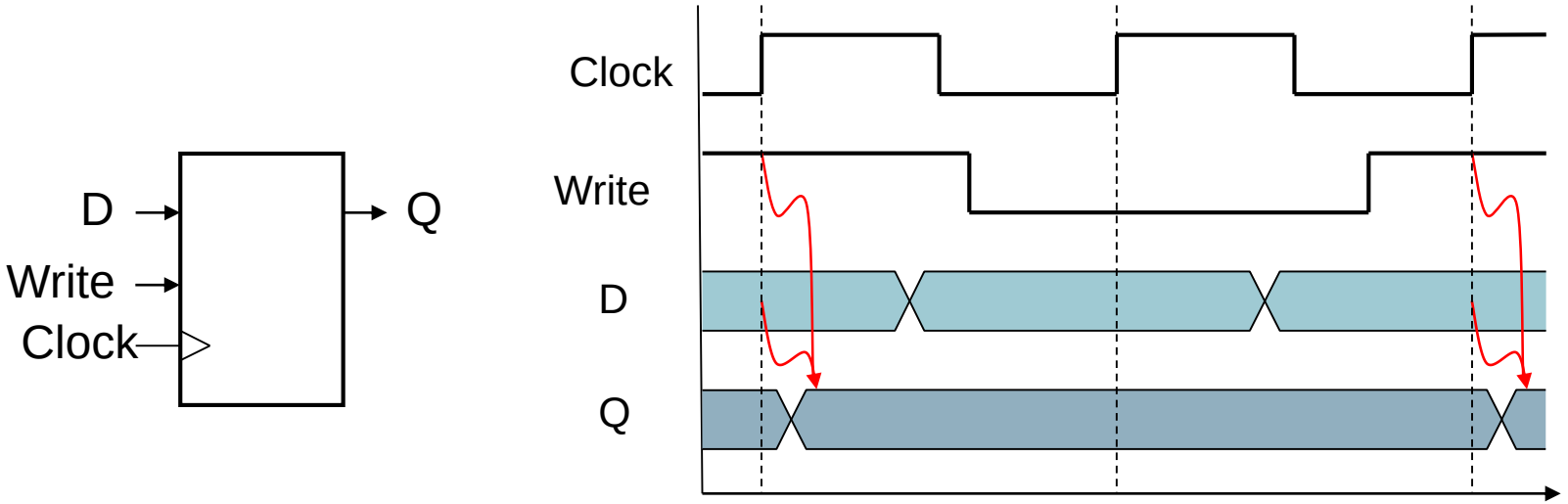
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clock changes from 0 to 1





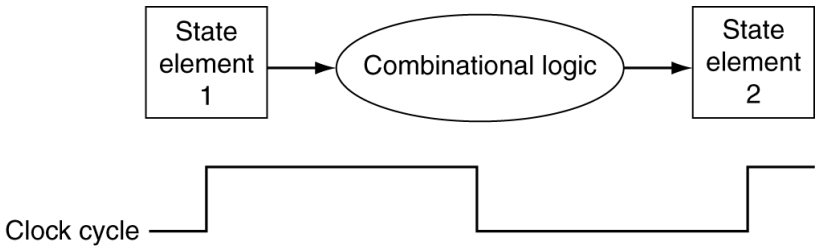
Sequential Elements

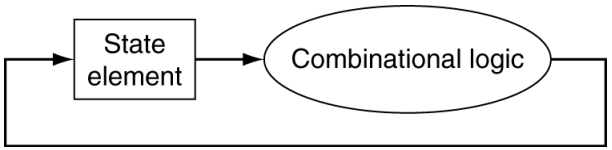
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



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

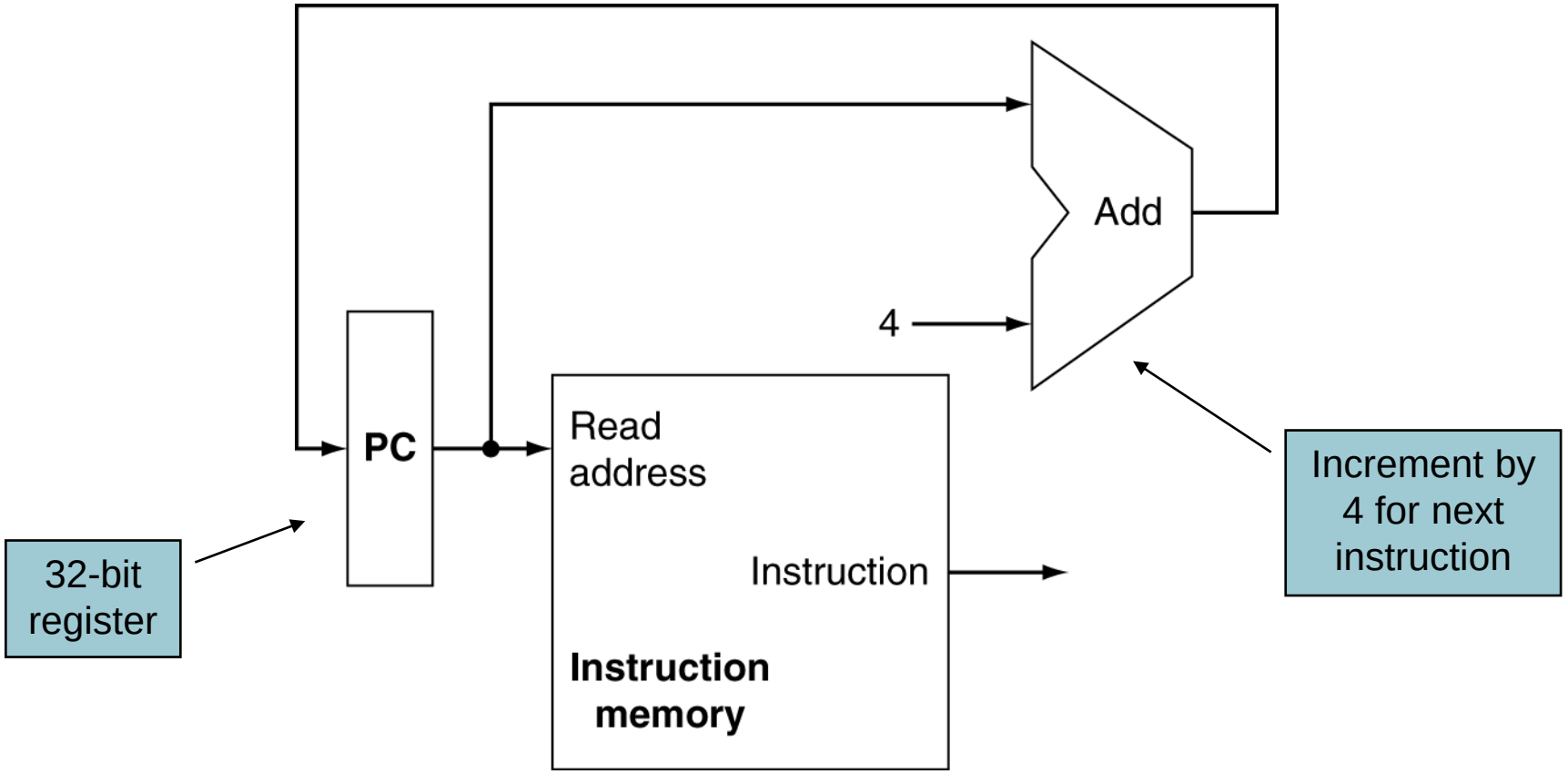




Building a Datapath

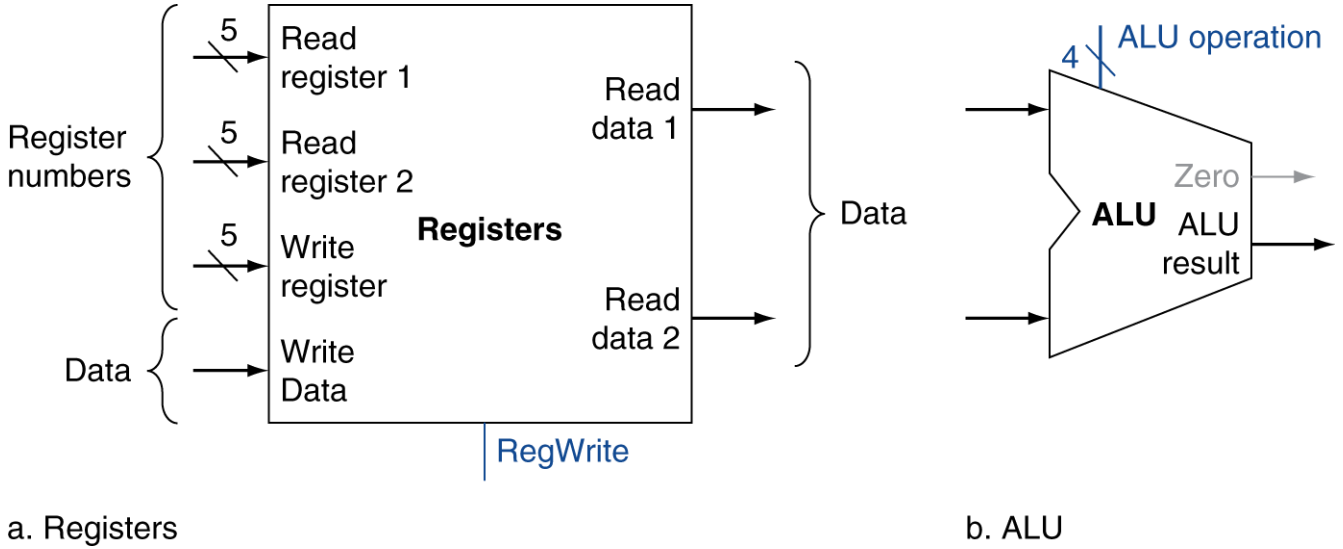
- Datapath
  - Elements that process data and addresses within the CPU
    - Registers, ALUs, multiplexers, memories, etc.
- We will build a MIPS datapath incrementally, refining the overview design as we learn more

Instruction Fetch (IF)



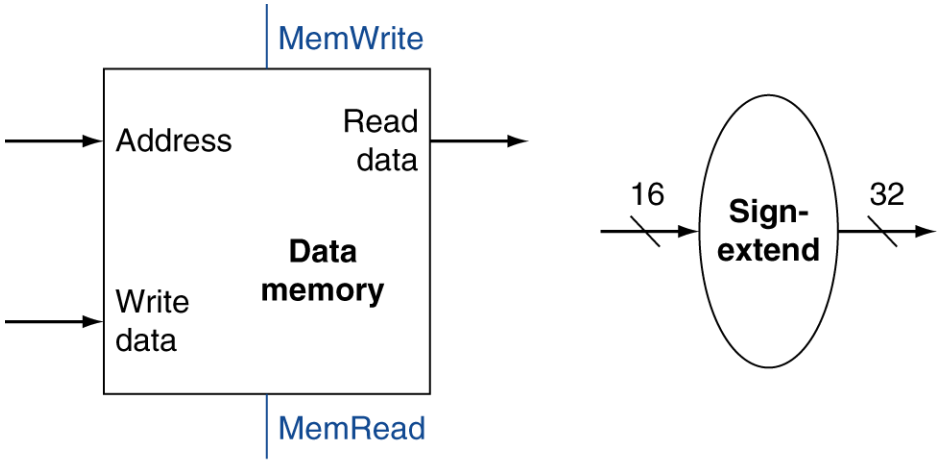
R-Format Instructions

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



Load/Store Instructions

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



a. Data memory unit

b. Sign extension unit

Pop Quiz

- lw $s1, -32($s3) means:

- A: $s1 ← M[$s3 - 32*4]
- B: $s1 ← M[$s3 - 32]
- C: $s1 ← M[$s3*4 - 32]
- D: $s1 ← M[$s3] - 32

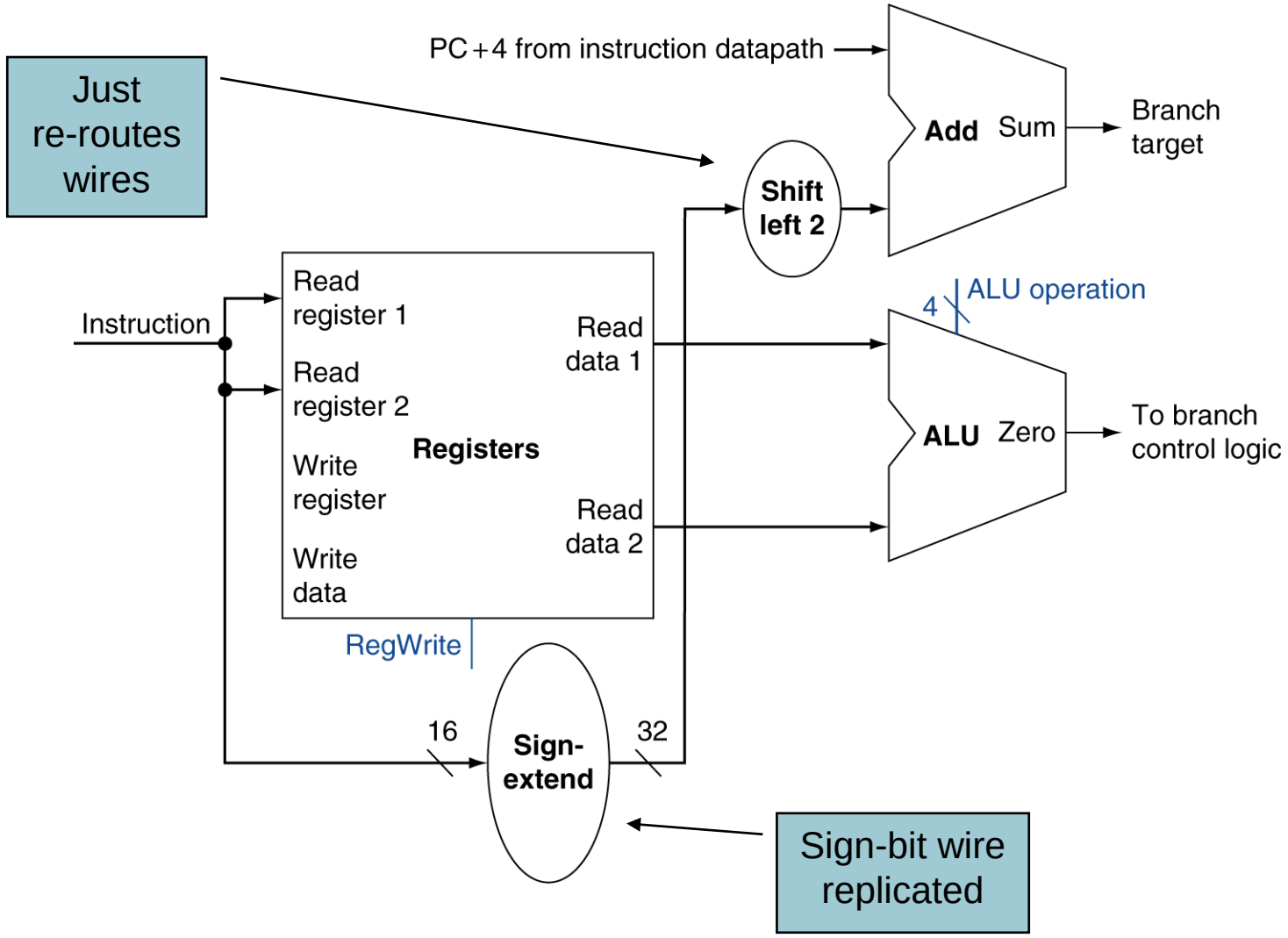
Computing the address

- We're given the 16-bit signed offset as part of the instruction
- We have a 32-bit ALU
  - need to sign-extend the offset to 32 bits
- Feed the 32-bit offset and the contents of a register to the ALU
- Tell the ALU to "add"

Branch Instructions

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

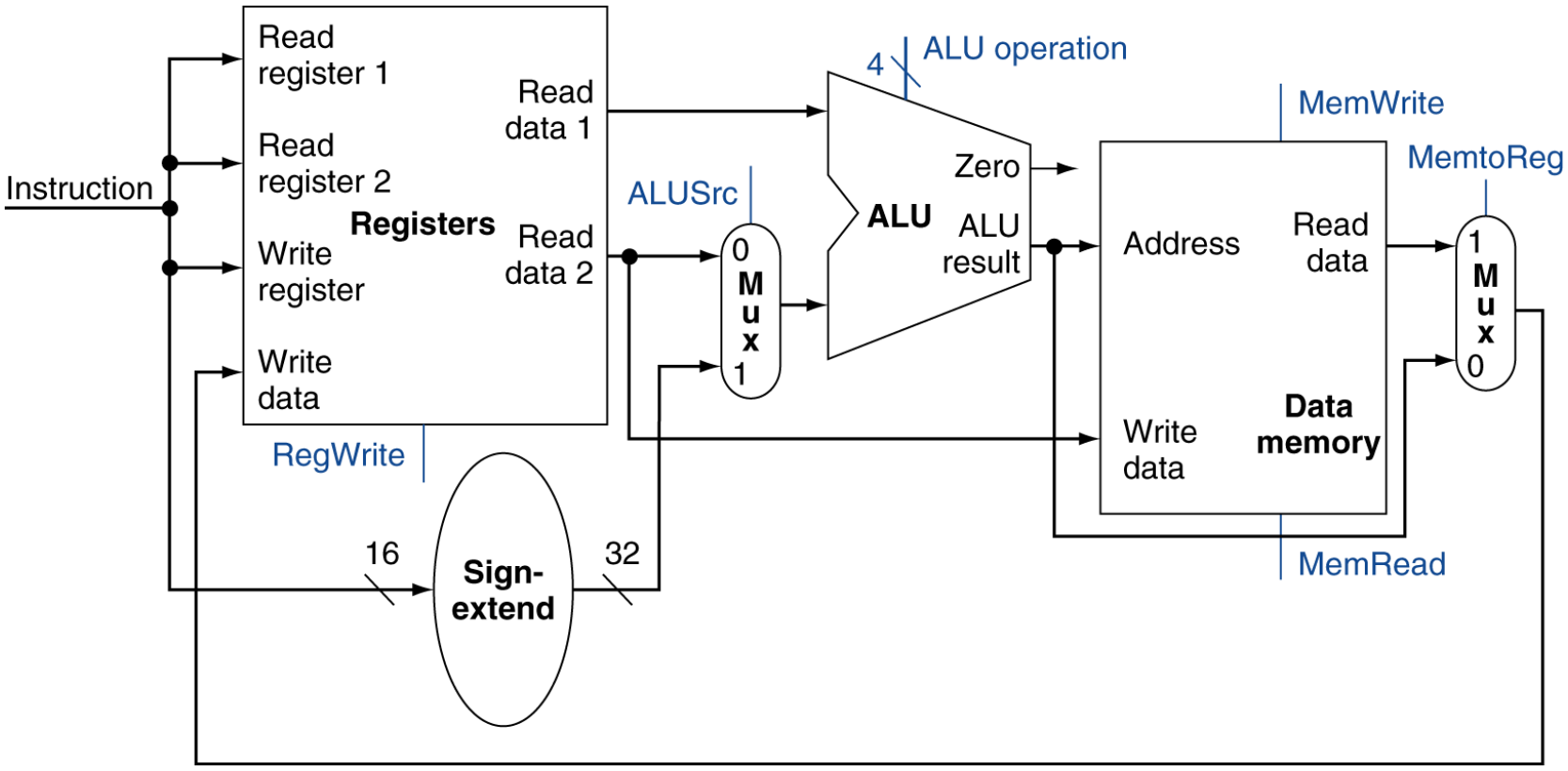
Branch Instructions



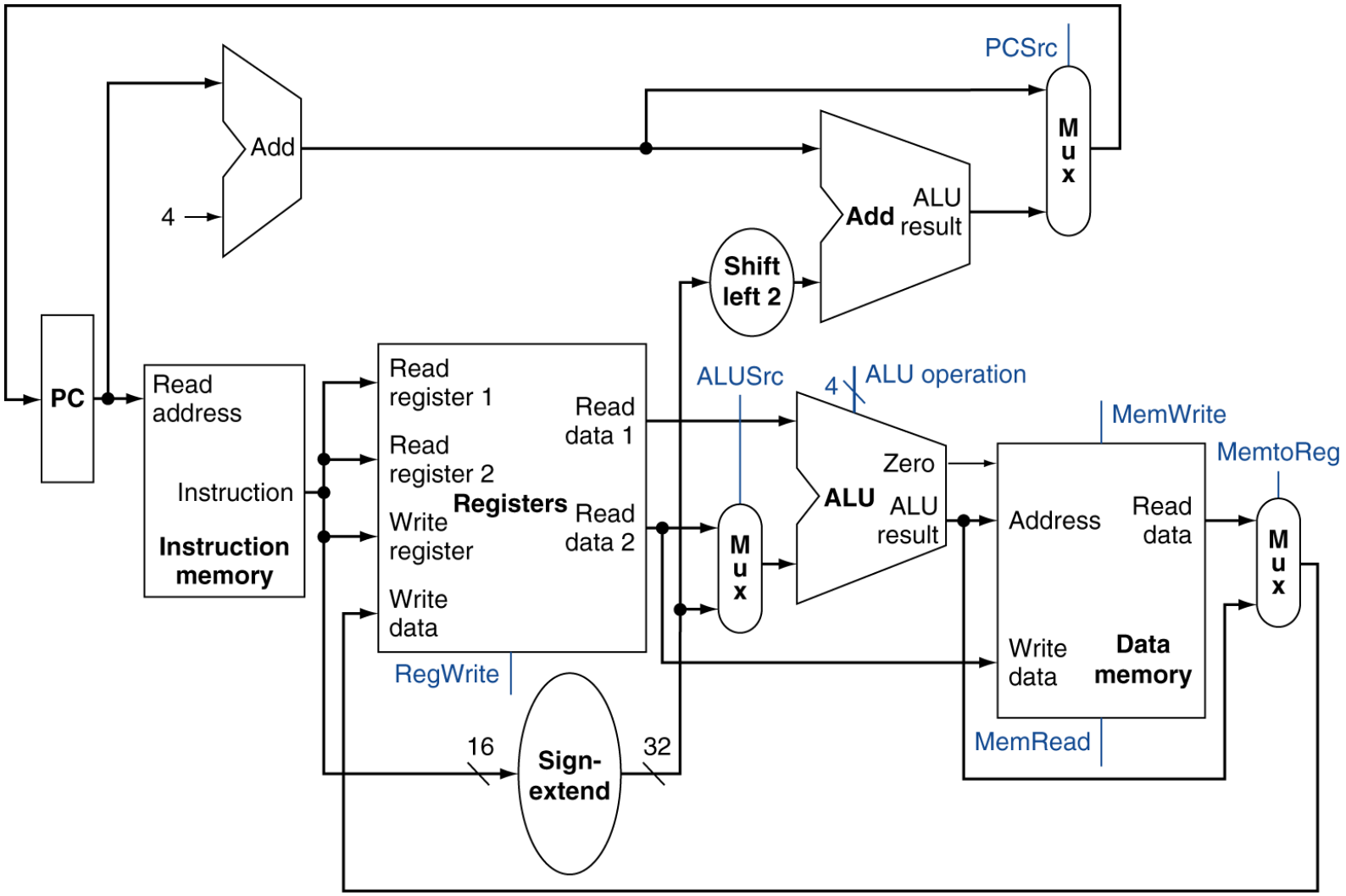
Composing the Elements

- 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



Control Unit

- We need something that can generate the *controls* in the datapath
- Depending on what kind of instruction we are executing, different controls should be turned on (*asserted)* and off (*deasserted)*
- We need to treat each control individually (as a separate Boolean function)

Controls

- Our datapath includes a bunch of *controls*:
  - ***ALU operation***
  - ***RegWrite***
  - ***ALUSrc***
  - ***MemWrite***
  - ***MemtoReg***
  - ***MemRead***
  - ***PCSrc***

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

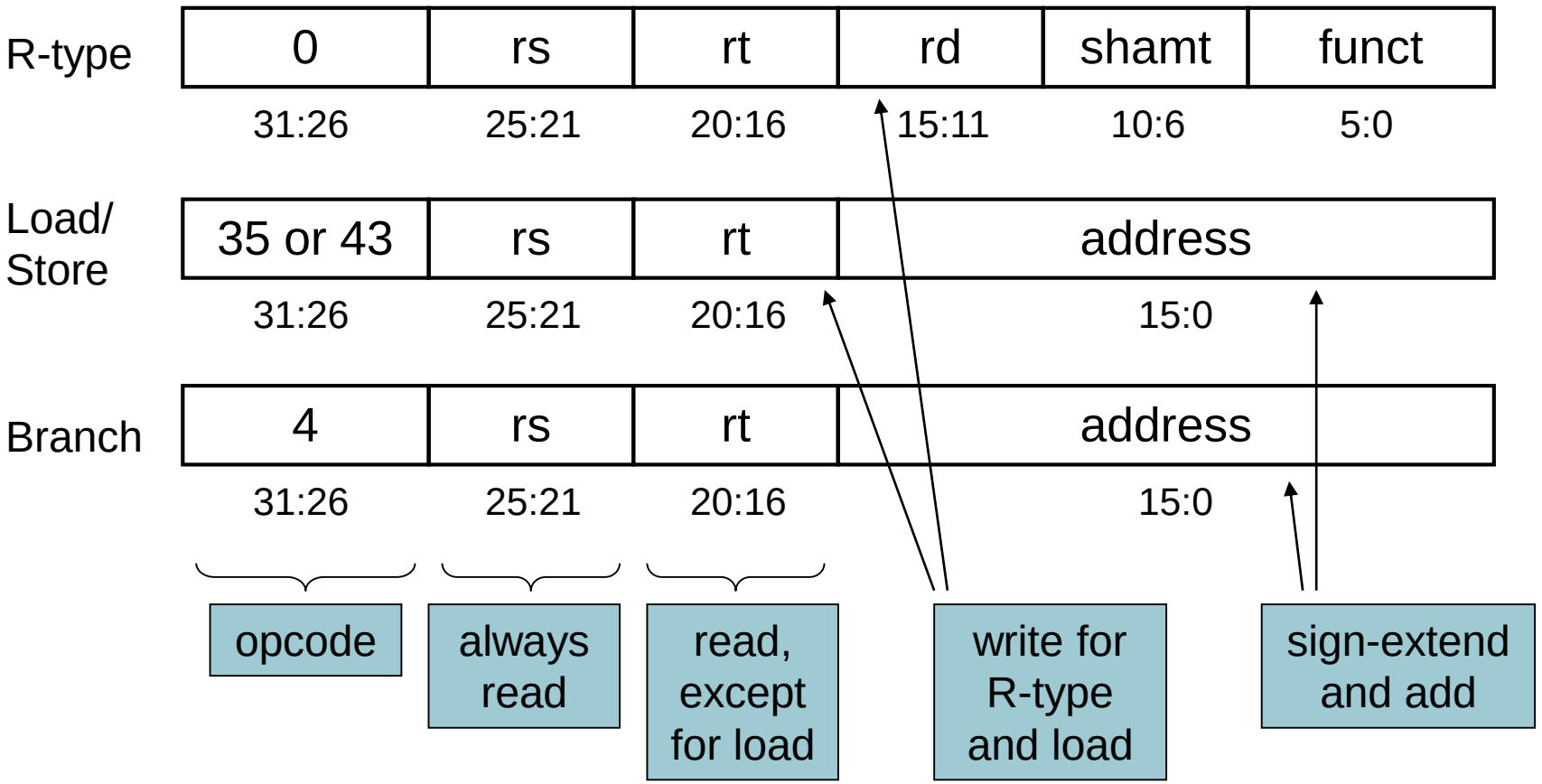
ALU Control

- 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

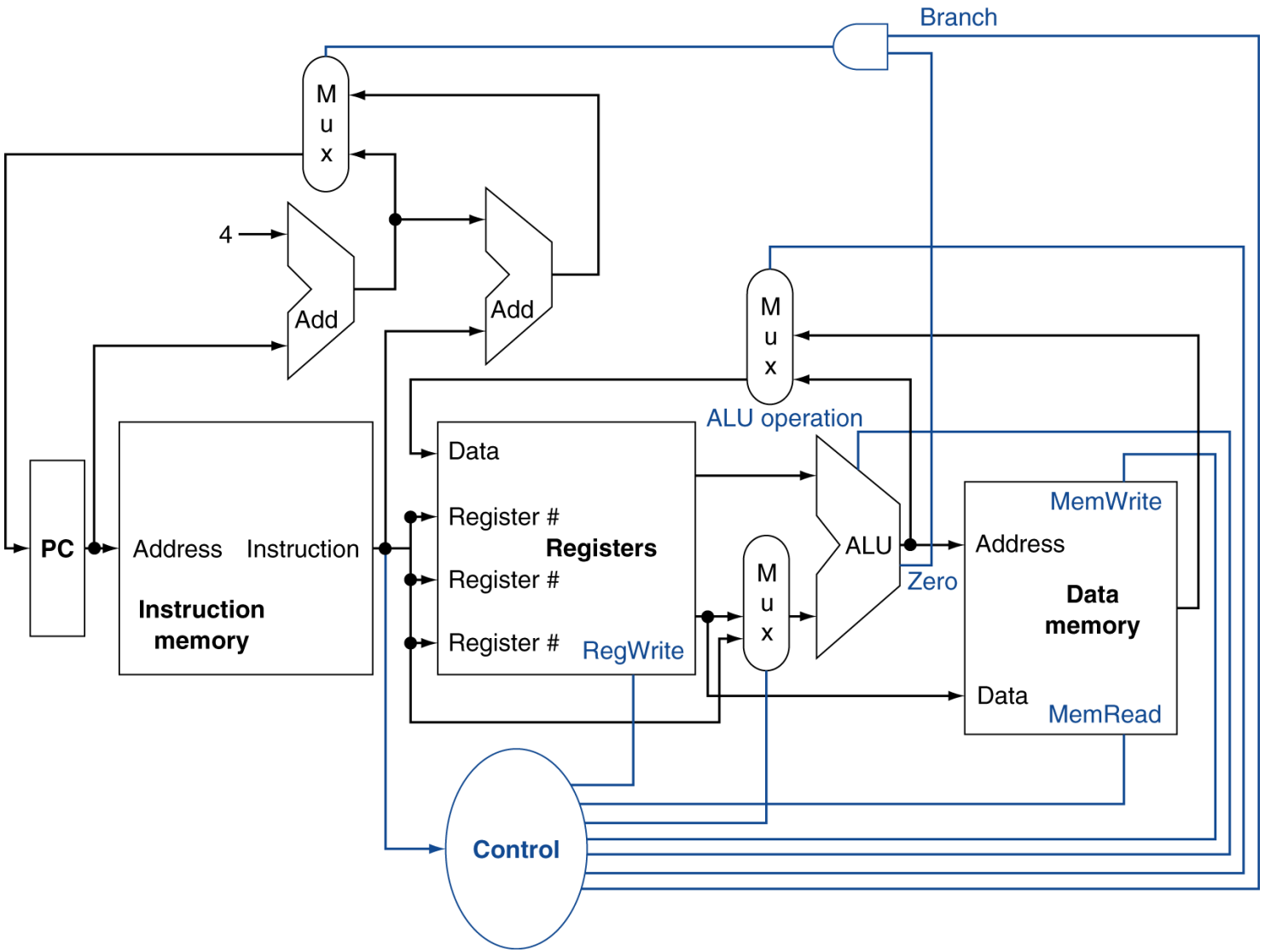
- Control signals derived from instruction



Adding the ALU Control

- We can now add the ALU control to the datapath:
  - inputs to this control come from the instruction and from ALUOp
- If we try to show all the details, the picture becomes too complex:
  - just drop in an "ALU Control" box

Control

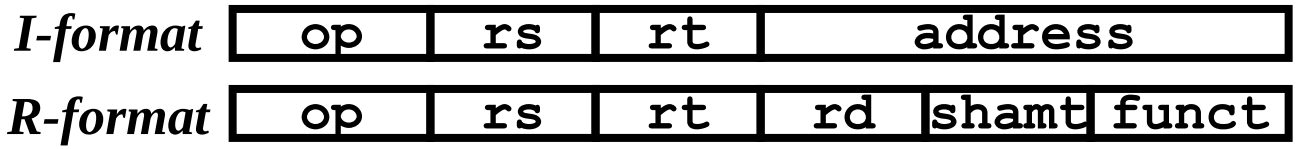

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Implementing Other Controls

- The other controls in out datapath must also be specified as functions
- We need to determine the inputs to all the functions
  - primarily the inputs are part of the instructions, but there are exceptions
- Need to define precisely what conditions should turn on each control

RegDst Control Line

- Controls a multiplexor that selects one of the fields `rt` or `rd` from an R-format or I-format instruction
- I-Format is used for load and store
- **`sw`** needs to *write* to the register `rt`



RegDst usage

- **RegDst** should be
  - 0 to send **rt** to the write register # input
  - 1 to send **rd** to the write register # input

- **RegDst** is a function of the opcode field:
  - If instruction is **lw**, **RegDst** should be 0
  - For all other instructions, **RegDst** is 1

`RegWrite` Control

- Using a 1 tells the register file to write a register
  - whatever register is specified by the write register # input is written with the data on the write register data input
- Should be a `1` for arithmetic/logical instructions and for a load
- Should be a `0` for store or **`beq`**

ALUSrc Control

- MUX that selects the source for the second ALU operand
  - 1 means select the second register file output (read data 2)
  - 0 means select the sign-extended 16 bit offset (part of the instruction)

- Should be a 1 for load and store
- Should be a 0 for everything else

MemRead Control

- A 1 tells the memory to put the contents of the memory location (specified by the address lines) on the Read data output

- Should be 1 for load
- Should be 0 for everything else

MemWrite Control

- 1 means that memory location (specified by memory address lines) should get the value specified on the memory Write Data input

- Should be 1 for store
- Should be 0 for everything else

MemToReg Control

- MUX that selects the value to be stored in a register (that goes to the register write data input)
  - 1 means selected value coming from memory data output
  - 0 means select value coming from ALU output
- Should be 1 for load
- Should be 0 for arithmetic/logical operations

PCSrc Control

- MUX that select the source for the value written to PC register
  - 1 means select the output of the Adder used to compute relative branch address
  - 0 means select output of PC+4 adder
- Should be 1 for beq if registers are equal
- Should be 0 for other instructions or registers are different

PCSrc depends on result of ALU operation!

- This control line can't be simply a function of the instruction (all the others can)

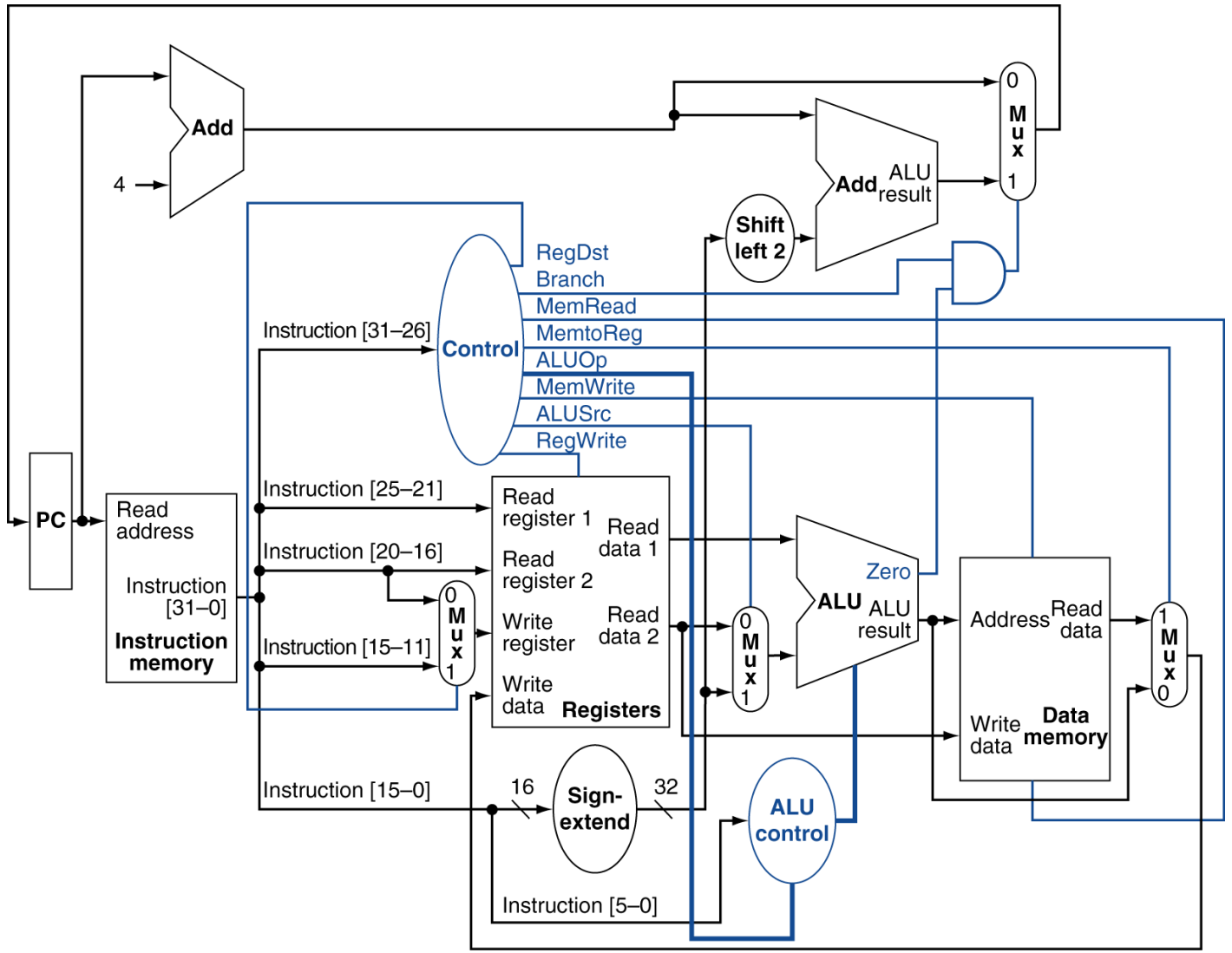
- **PCSrc** should be a 1 only when:
  - **beq** AND **ALU zero** output is a 1
- We will generate a signal called "branch" that we can AND with the ALU zero output

Control Lines Summary

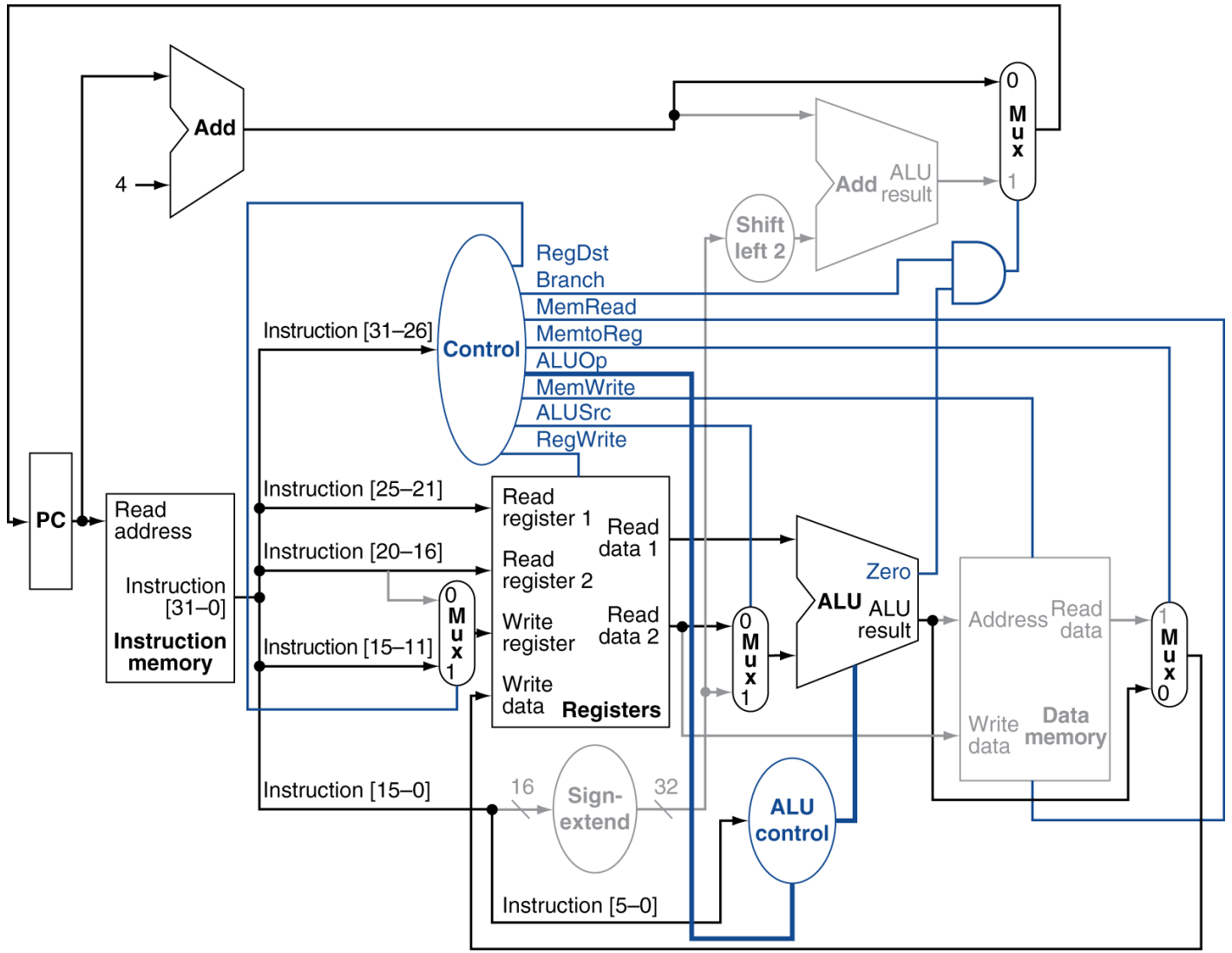
- Control all possible decisions by turning bits on/off
- Essentially, setting "flags" to indicate desired behavior
- All encoded within the instructions themselves!

- See Figure 4.18 in P&H for details!

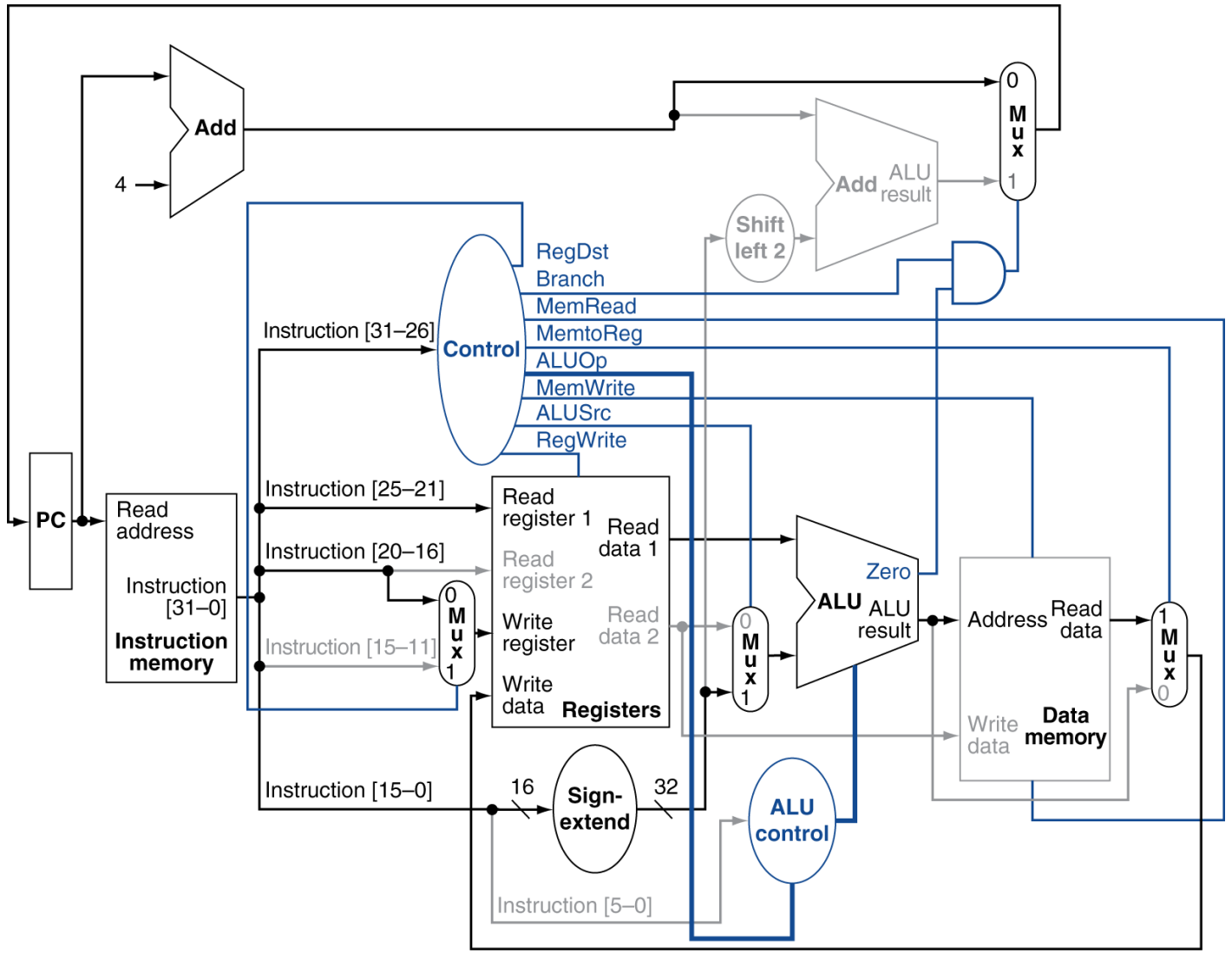
Datapath With Control



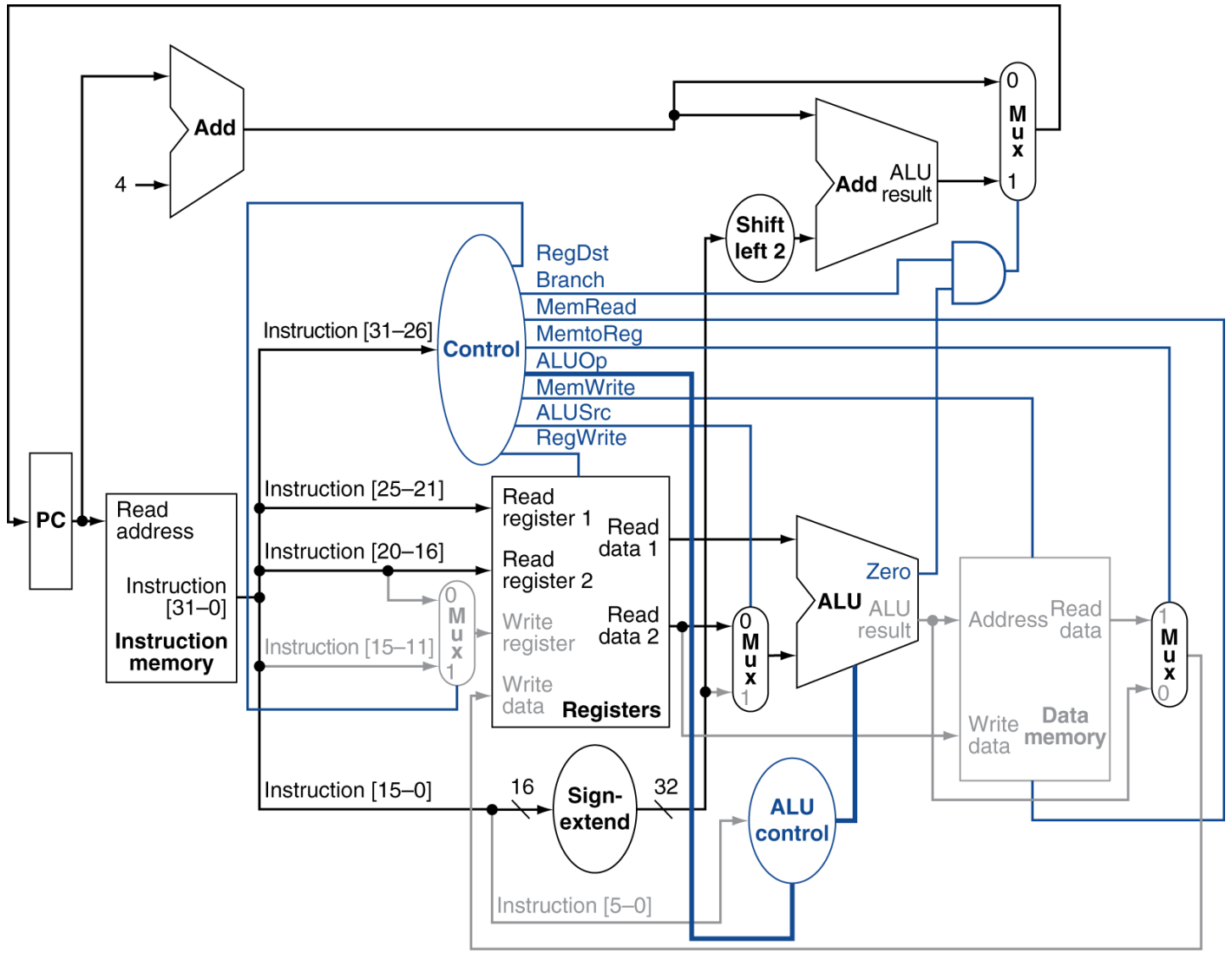
R-Type Instruction



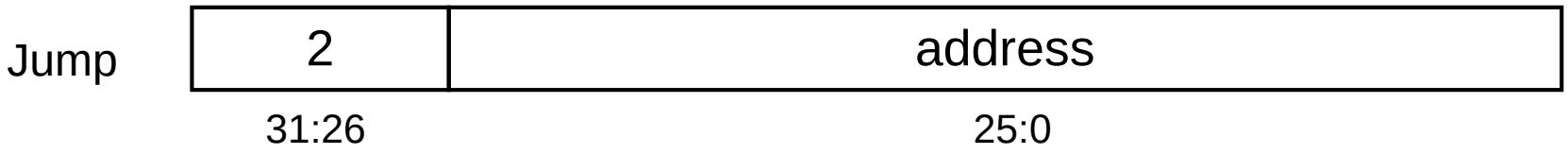
Load Instruction



Branch-on-Equal Instruction

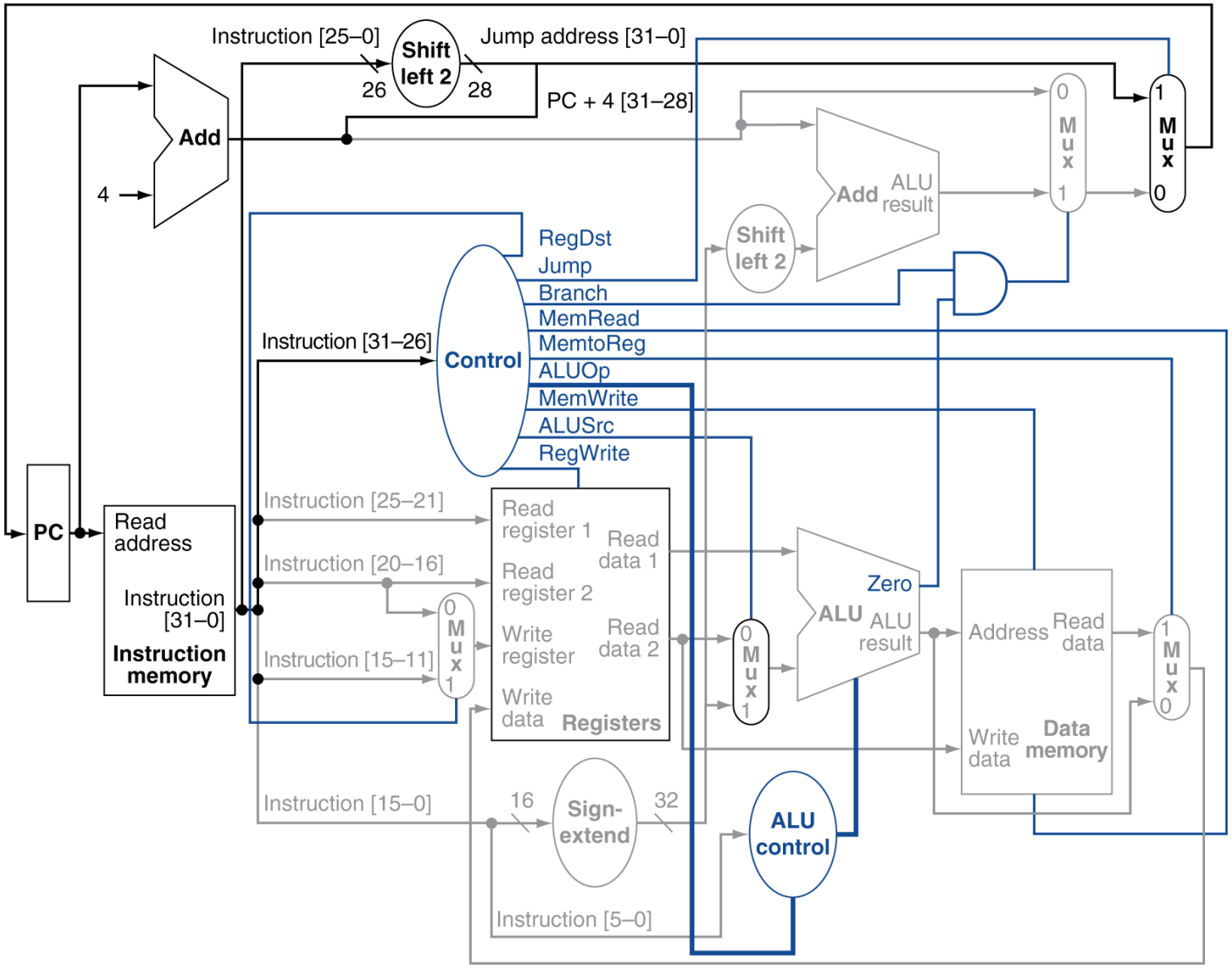


Implementing Jumps



- 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

- 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 implementing *pipelining*