CSEE 3827: Fundamentals of Computer Systems

Lecture 18, 19, & 20

April 2009

Martha Kim
martha@cs.columbia.edu

Outline

- We will examine two MIPS implementations
  - A single-cycle version
  - A pipelined version
- Simple subset of MIPS, showing most aspects
  - Memory reference: lw, sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j
- CPU performance factors
  - Instruction count (determined by ISA and compiler)
  - Cycles per instruction and cycle time (determined by CPU hardware)

Instruction Execution

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

CPU Overview

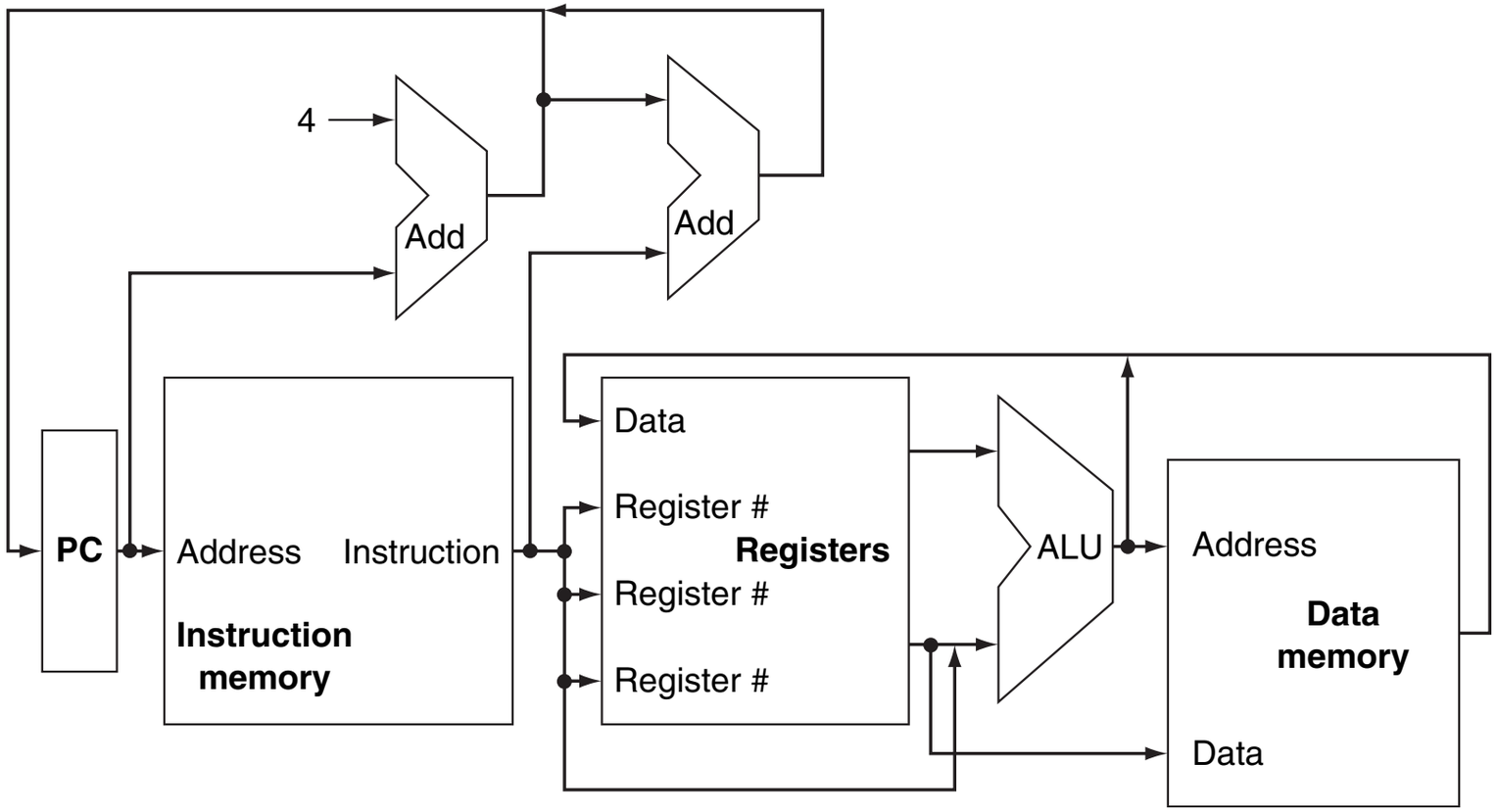


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

Can’t just join wires together, use muxes

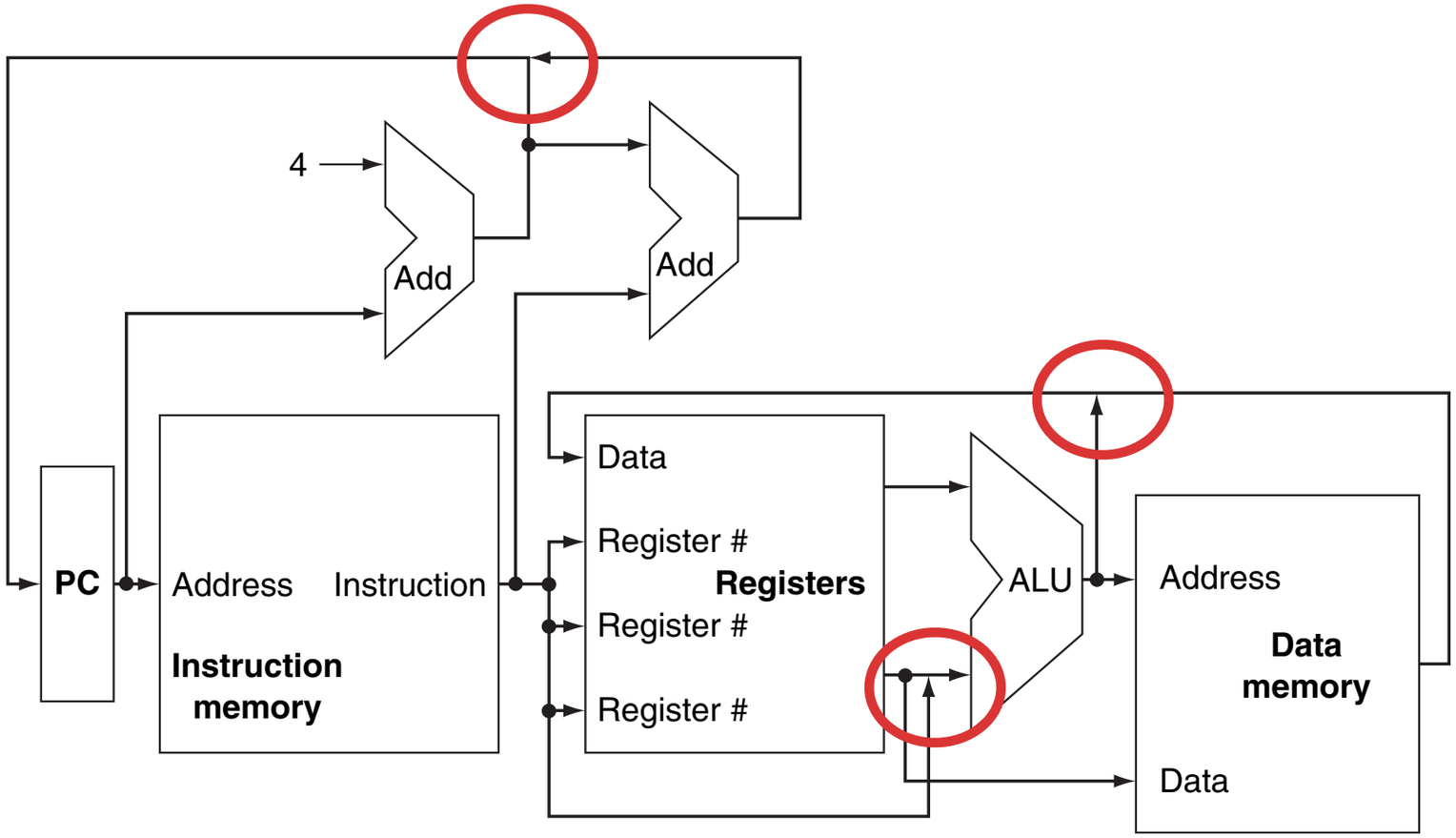


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

Control

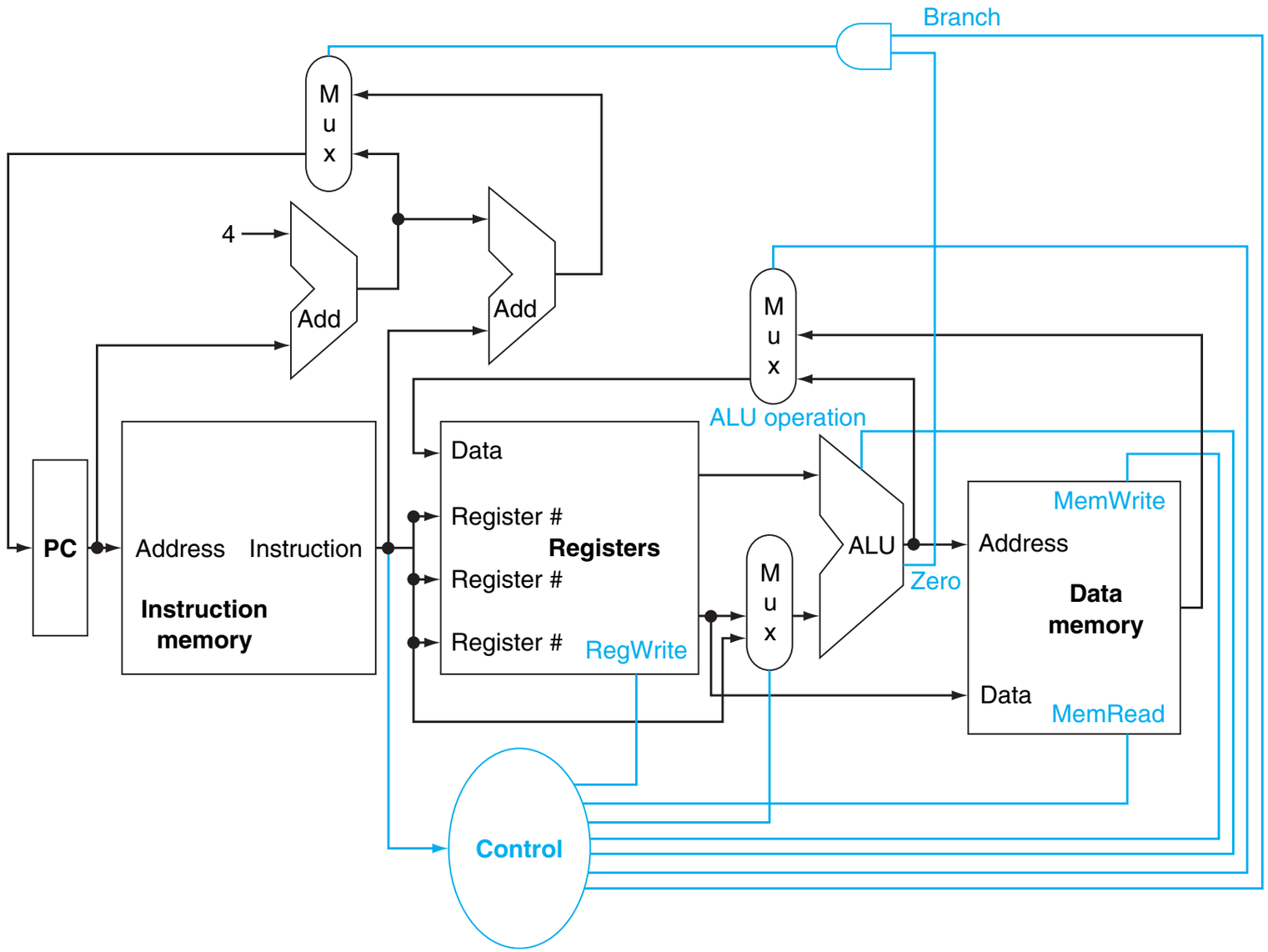
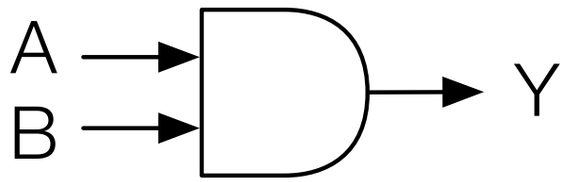


FIGURE 4.2 The basic implementation of the MIPS subset, including the necessary multiplexors and control lines. The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction OR a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

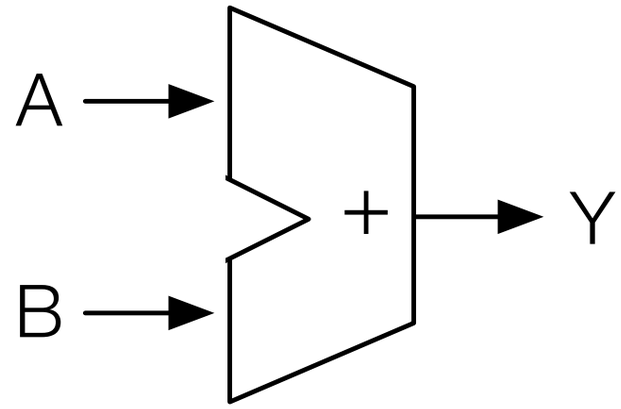
MIPS Datapath

Combinational Elements

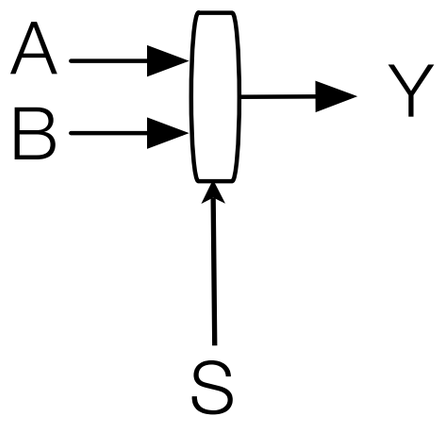
- AND gate (Y = A & B)



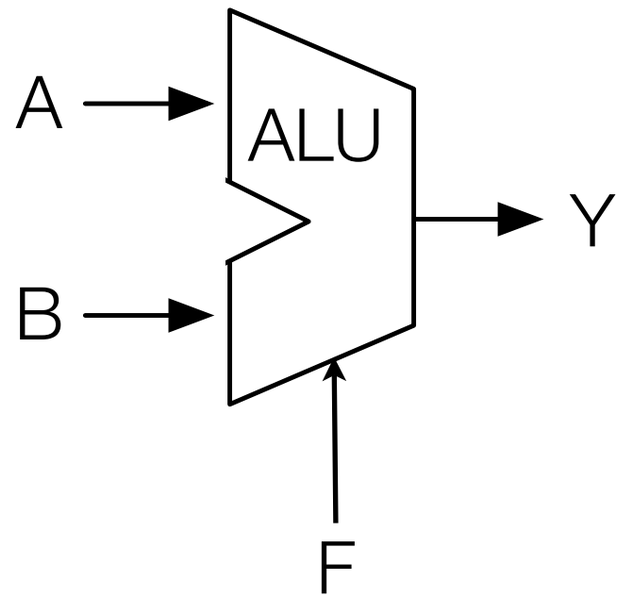
- Adder (Y = A + B)



- Multiplexer (Y = S ? A : B)



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



Clocking Methodology

Combinational logic transforms data during clock cycles.
Longest combinational delay determines clock period.

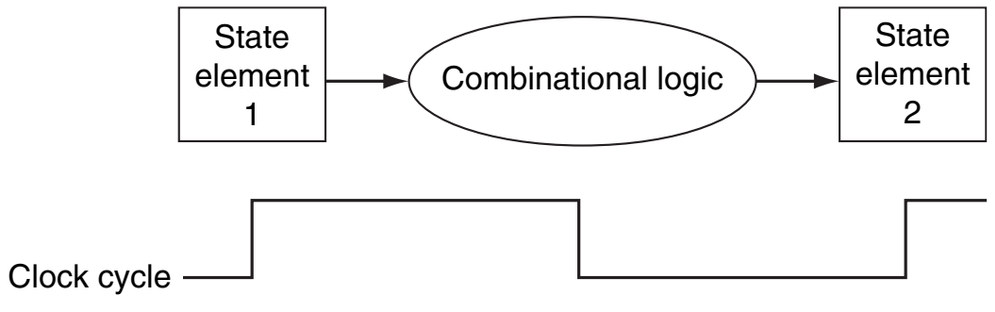


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed to be edge-triggered. Copyright © 2009 Elsevier, Inc. All rights reserved.

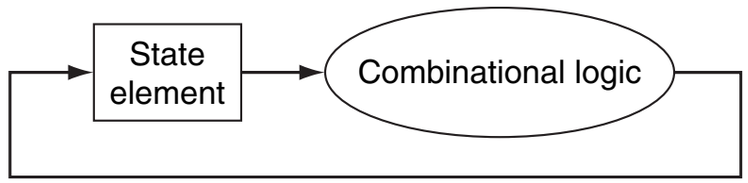


FIGURE 4.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure. Copyright © 2009 Elsevier, Inc. All rights reserved.

Building a datapath incrementally

- Datapath: elements that process data and addresses in the CPU
- Datapath will execute 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

Instruction Fetch

- Fetch Instruction contained in PC register from memory
- Compute PC + 4 for next instruction

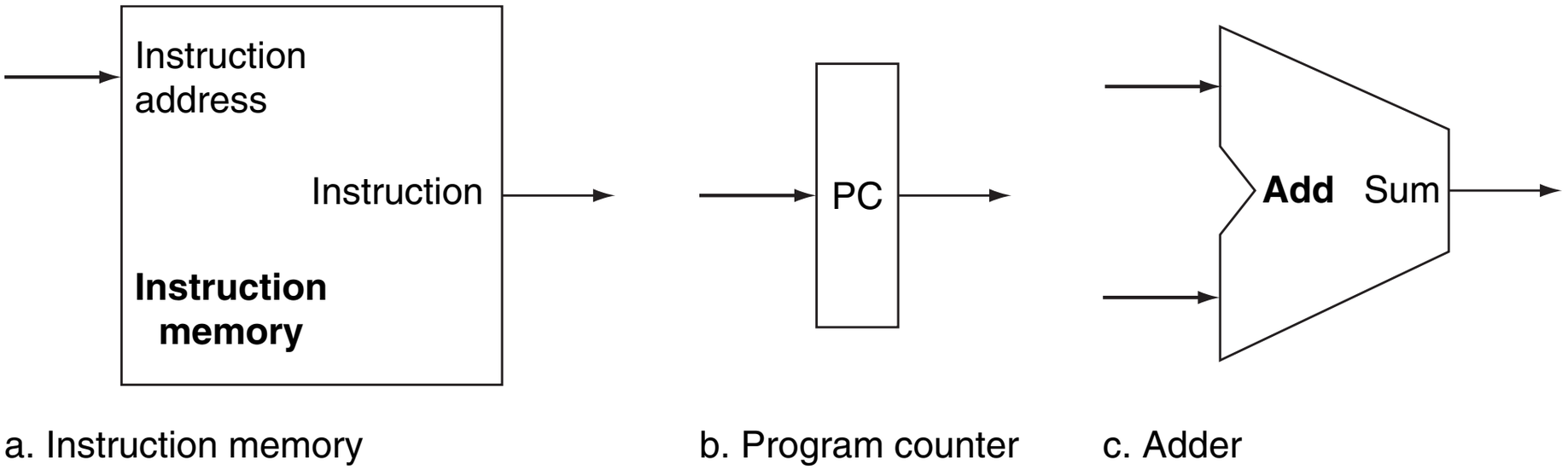


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

Part 1: Instruction Fetch

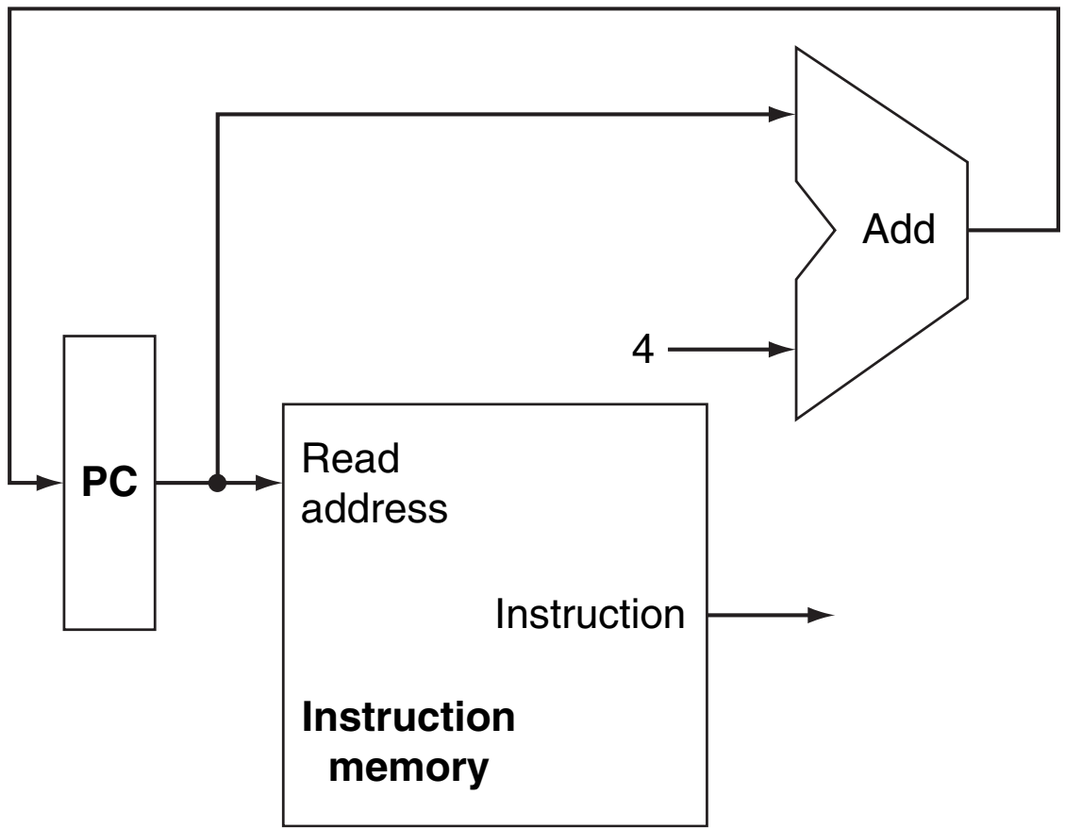


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

R-Format Instructions

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

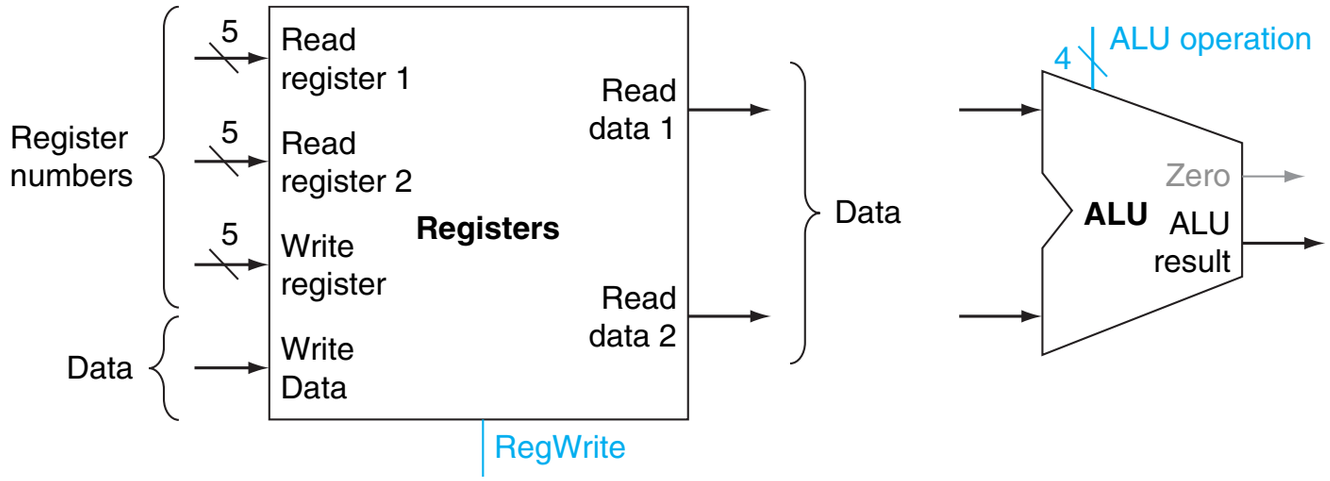
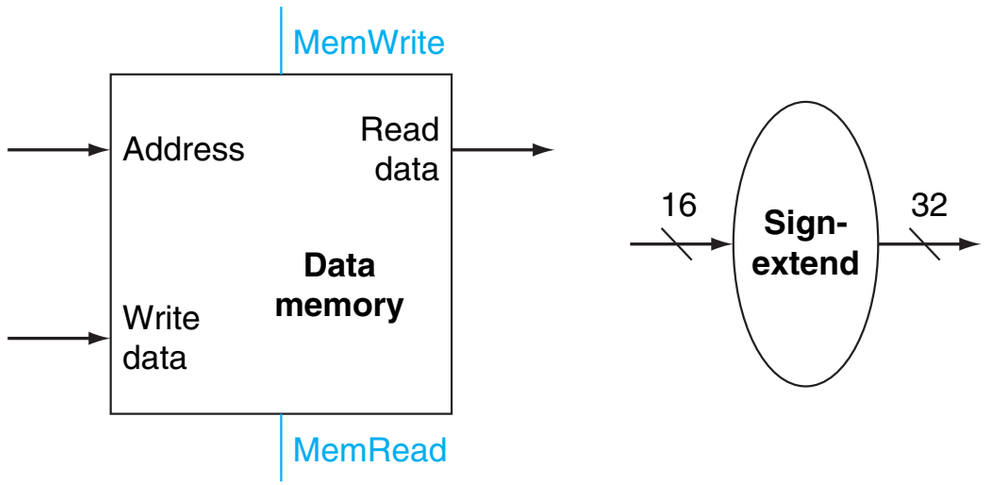


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section C.8 of Appendix C. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in Appendix C. We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 4.9, when we discuss exceptions; we omit it until then.

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

FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section C.8 of Appendix C for further discussion of how real memory chips work.

Part 2: R-Type/Load/Store Datapath

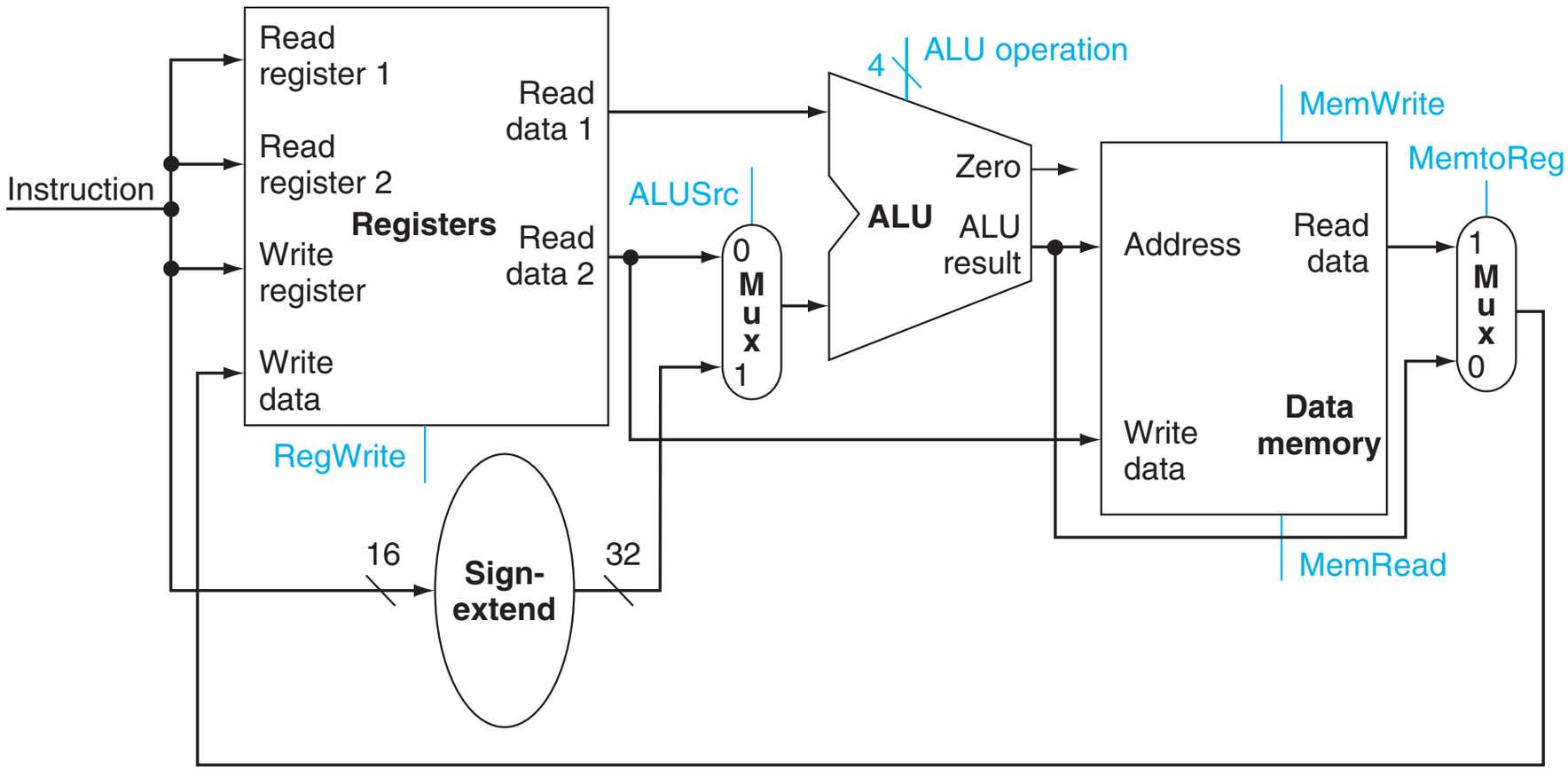


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example. Copyright © 2009 Elsevier, Inc. All rights reserved.

Branch Instructions

- Read register operands
- Compare operands (use ALU: subtract and check zero output)
- Calculate target address
  - Sign-extend displacement
  - Shift left two places (word displacement)
  - Add to PC+4 (already calculated by instruction fetch)

Part 3: Instruction Fetch w. Branch

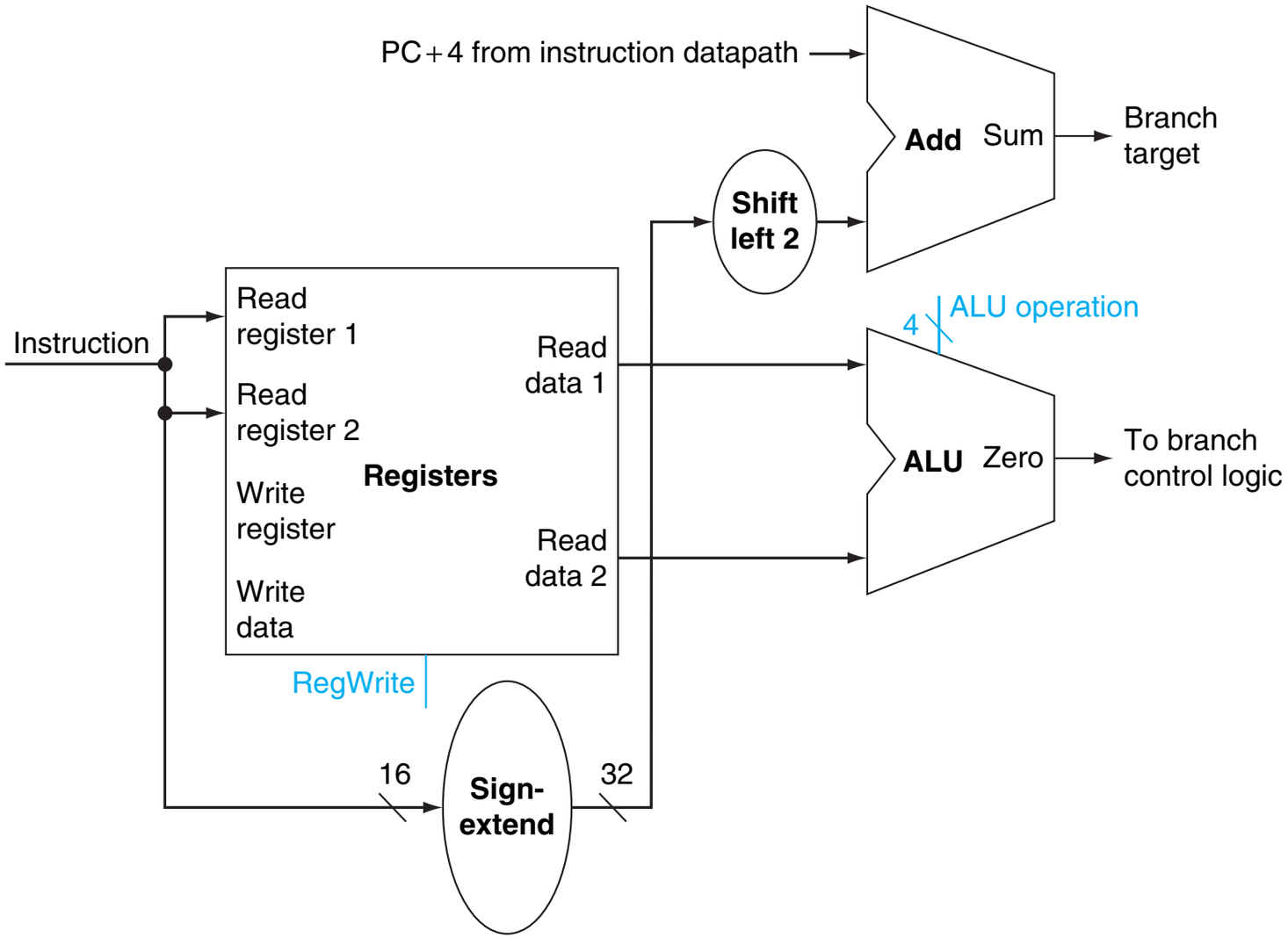


FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the "shift" is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only "sign bits." Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Full Datapath

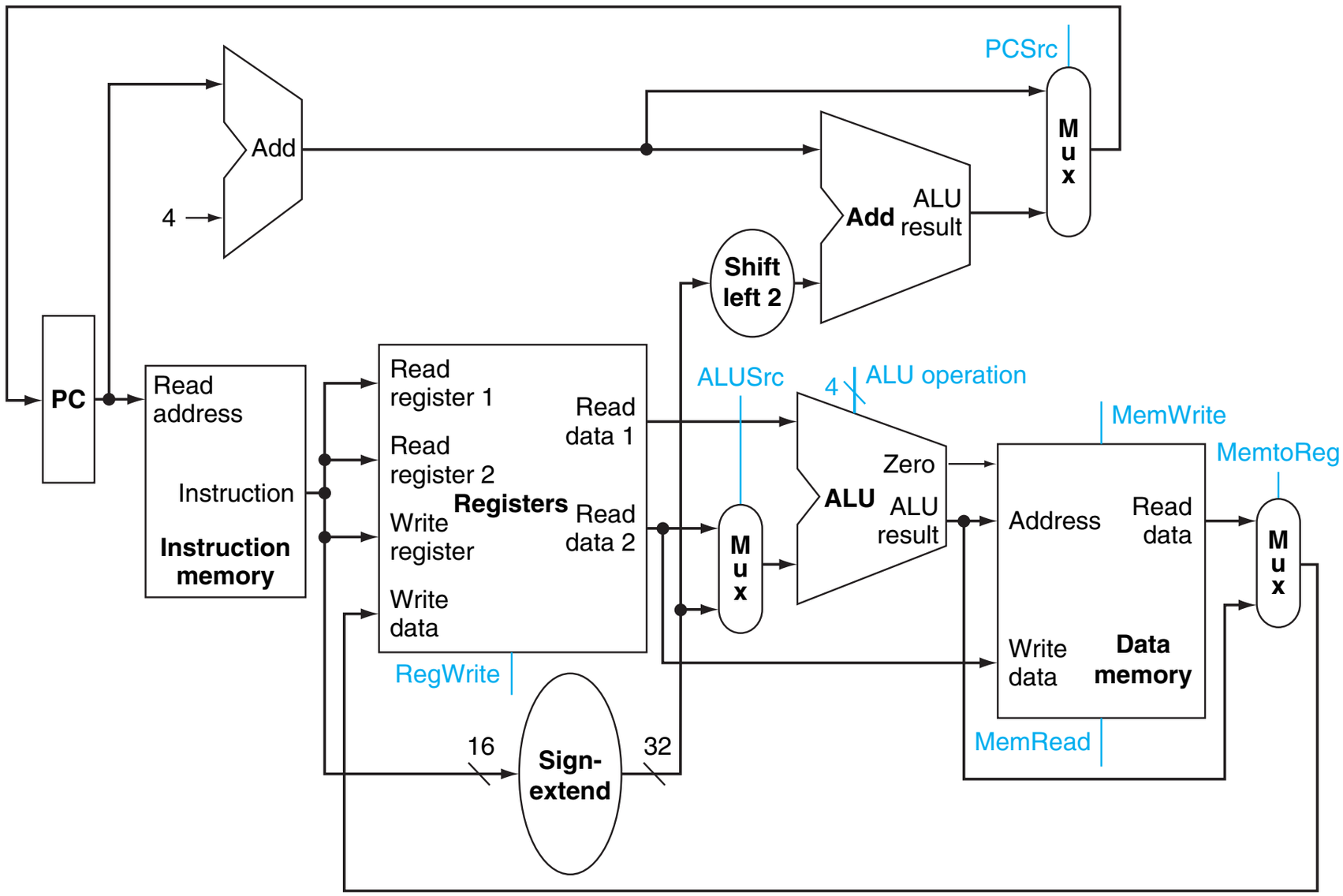
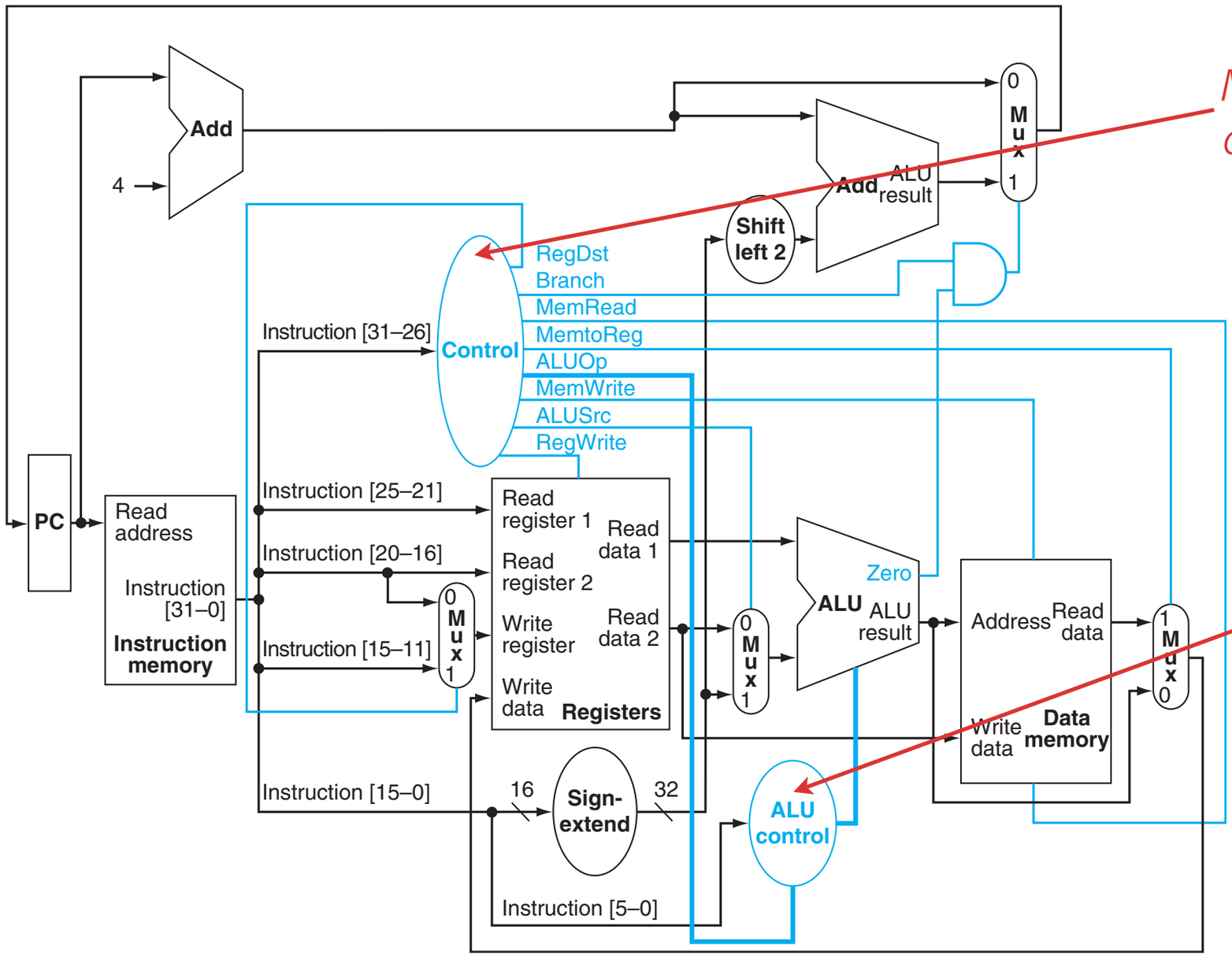


FIGURE 4.11 The simple datapath for the MIPS architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. An additional multiplexor is needed to integrate branches. The support for jumps will be added later.

MIPS Datapath Control

Datapath Control Scheme

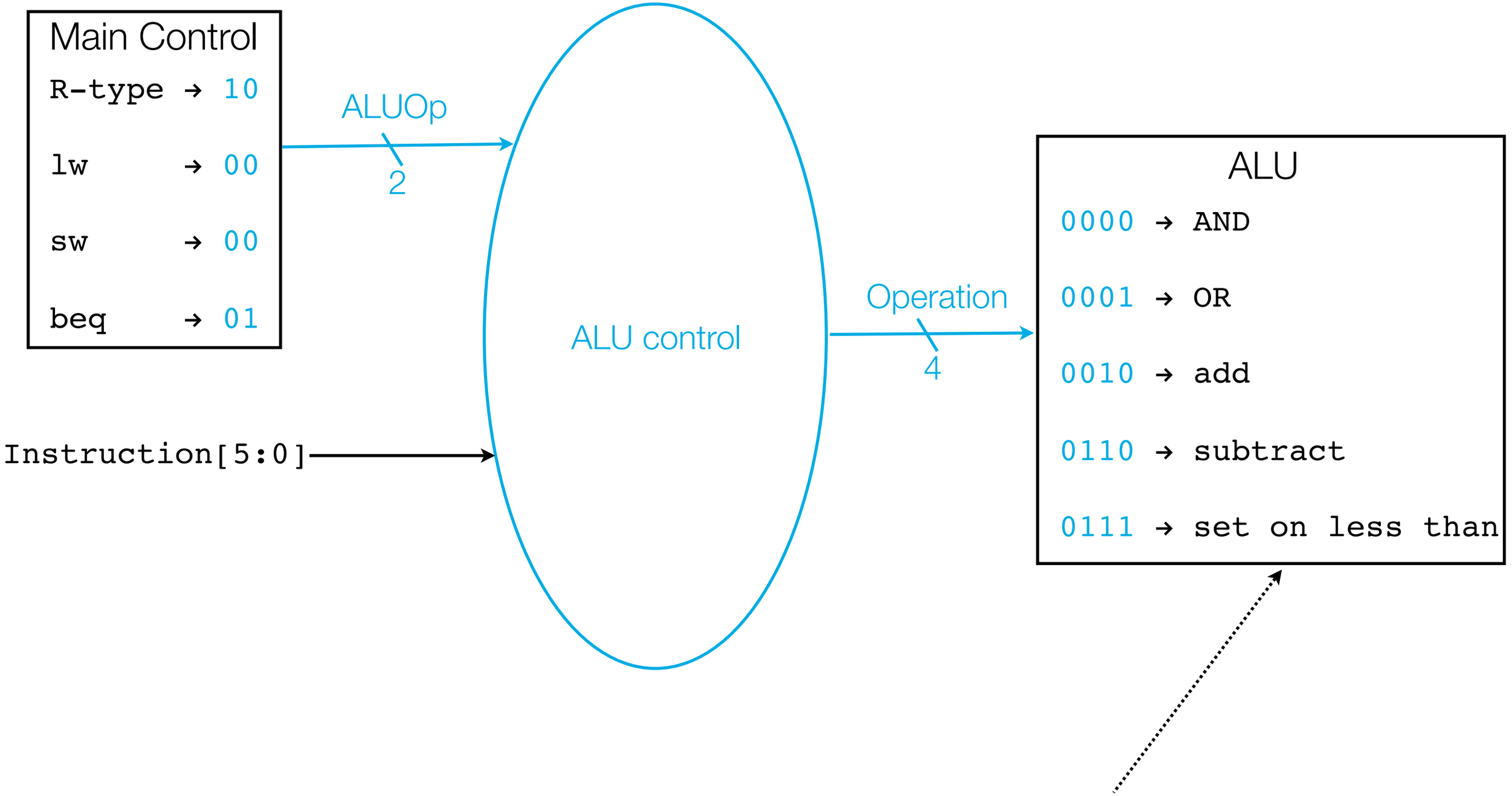


Main control controls whole datapath based on opcode

ALU control controls ALU based on opcode (ALUOp) and function field (funct)

FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

ALU Control Inputs/Outputs



(See Appendix C of text for implementation of corresponding ALU.)

ALU Control Implementation

opcode	ALUOp from main control	Instruction[5:0]		Operation
lw	→ 00	xxxxxx → load word	→ add	→ 0010
sw	→ 00	xxxxxx → store word	→ add	→ 0010
beq	→ 01	xxxxxx → branch equal	→ subtract	→ 0110
R-type	→ 10	100000 → add	→ add	→ 0010
R-type	→ 10	100010 → subtract	→ subtract	→ 0110
R-type	→ 10	100100 → AND	→ AND	→ 0000
R-type	→ 10	100101 → OR	→ OR	→ 0001
R-type	→ 10	101010 → set on less than	→ set on less than	→ 0111

ALU Control Truth Table

ALUOp from main control	Instruction[5:0]	Operation
00	xxxxxx	0010
00	xxxxxx	0010
01	xxxxxx	0110
10	100000	0010
10	100010	0110
10	100100	0000
10	100101	0001
10	101010	0111

ALU Control Truth Table 2

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Datapath Control Scheme

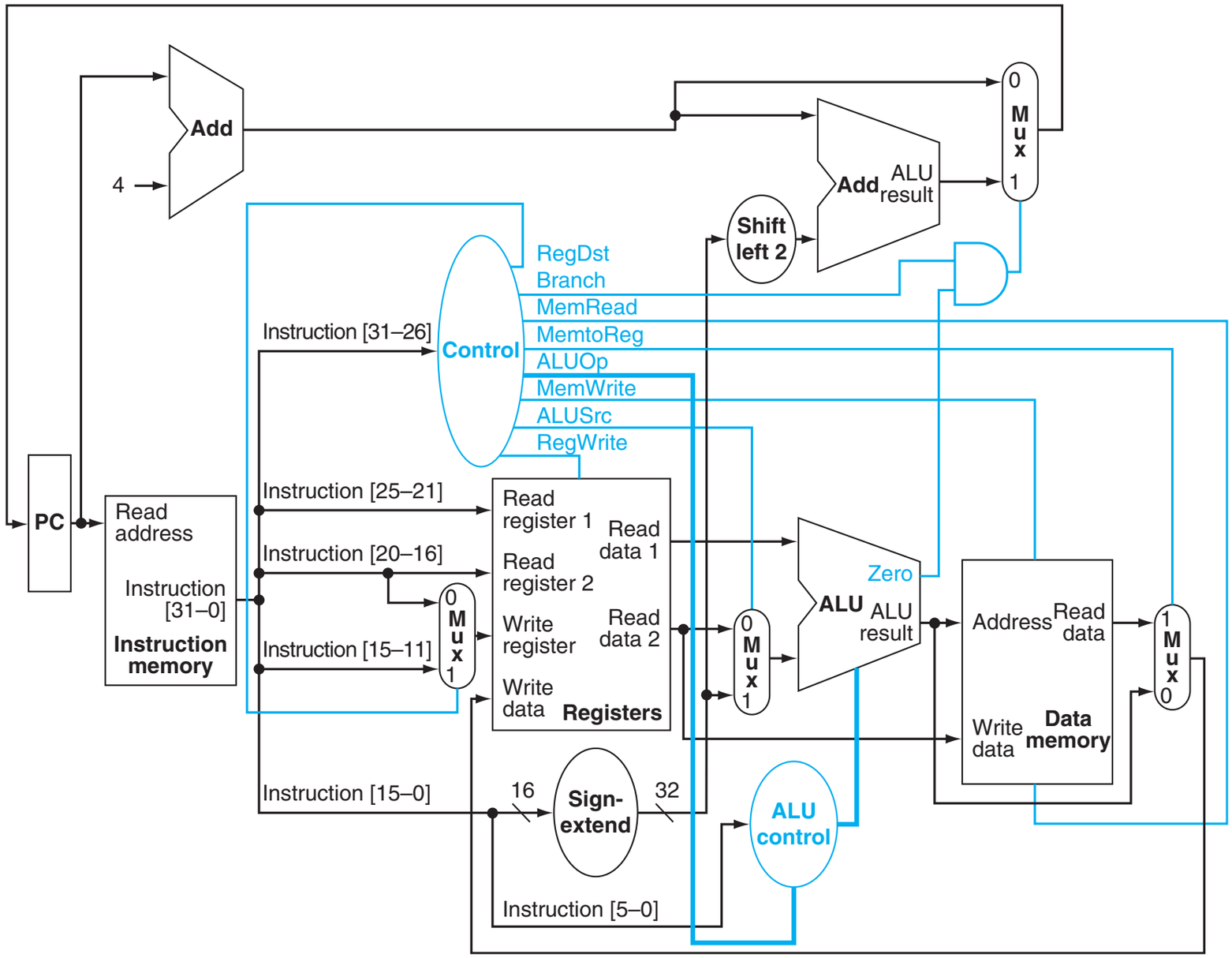
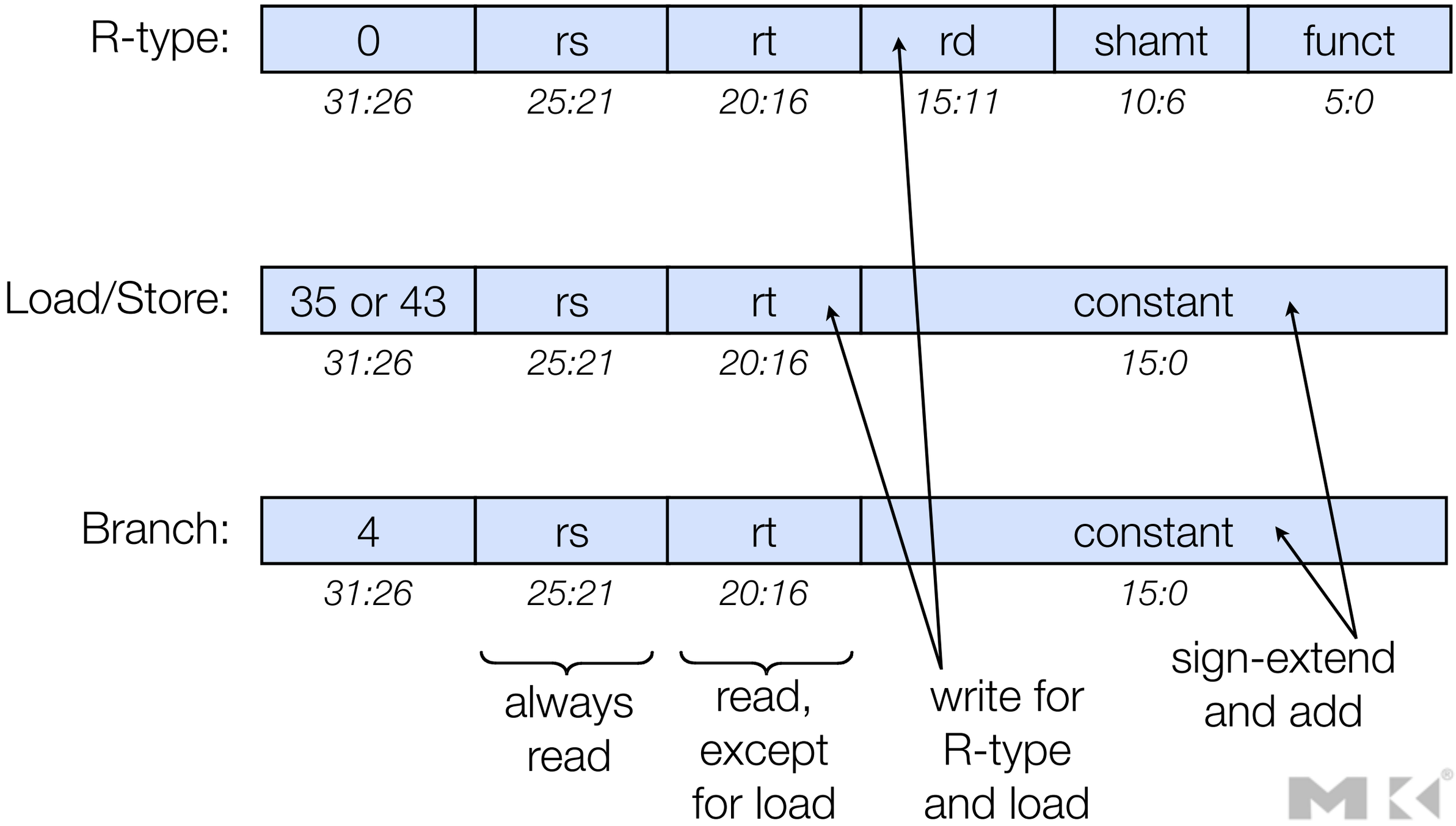
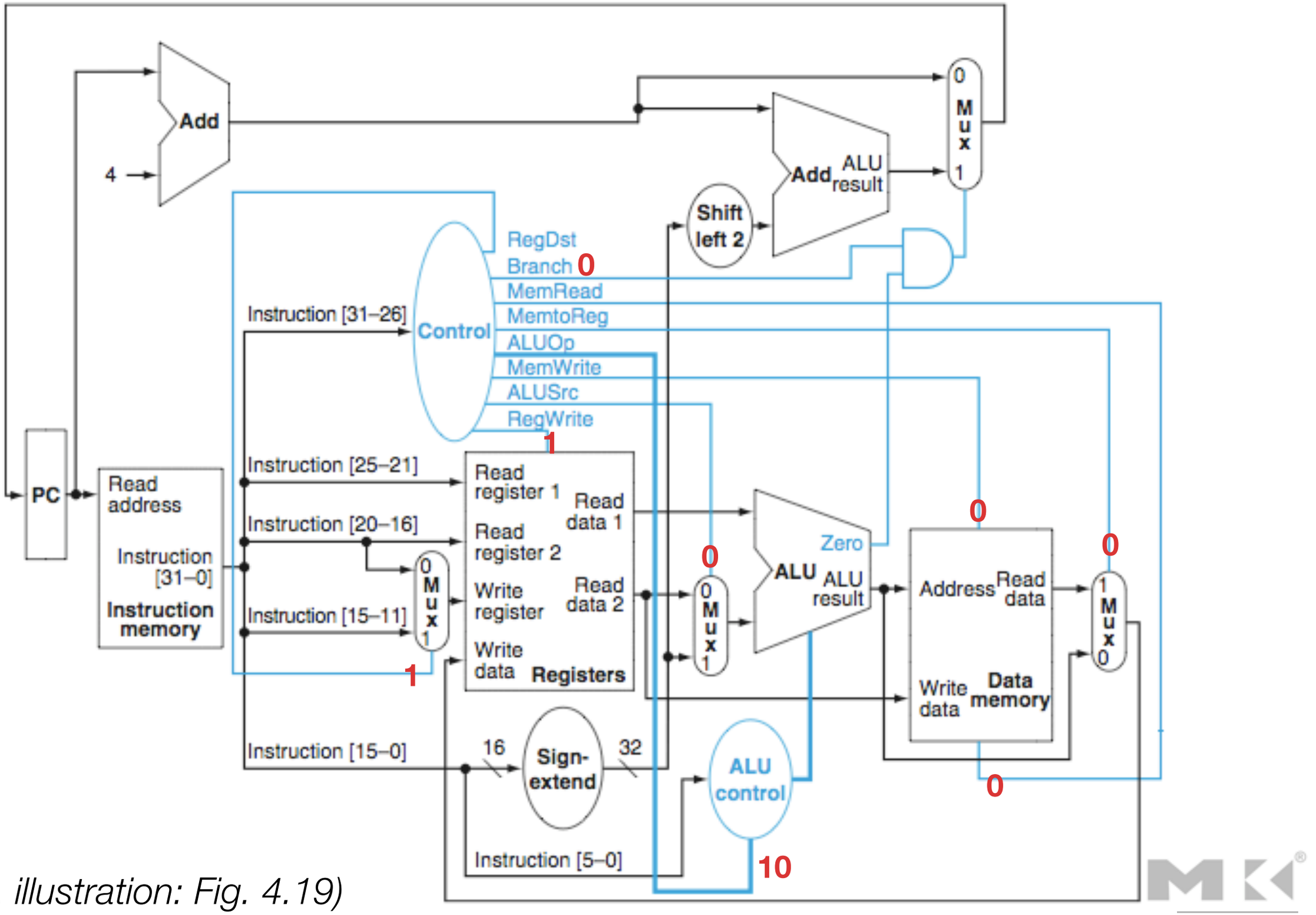


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

Main control signals derive from instruction types

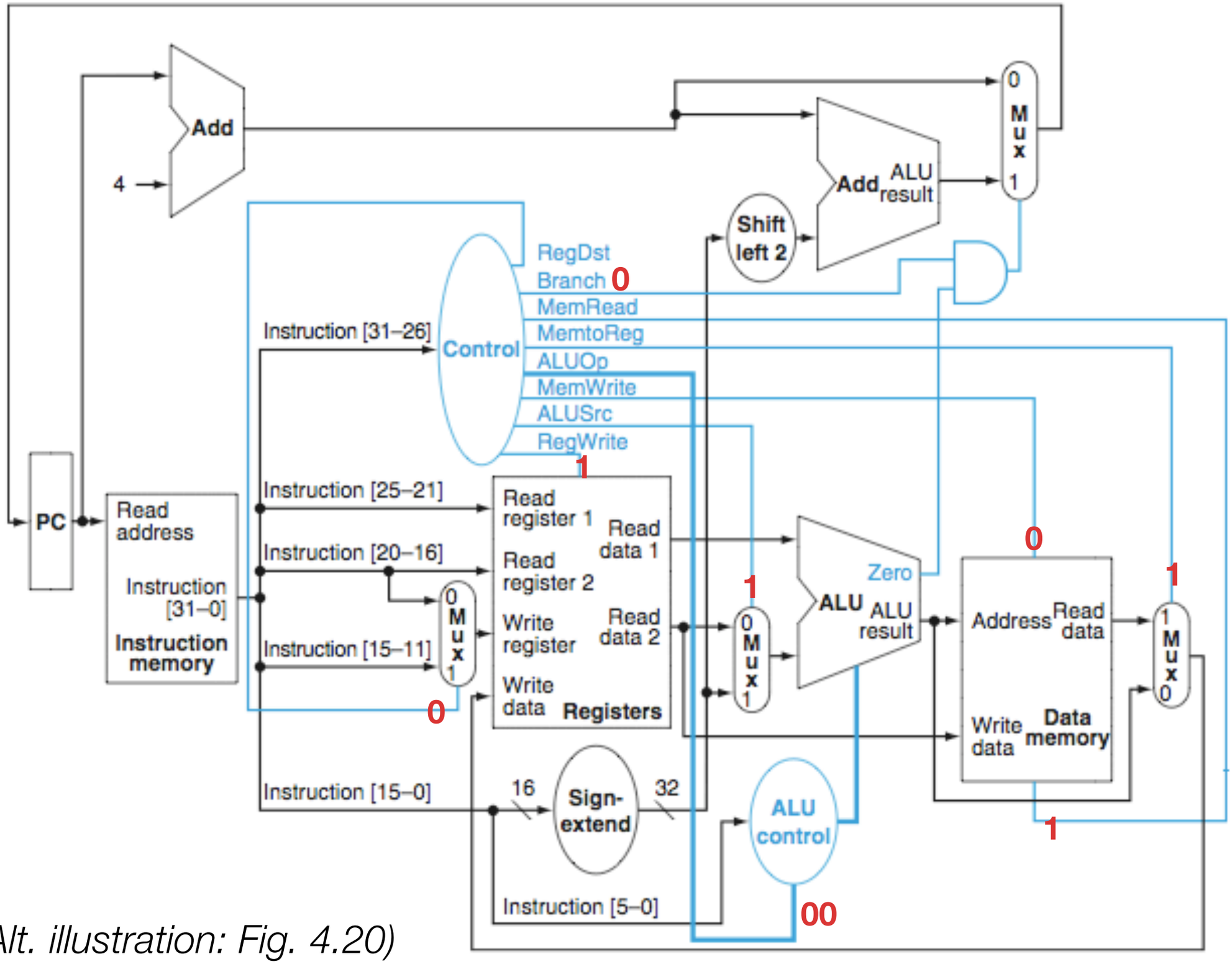


R-Type Control Signals



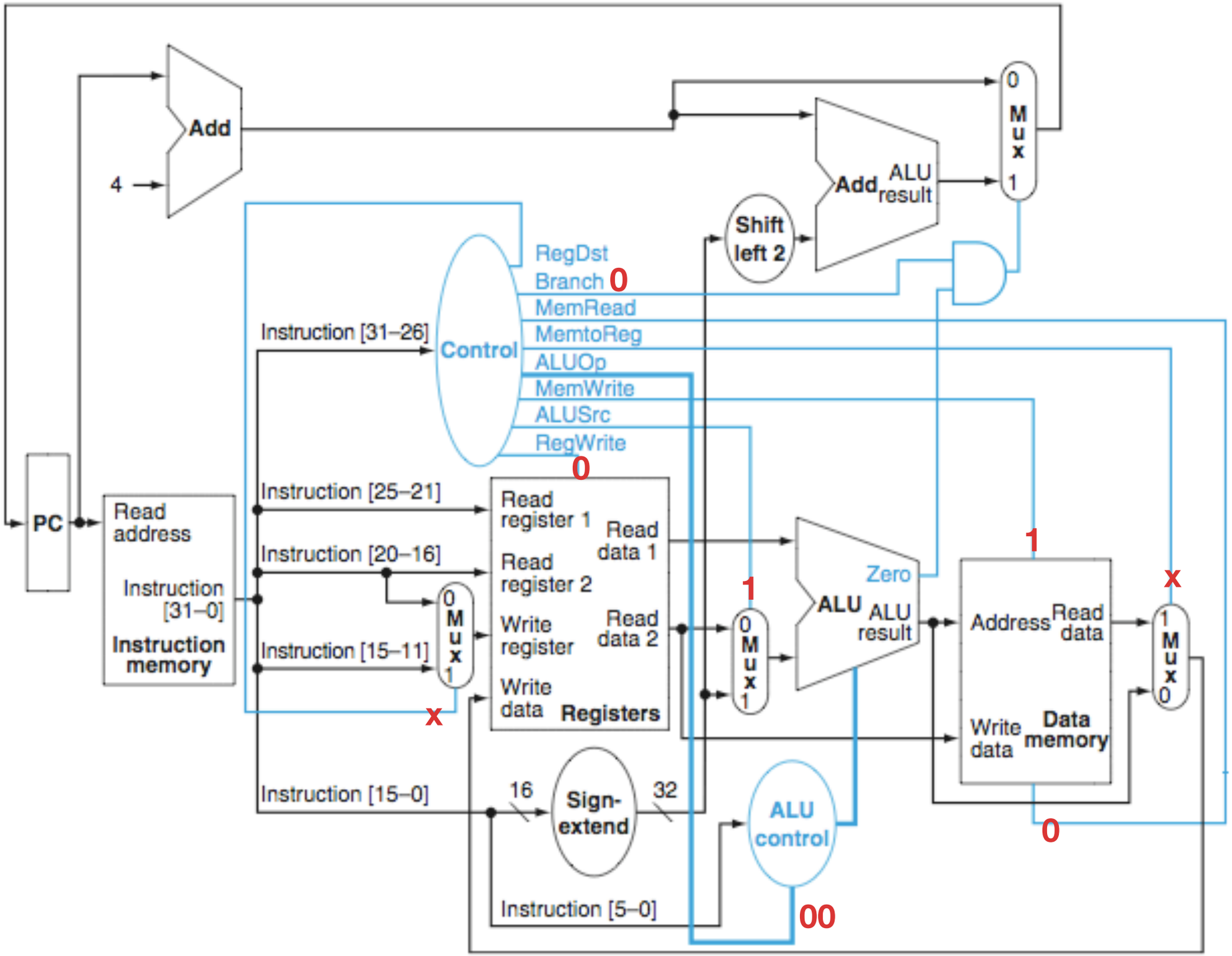
(Alt. illustration: Fig. 4.19)

`lw` Control Signals

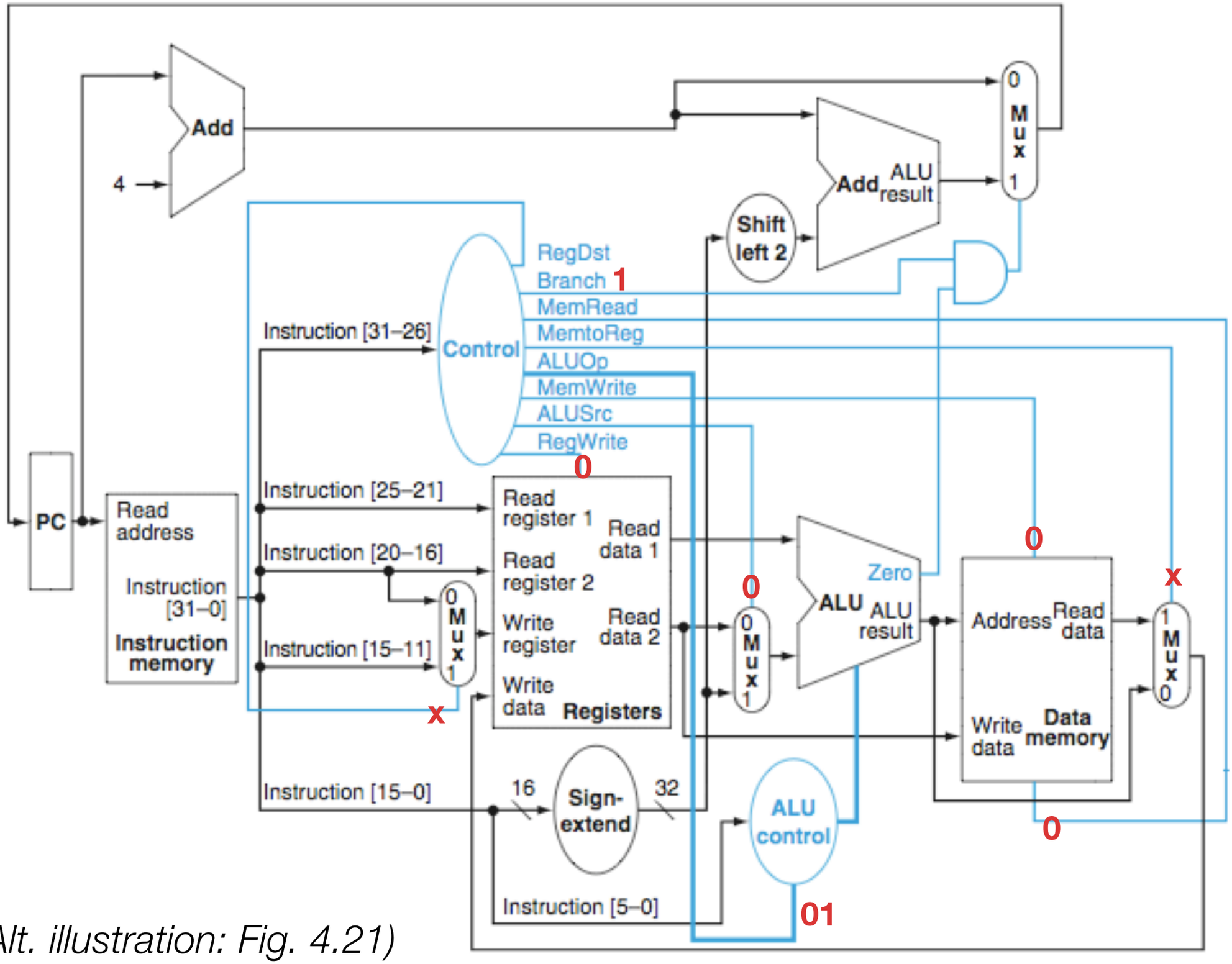


(Alt. illustration: Fig. 4.20)

`sw` Control Signals



`beq` Control Signals



(Alt. illustration: Fig. 4.21)

Main Control Truth Table

Instruction[31:26]	Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
000000	R-format	1	0	0	1	0	0	0	1	0
100011	lw	0	1	1	1	1	0	0	0	0
101011	sw	X	1	X	0	0	1	0	0	0
000100	beq	X	0	X	0	0	0	1	0	1

FIGURE 4.18 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. Furthermore, an 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. The second and third rows of this table give the control signal settings for `lw` and `sw`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

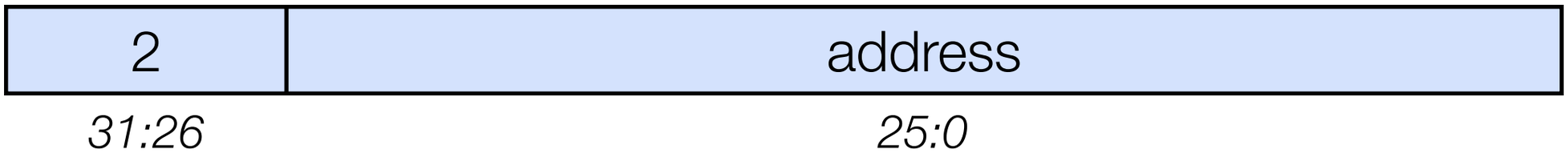
Implementing Jumps

The `j` instruction

- Unconditional jump to instruction at `label`

```
j label
```

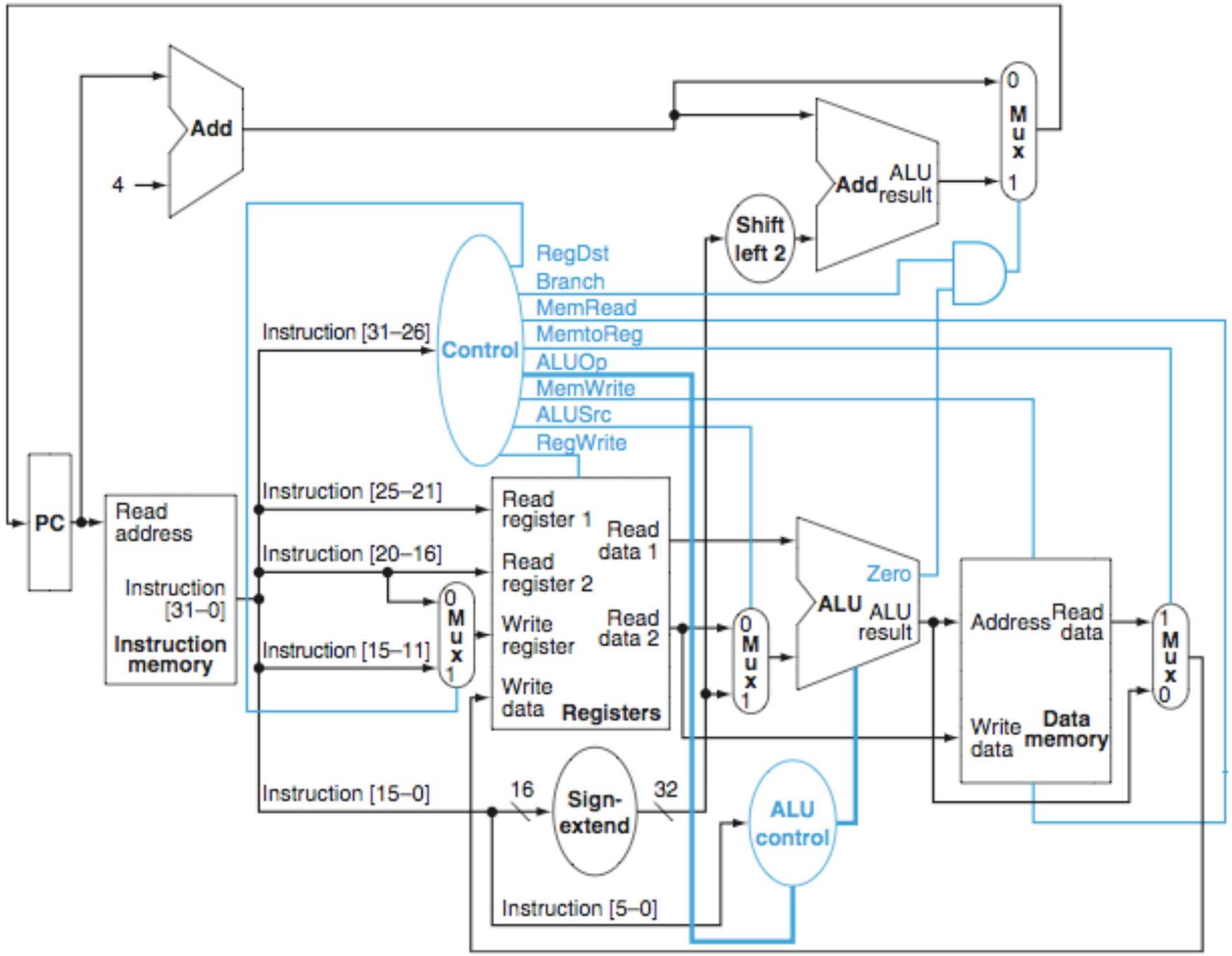
- Instruction encoded in J-type format



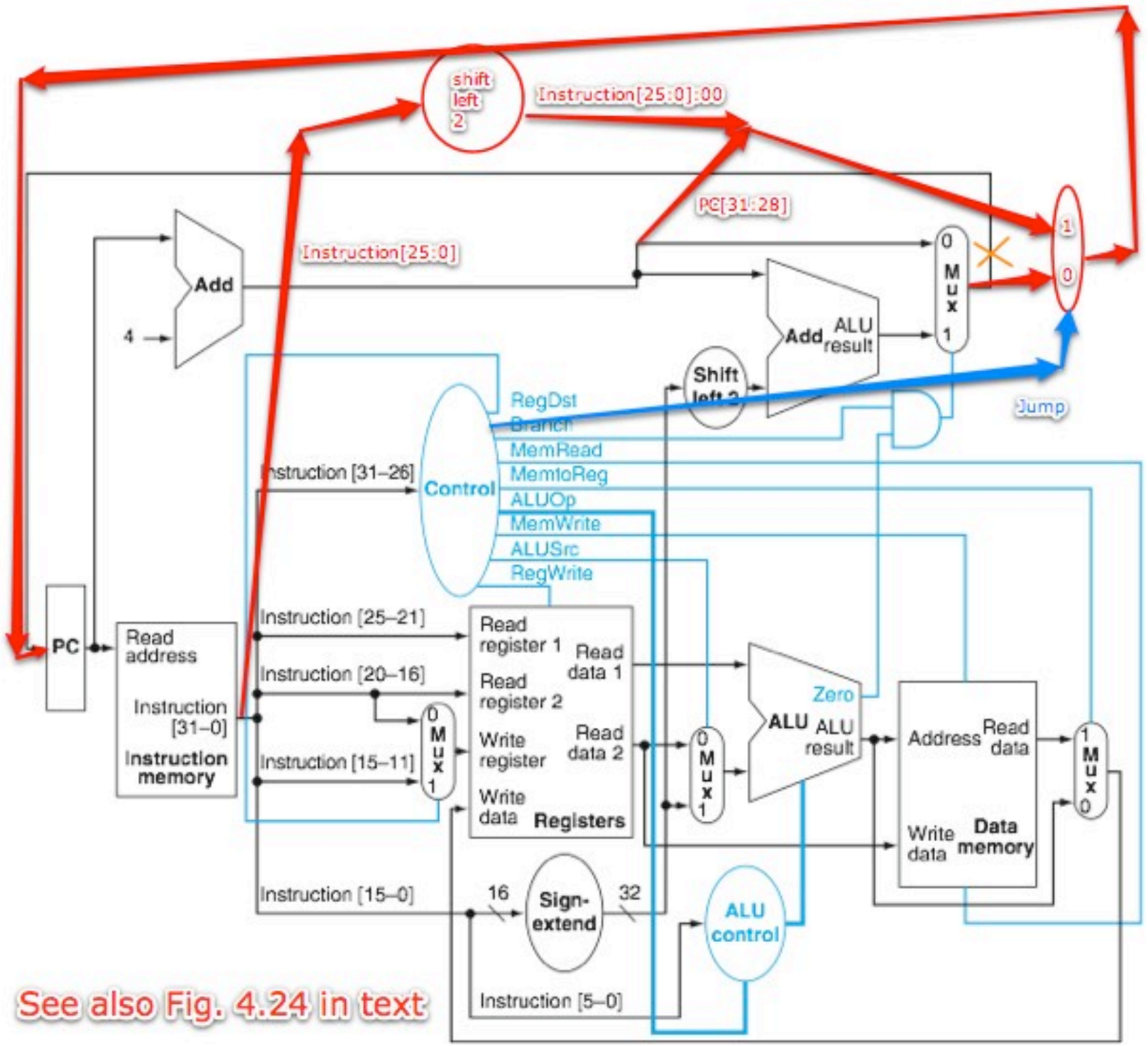
- Jump uses word addresses

- Update PC with concatenation of:
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00

Implementing the jump instruction



Implementing the jump instruction -- in class soln



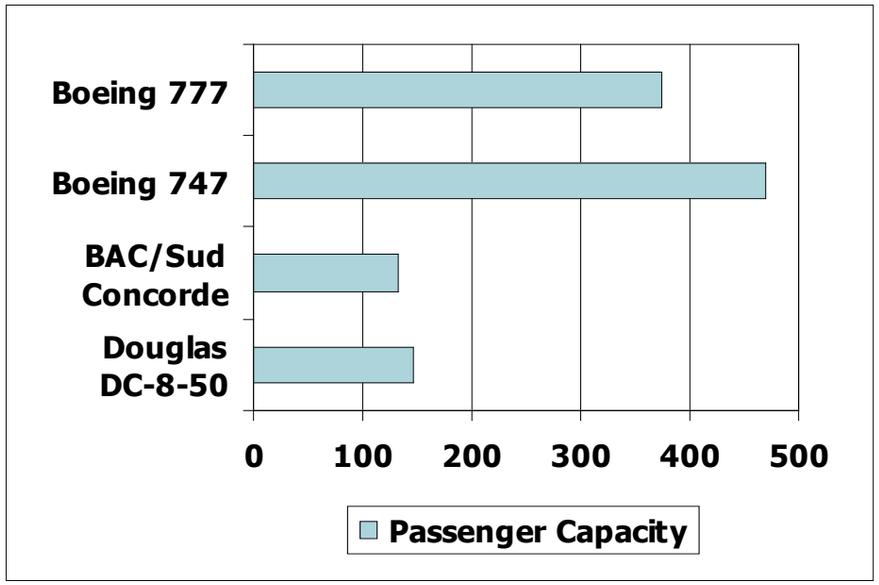
CPU Performance

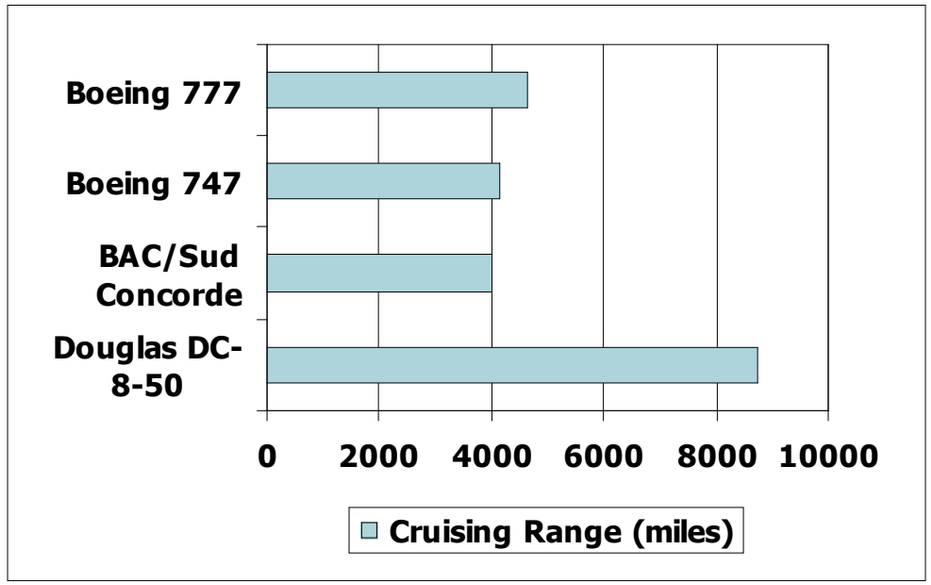
Understanding Performance

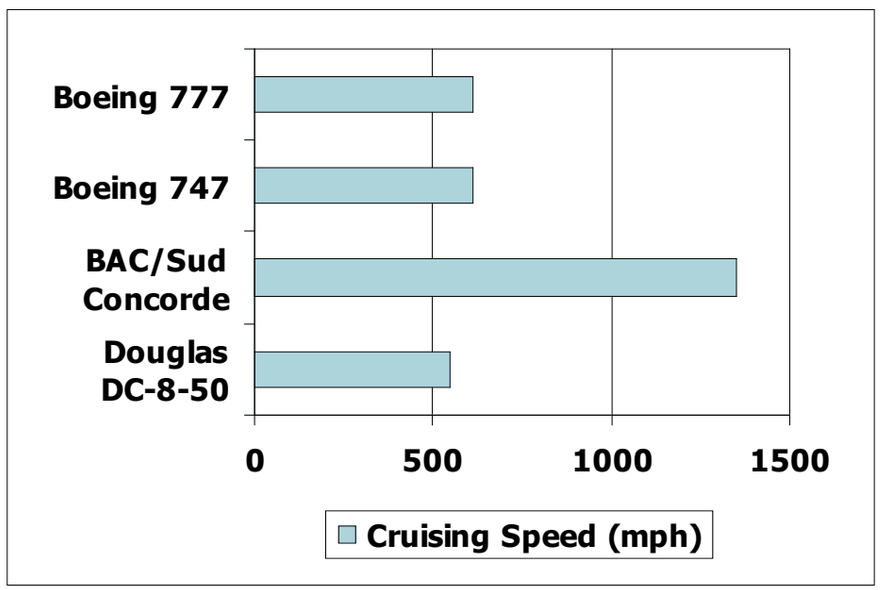
- Algorithm → number of operations executed
- Programming language, compiler, architecture → determine number of machine instructions executed per operation
- Processor and memory system → determines how fast instructions are executed
- I/O system (including OS) → determines how fast I/O operations are executed

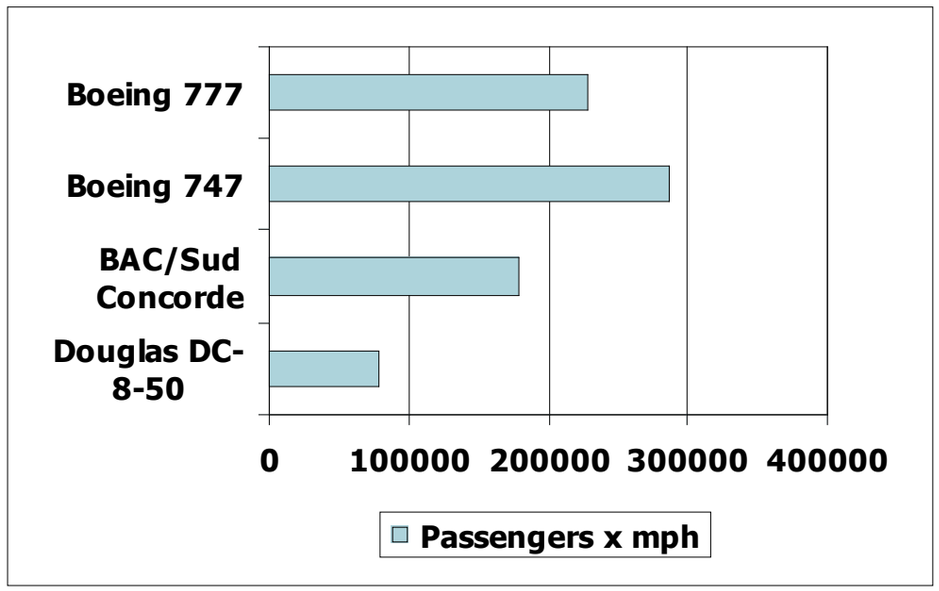
Defining Performance

- Which airplane has the best performance?









Response Time and Throughput

Response time:
how long it takes to do a task,
sometimes also called latency [time/work]

Throughput:
total work done per unit time [work/time]

How are response time and throughput affected by. . .
Replacing the processor with a faster version?

Adding more processors?

For now, we'll focus on response time

Relative Performance

Define: Performance = 1 / Execution Time

"X is n times faster than Y" →
Performance X / Performance Y =
Execution Time Y / Execution Time X =
n

Example:

Program takes 10 s to run on machine A, 15 s on machine B
Execution Time B / Execution Time A = 15 / 10 = 1.5
"A is 1.5 times faster than B"

Measuring Execution Time

Define: Elapsed Time

Total response time including all aspects
(Processing, I/O, overhead, idle time)

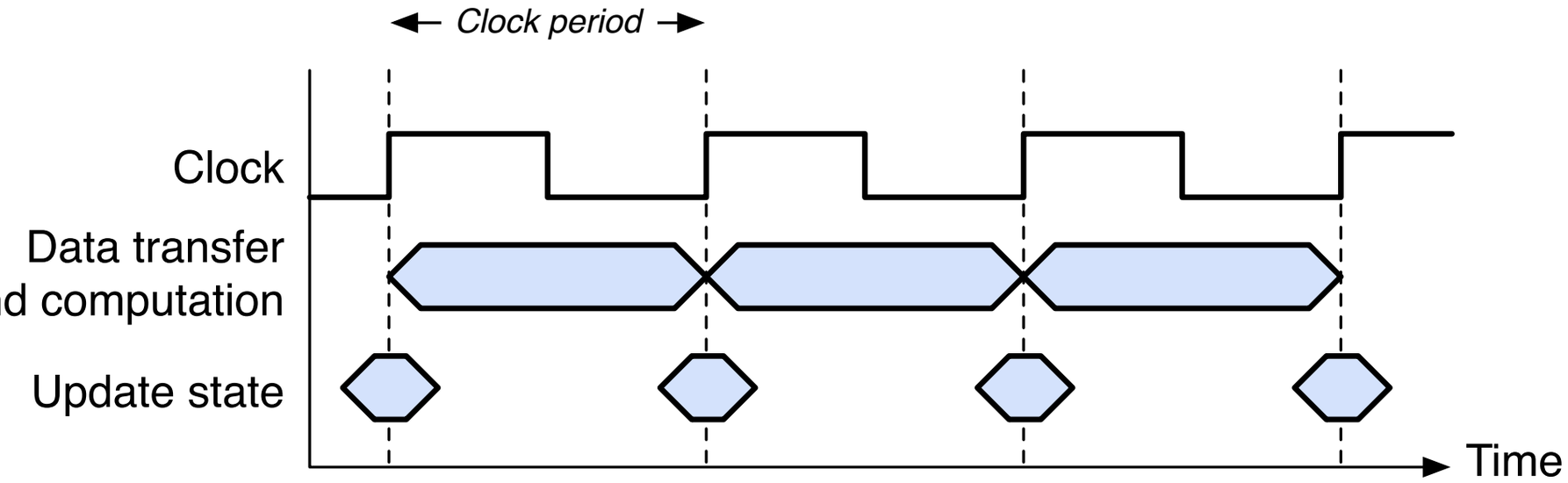
Define: CPU Time

Time spent processing a given job
(discounts I/O time, other jobs shares)

Elapsed Time > CPU Time

CPU Clocking

Operation of digital hardware governed by a constant-rate clock



Clock period: duration of a clock cycle
e.g., 250ps = 0.25ns

Clock frequency (rate): cycles per second
e.g., 4.0GHz = 4000MHz

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} * \text{Clock Cycle Time}$$

$$= \text{CPU Clock Cycles} / \text{Clock Rate}$$

Performance improved by:

1. Reducing number of clock cycles
2. Increasing clock rate (reducing clock period)

Hardware designer must often trade off clock rate against cycle count.

CPU Time Example

Computer A: 2GHz clock, 10s CPU time

Designing Computer B:

- Aim for 6s CPU Time
- Clock rate increase requires 1.2x the number of cycles

How fast must Computer B's clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

Instruction count

Determined by program, ISA, and compiler

Average cycles per instruction (CPI)

- Determined by CPU hardware
- If different instructions have different CPI, can compute a weighted average based on instruction mix

Clock Cycles	=	Instruction Count * Cycles per Instruction
CPU Time	=	Instruction Count * CPI * Clock Cycle Time
	=	(Instruction Count * CPI) / Clock Rate

CPI Example

Computer A: cycle time = 250ps, CPI=2.0
Computer B: cycle time = 500ps, CPI=1.2

Same ISA

Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster...

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

… by this much

Amdahl's Law

Be aware when optimizing. . .

$$T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{unaffected}$$

Example: On machine A, multiplication accounts for 80s out of 100s total CPU time.

How much improvement in multiplication performance to get 5x speedup overall?

Corollary: make the common case fast

Performance Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Algorithm, programming language and compiler compiler affect these terms.

ISA affects all three.

Performance depends on all of these things.

Single-Cycle CPU Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - instruction memory → register file → ALU → data memory → register file
- Not feasible to vary clock period for different instructions
- We will improve performance by pipelining

Pipelining Preview: Laundry Analogy

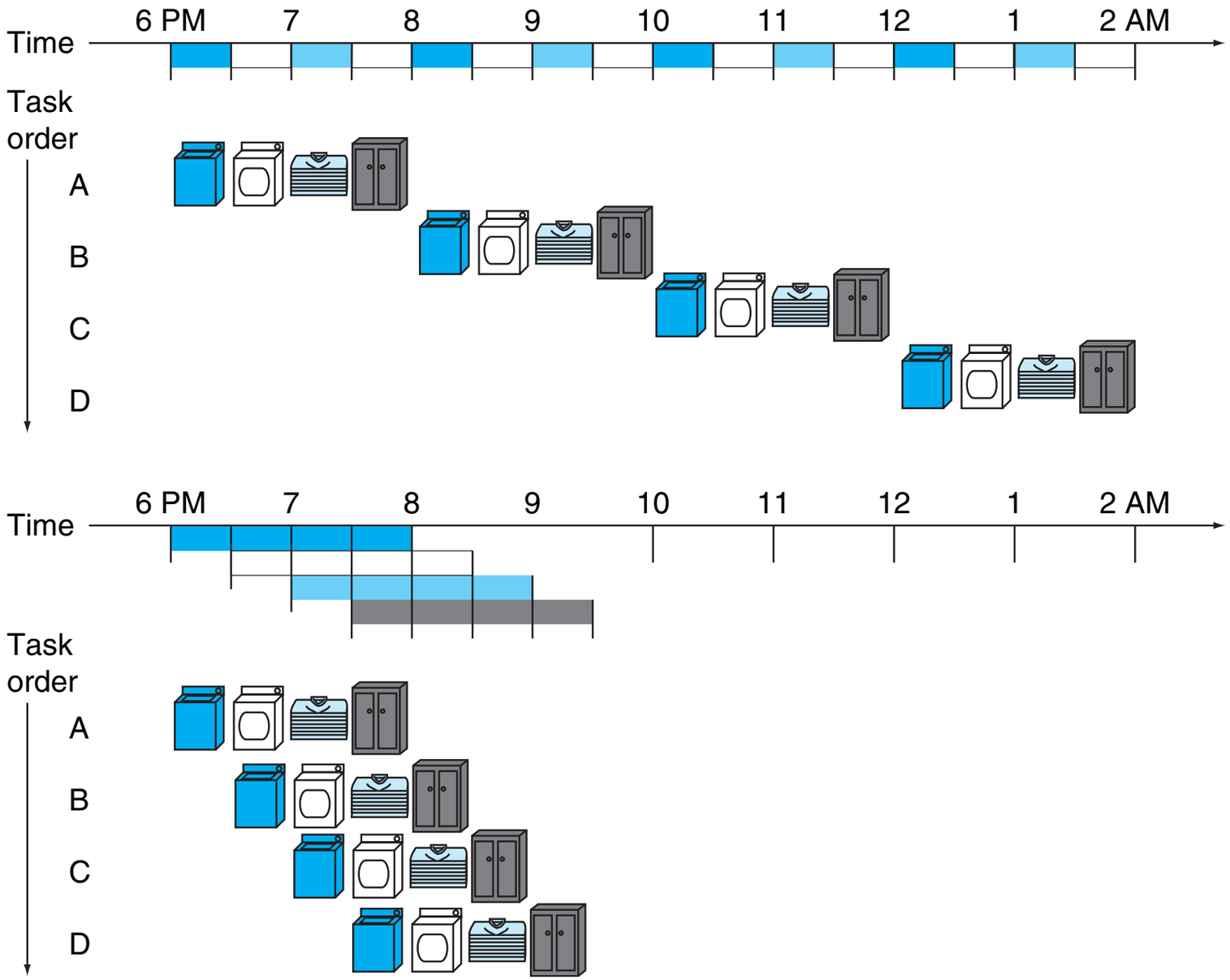


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.