Chapter - 5

The Processor: Datapath and Control

5.1 Introduction

A subset of the core MIPS instruction set:

- The memory-reference instruction – lw, sw
- The arithmetic-logical instruction – add, sub, and, or, slt
- The branch instructions - branch equal (beq), jump (j)

An overview of the implementation

For every instruction, the first two steps are identical:

1. Sent the PC to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instruction require two registers.

An overview of the implementation (Cont.)

Even across different instruction classes there are some similarities: Example

- All instruction classes use the ALU after reading the registers.
  - Memory-reference – address calculation
  - Arithmetic-logic – operation execution
  - Branch – comparison.

An overview of the implementation (Cont.)

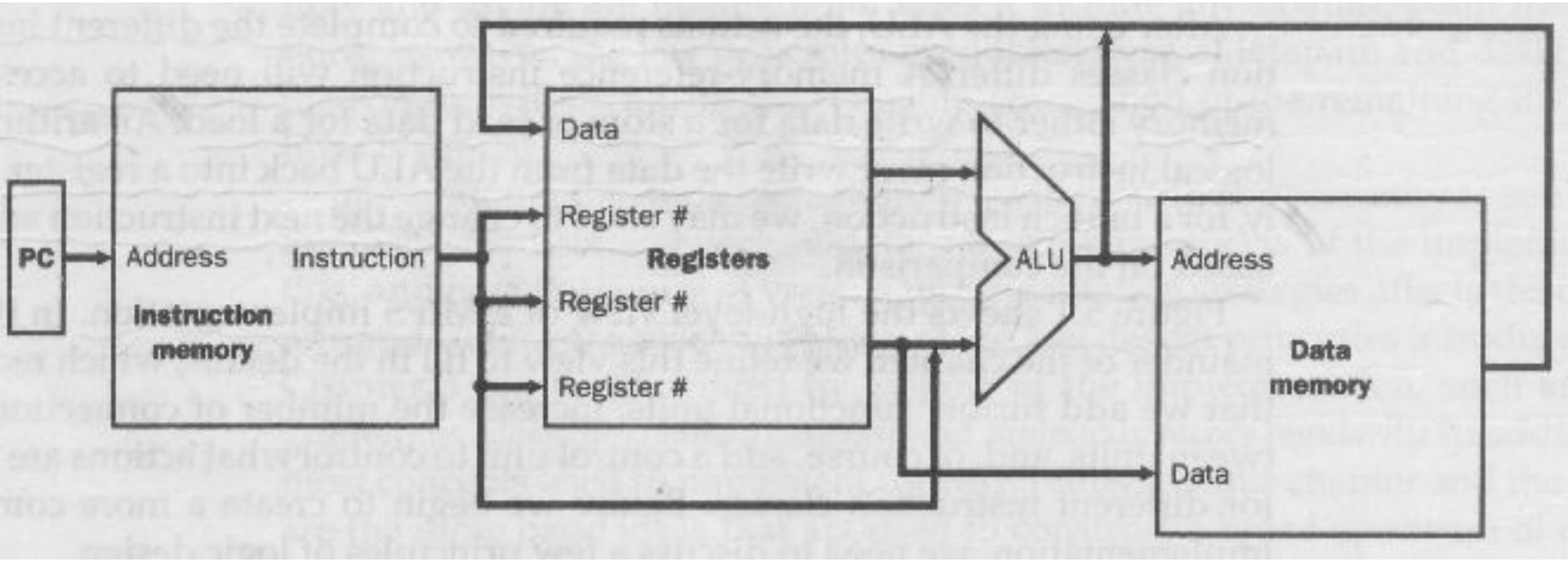


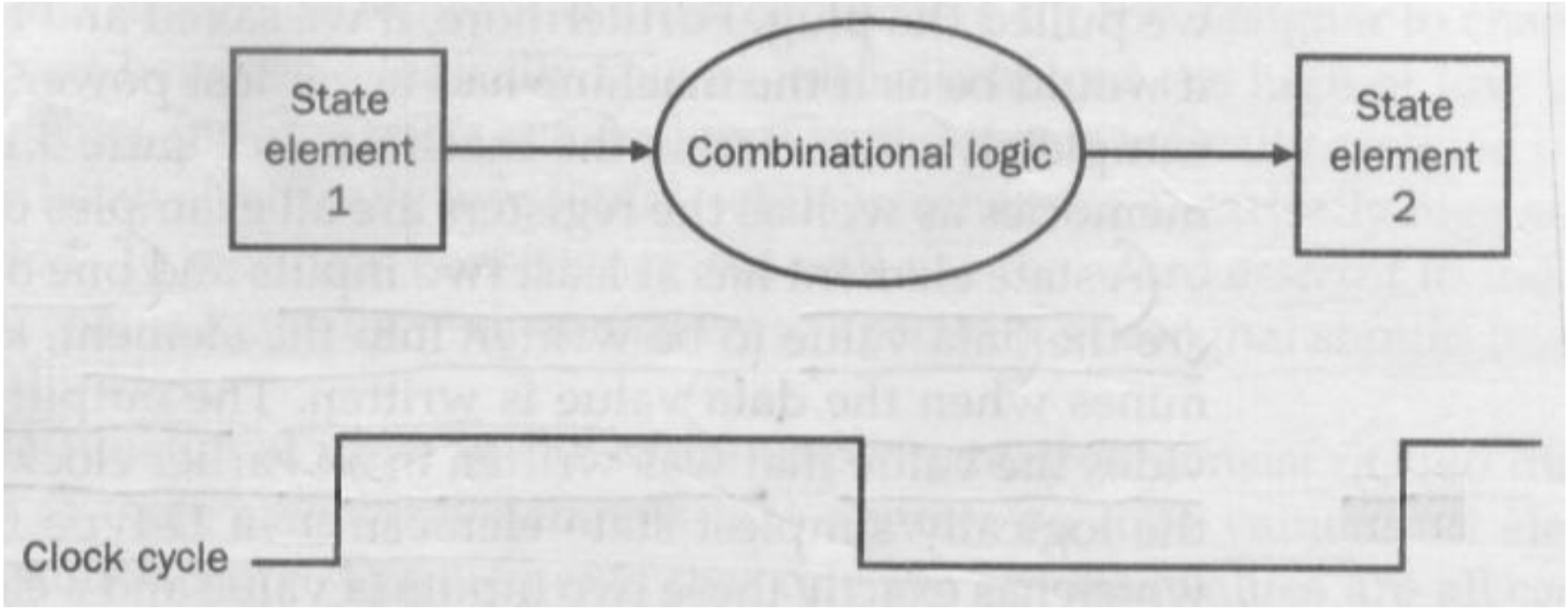
Fig. – 5.1 An abstract view of the implementation.

Clocking Methodology

- A clocking methodology defines when signal can be read and when they can be written.
- If a signal is **written at the same time it is read**, the value of the read could correspond to the old value, the newly written value, or even some mix of two **clocking methodology prevents this circumstances**
- We assume *edge-triggered* clocking.
- Any values stored in the machine are updated only on a clock edge.

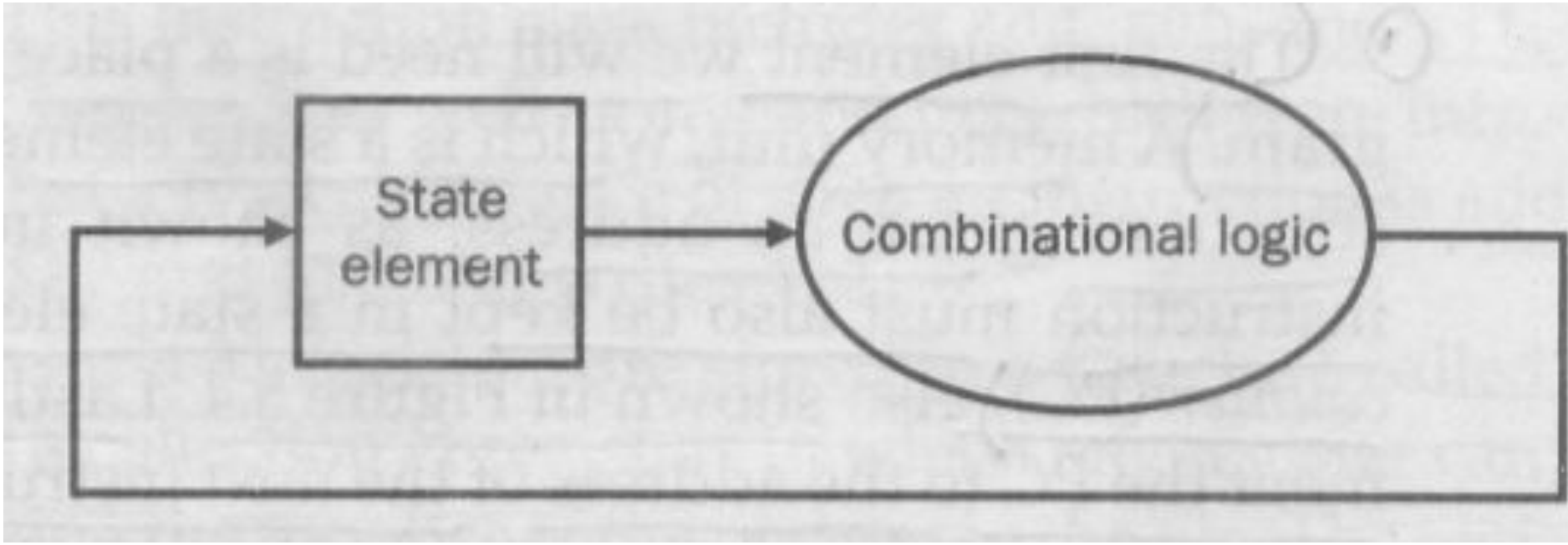
Clocking Methodology (Cont.)

- Combinational logic, which operate in a single clock cycle.

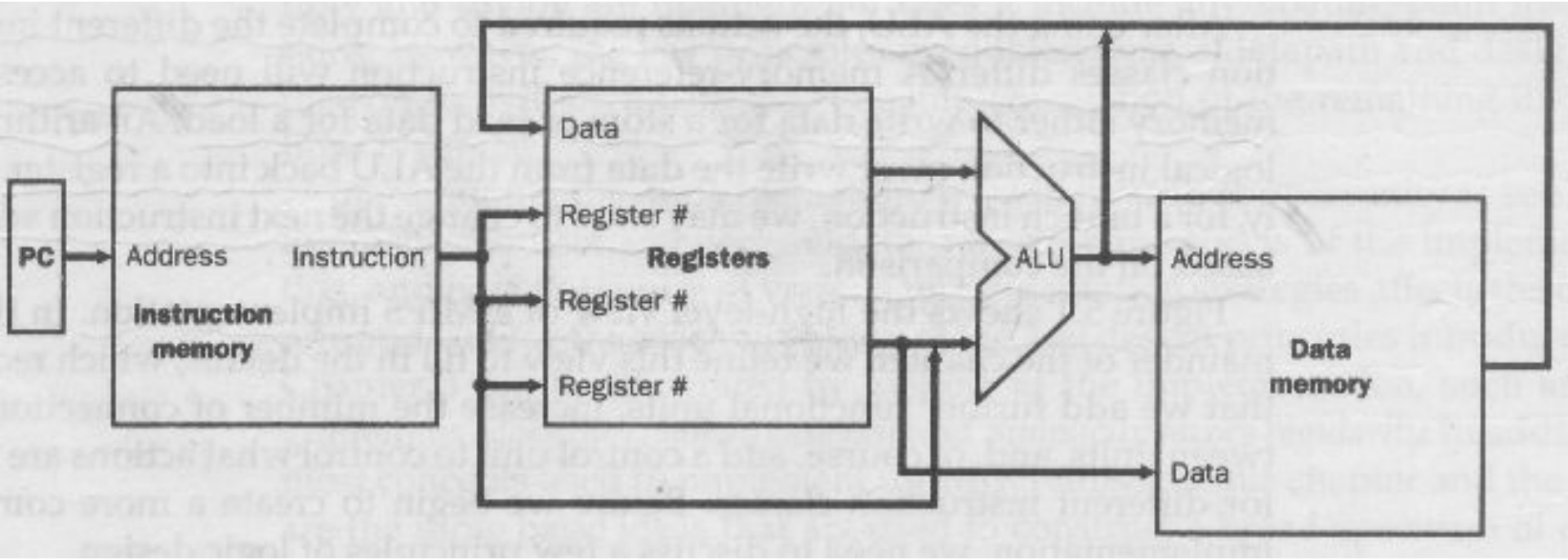


Clocking Methodology (Cont.)

- An edge triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write the register in the same clock cycle.
- Designs in this chapter and the next rely on the edge-triggered timing methodology.



The MIPS Subset Implementation



Uses a single long clock cycle for every instruction.

Begins execution on one clock edge and completes execution on the next clock edge.

Building a Datapath

- **Fetching Instructions**

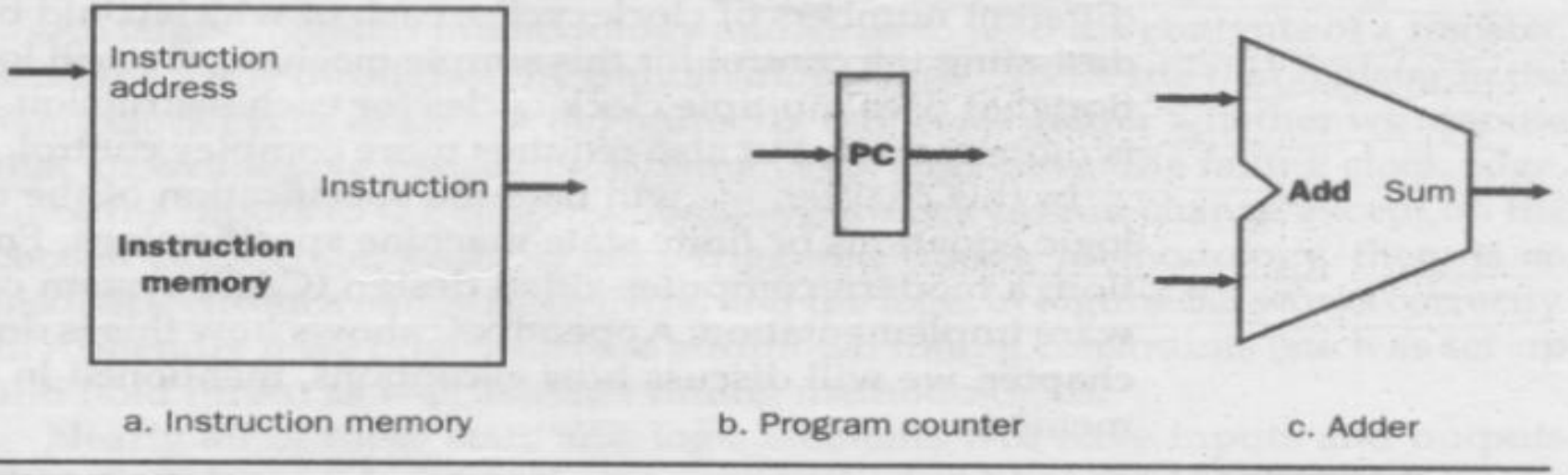
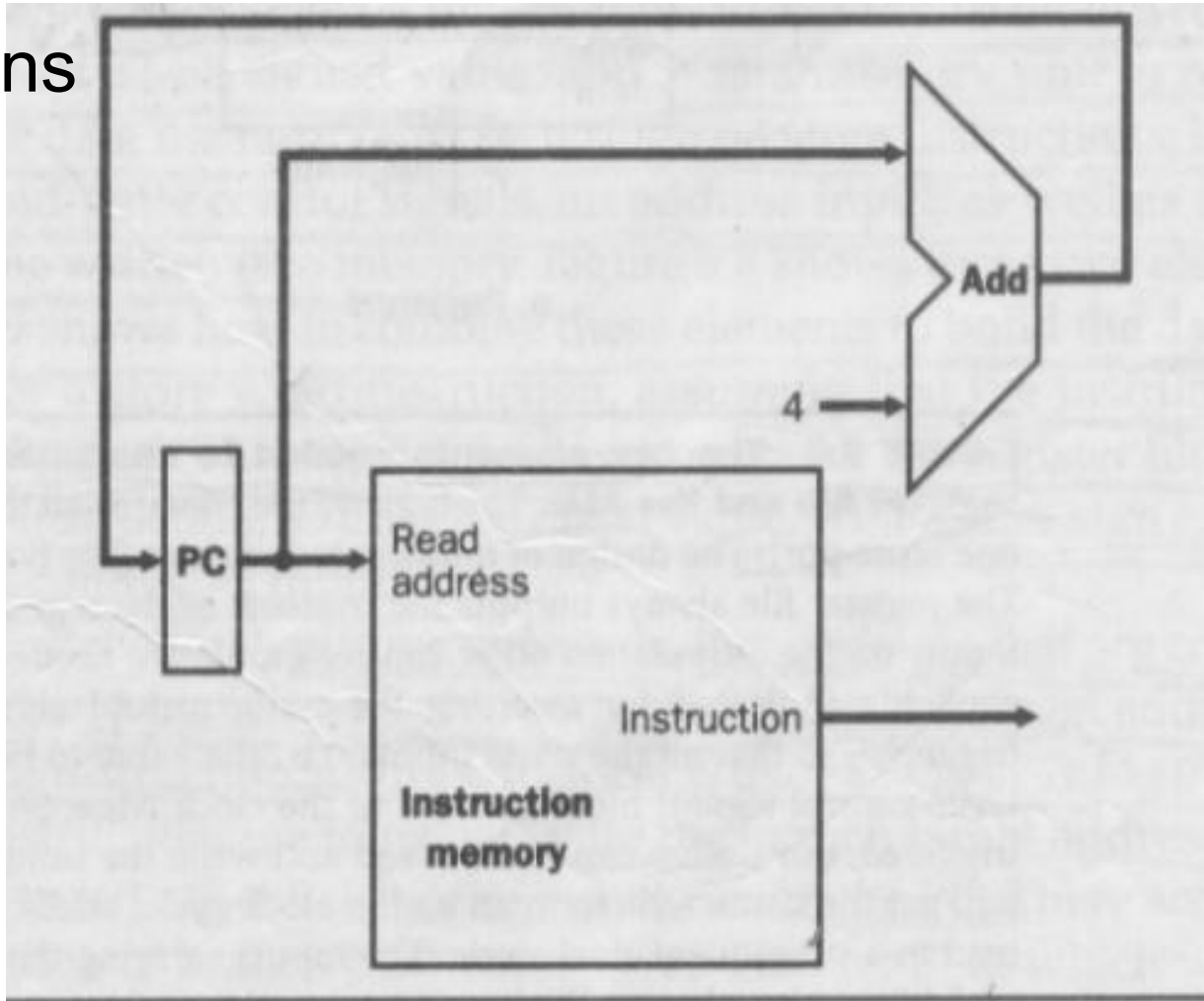


FIGURE 5.4 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

- Instruction memory (IM) and PC are state elements.
- IM is used to hold and supply instruction given an address
- PC keeps the address of the instructions.
- An adder to increment the PC to the address of the next instruction

Building a Datapath (Cont.)

- Fetching Instructions



A portion of the datapath used for fetching instructions and incrementing the program counter.

Building a Datapath (Cont.)

- Arithmetic-logical Instructions
- add $t1, $t2, $t3

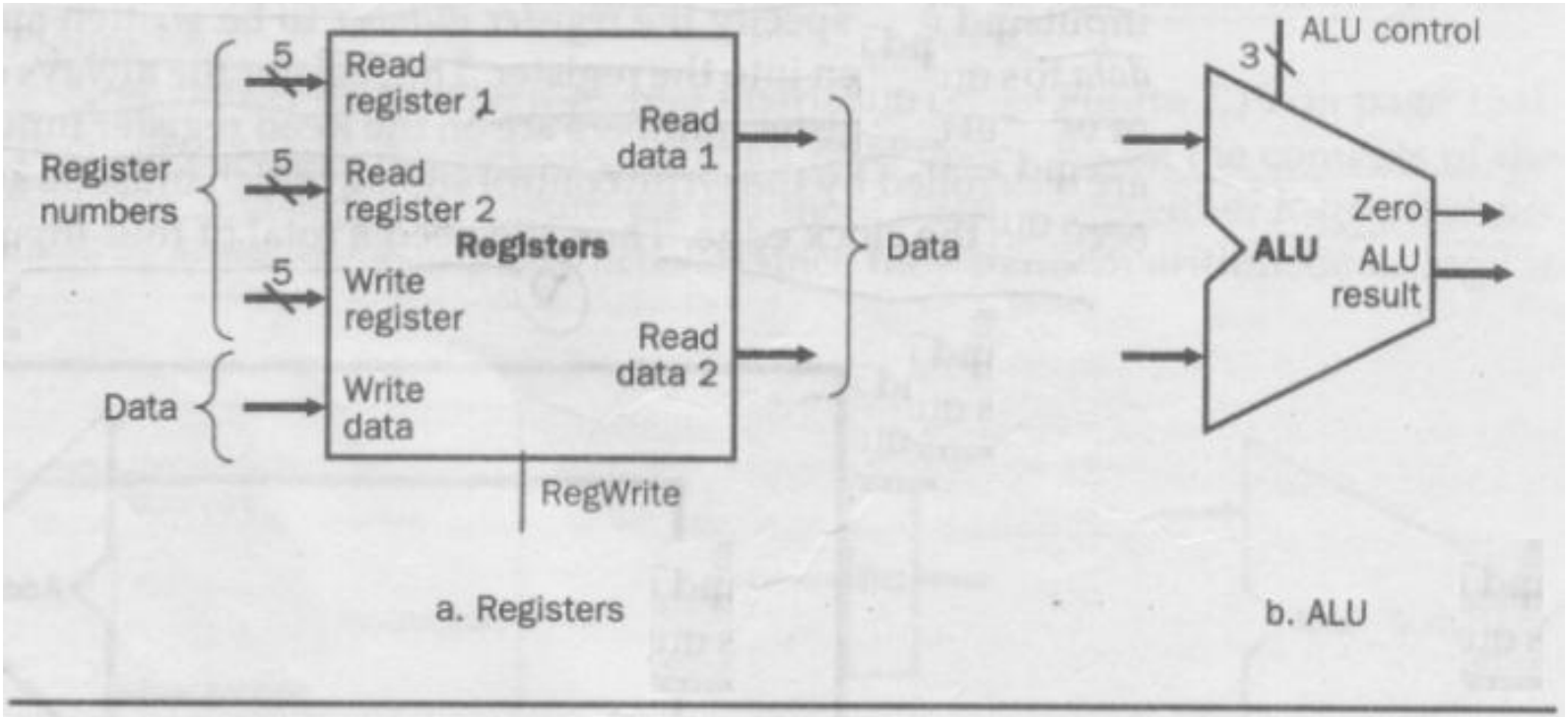


FIGURE 5.6 The two elements needed to implement R-format ALU operations are the register file and the ALU.

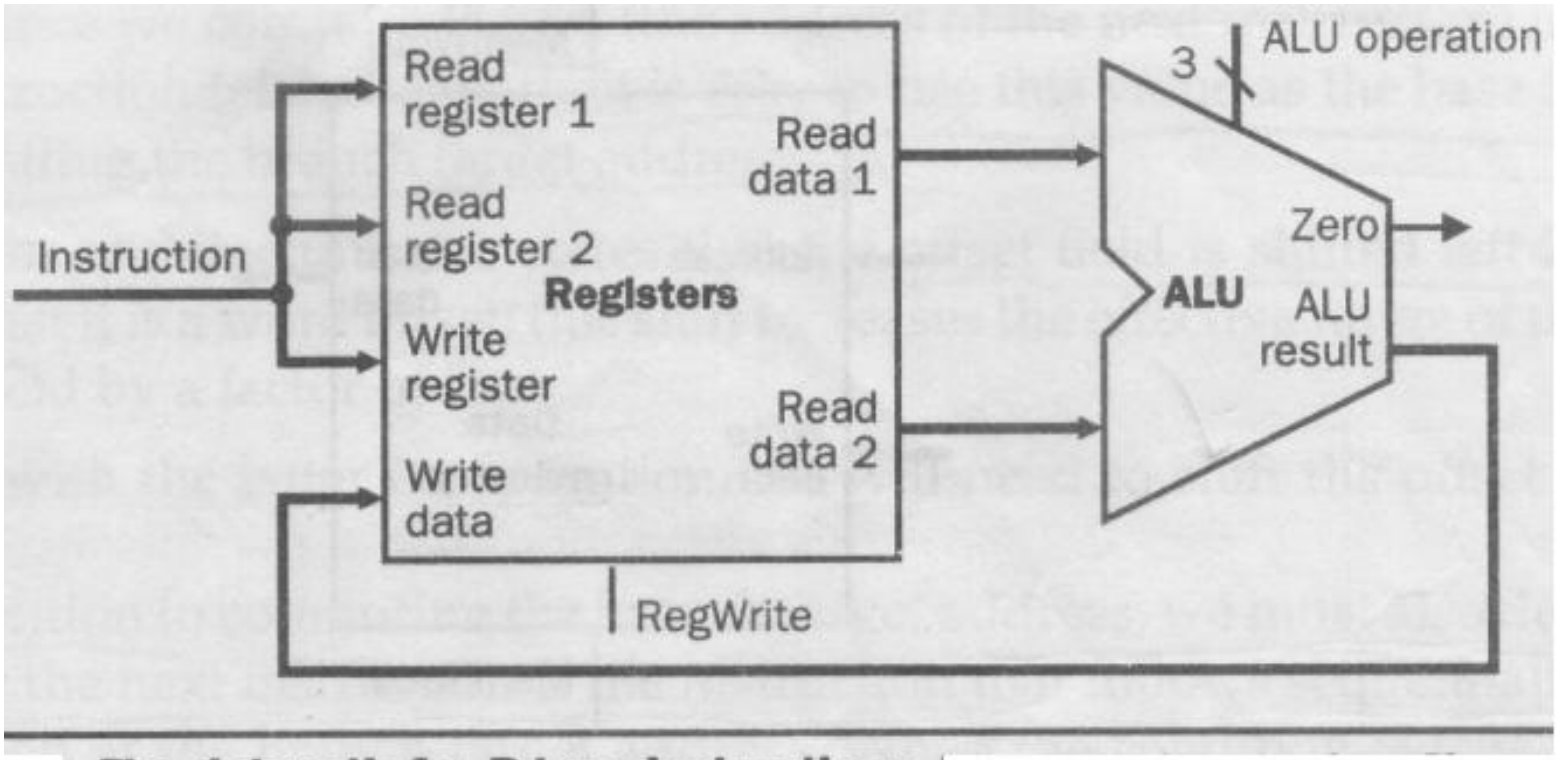
Building a Datapath (Cont.)

- Arithmetic logical operations:

- The register file contains all the registers and has two read ports and one write ports.
- Register file always outputs the contents of whatever register numbers are on the read registers inputs.
- Write is controlled by the write control signal
- We need total four inputs (three for register number and one for data)
- ALU is controlled with the ALU operation signal which is 32 bit wide.

Building a Datapath (Cont.)

- Arithmetic-logical Instructions
- add $t1, $t2, $t3



The datapath for R-type instructions.

Building a Datapath (Cont.)

- Load word / Store word
- lw $s1, 100 ($s2)

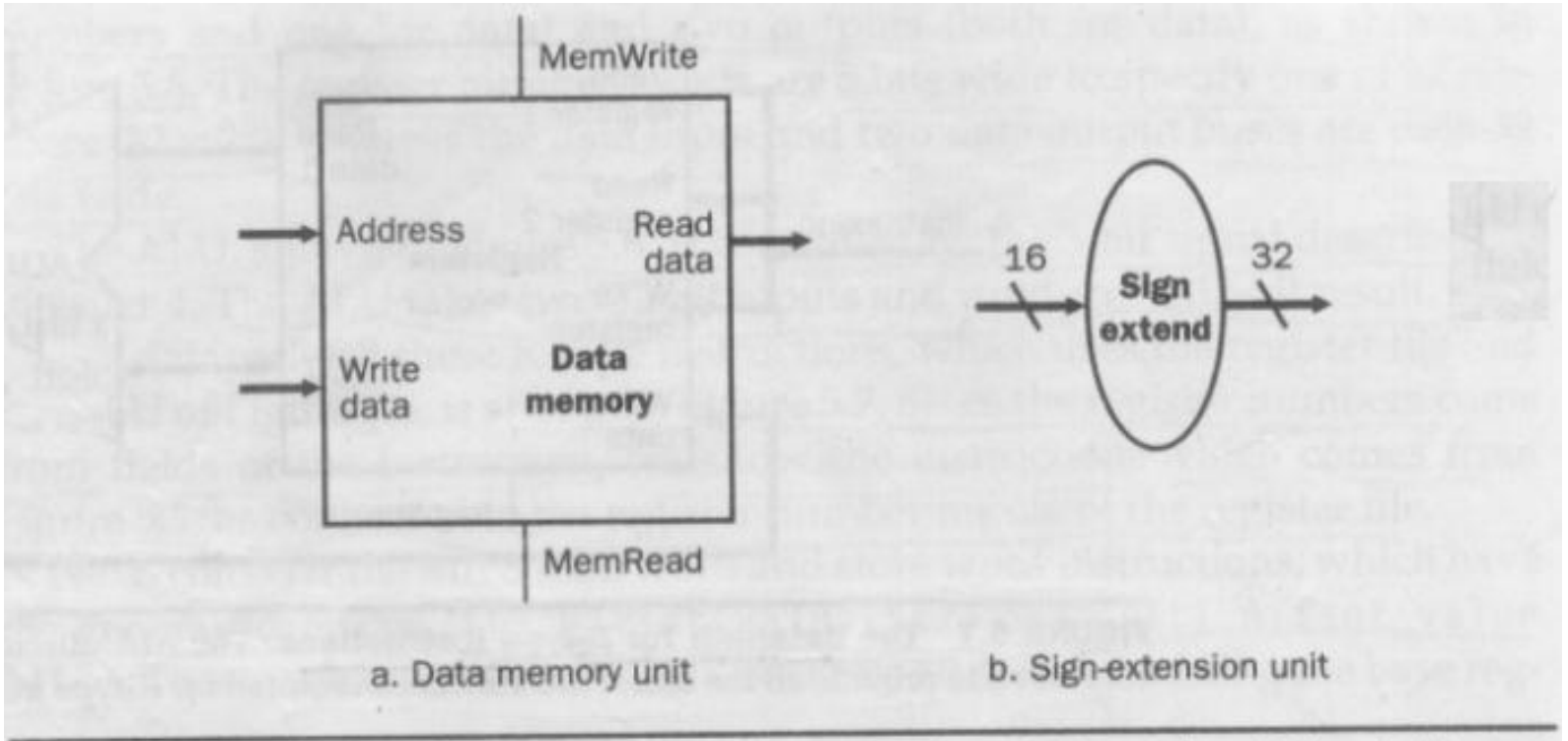


FIGURE 5.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 5.6, are the data memory unit and the sign extension unit.

Building a Datapath (Cont.)

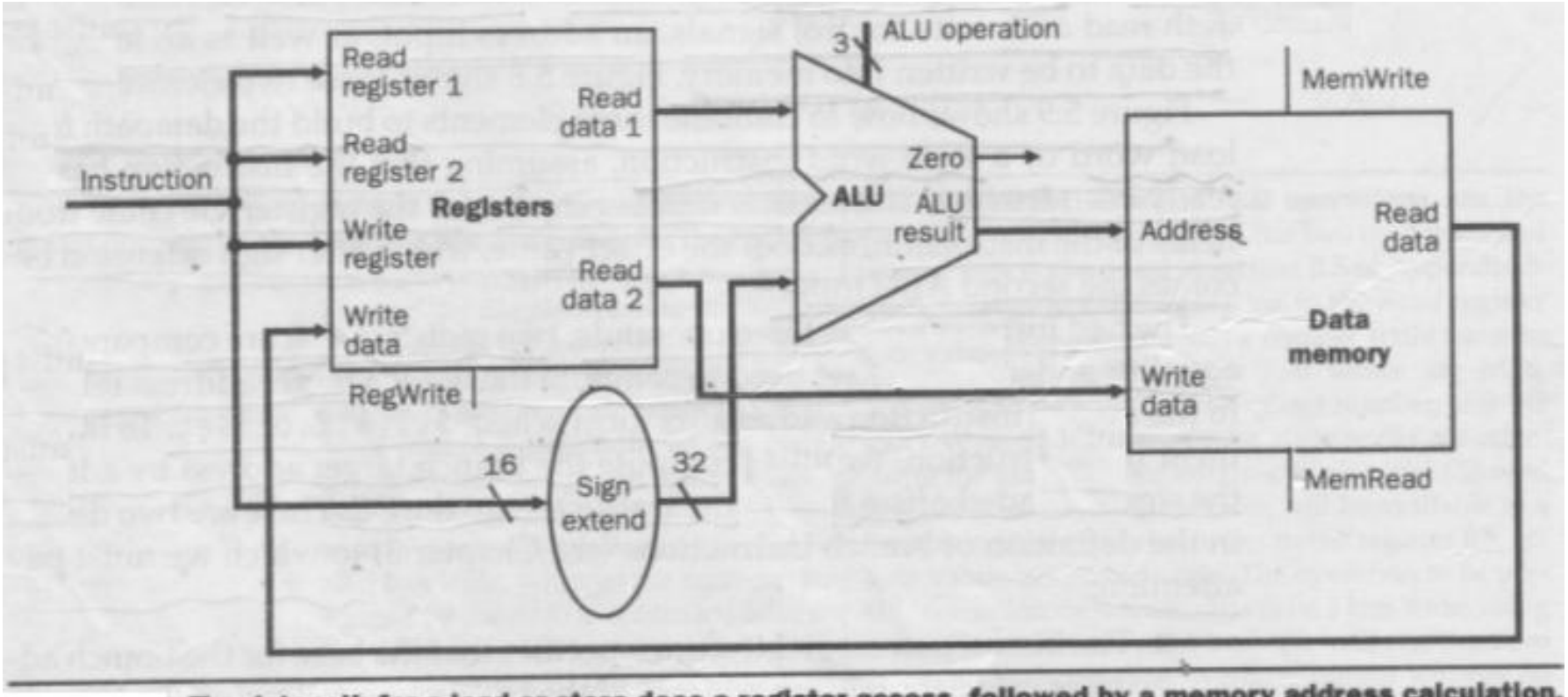
- Load/store word:
- The memory unit is a state element with inputs for the address and write data, having a single output for the read result

- Separate read and write control signal

- The sign-extension unit has a 16-bit input that is sign extended into a 32-bit result appearing on the output

Building a Datapath (Cont.)

- Load word / Store word
- lw $s1, 100 ($s2)



The datapath for a load or store does a register access, followed by a memory address calculation, then a read or write from memory, and a write into the register file if the instruction is a load.

Building a Datapath (Cont.)

- Branch Instructions
- beq $t1, $t2, offset

6 bits	5 bits	5 bits	16 bits
op	rs	rt	address / immediate

- Details of branch instructions:
  - It has three operands , two registers that are compared for equality and a 16 bit offset used to compute the branch target address.
  - Branch target address = sign extended offset field + PC
  - To compute the branch target address the branch datapath includes- 1. a sign extension and 2. adder
  - The offset field is shifted left 2 bits.

Building a Datapath (Cont.)

- Jump
- j 2500 # go to 10000

6 bits	5 bits	5 bits	16 bits
op	rs	rt	address / immediate

- Replace the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits.

Building a Datapath (Cont.)

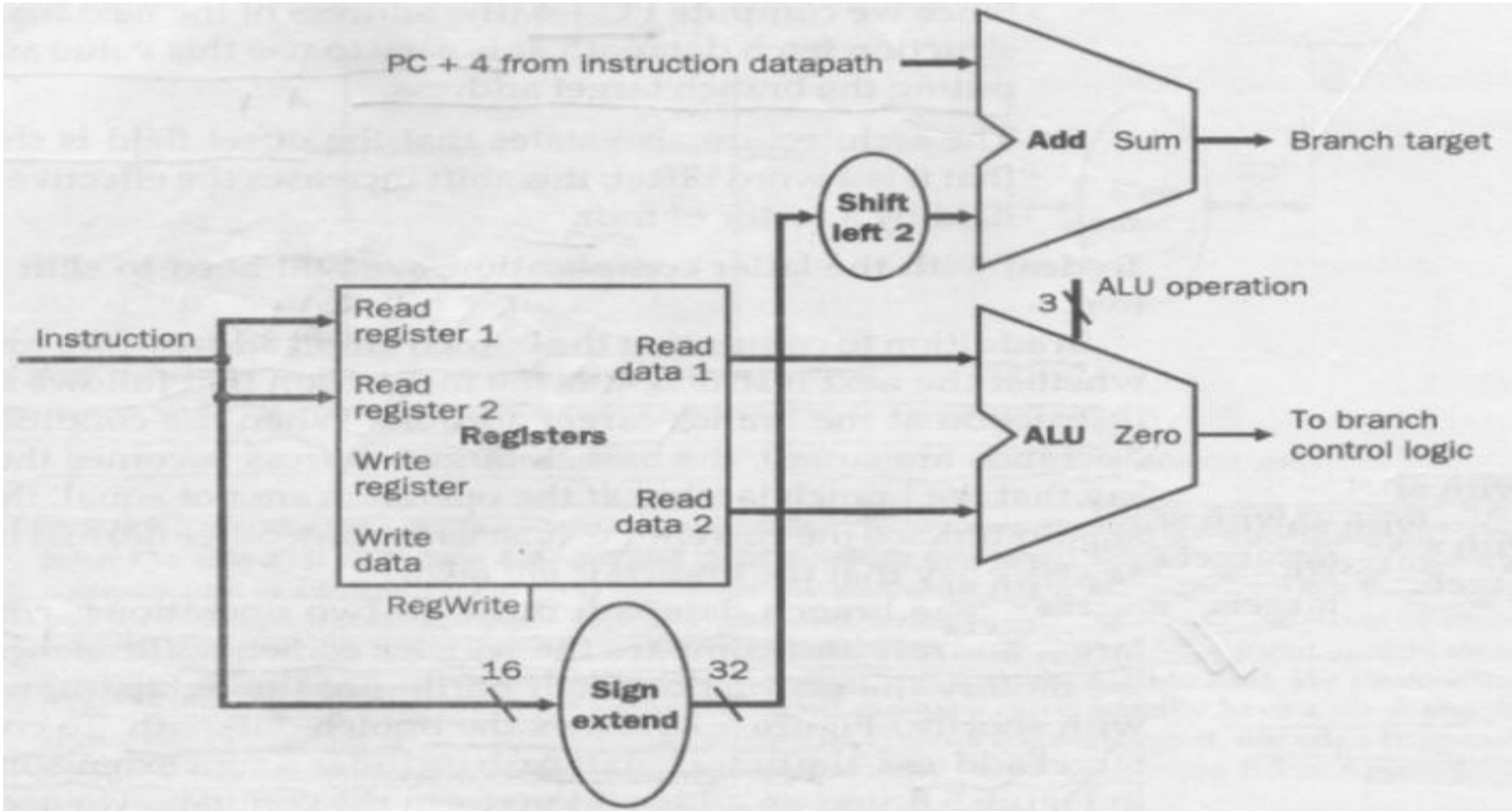


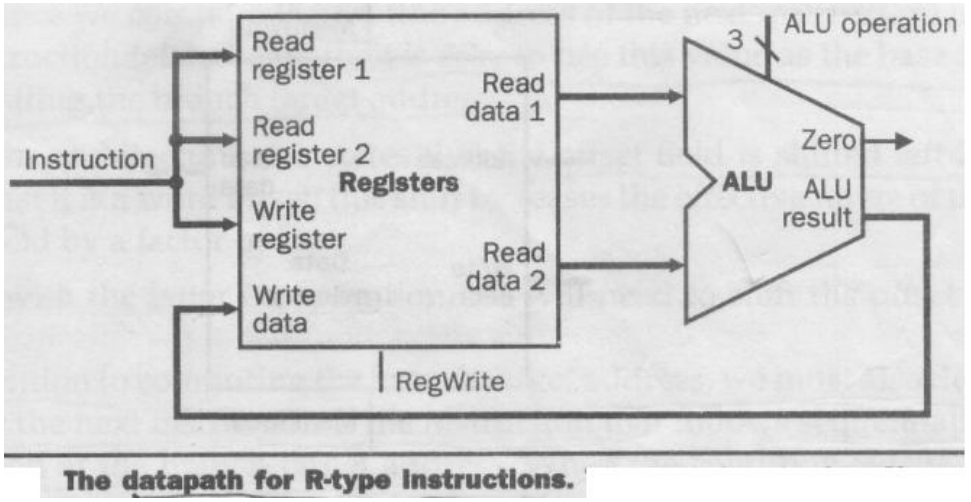
FIGURE 5.10 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.

A Simple Implementation Scheme

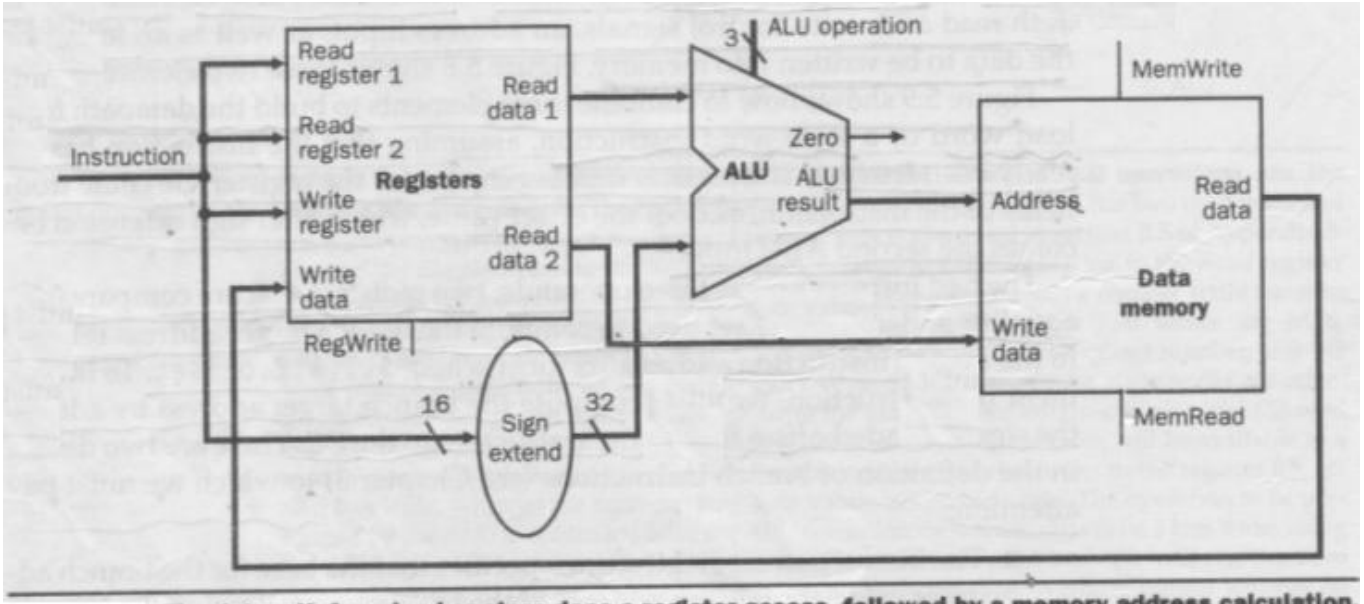
- This covers:
  - lw, sw, beq, add, sub, and, or, slt
- Creating a Single Datapath
  - Single cycle implementation
  - No datapath resource can be used more than once per instruction.
  - So any element needed more than once must be duplicated (instruction memory and data memory are separate)
  - To share a datapath element we need multiplexer / data selector.

A Simple Implementation Scheme

- Combining Datapath:



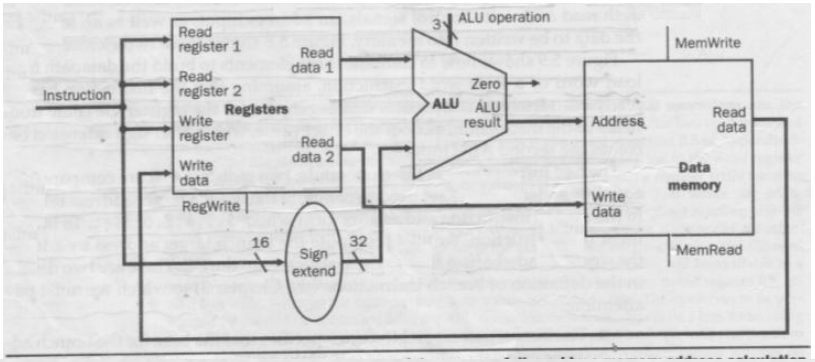
The datapath for R-type instructions.

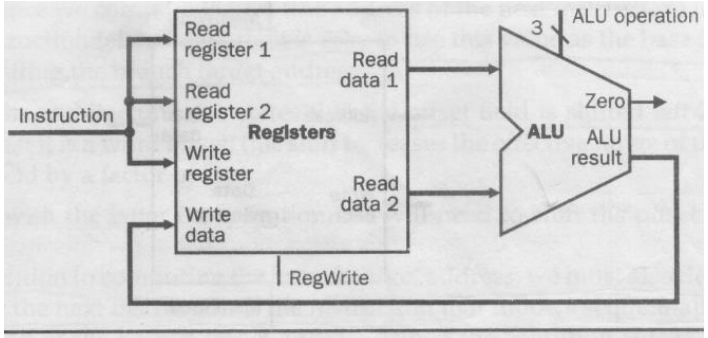


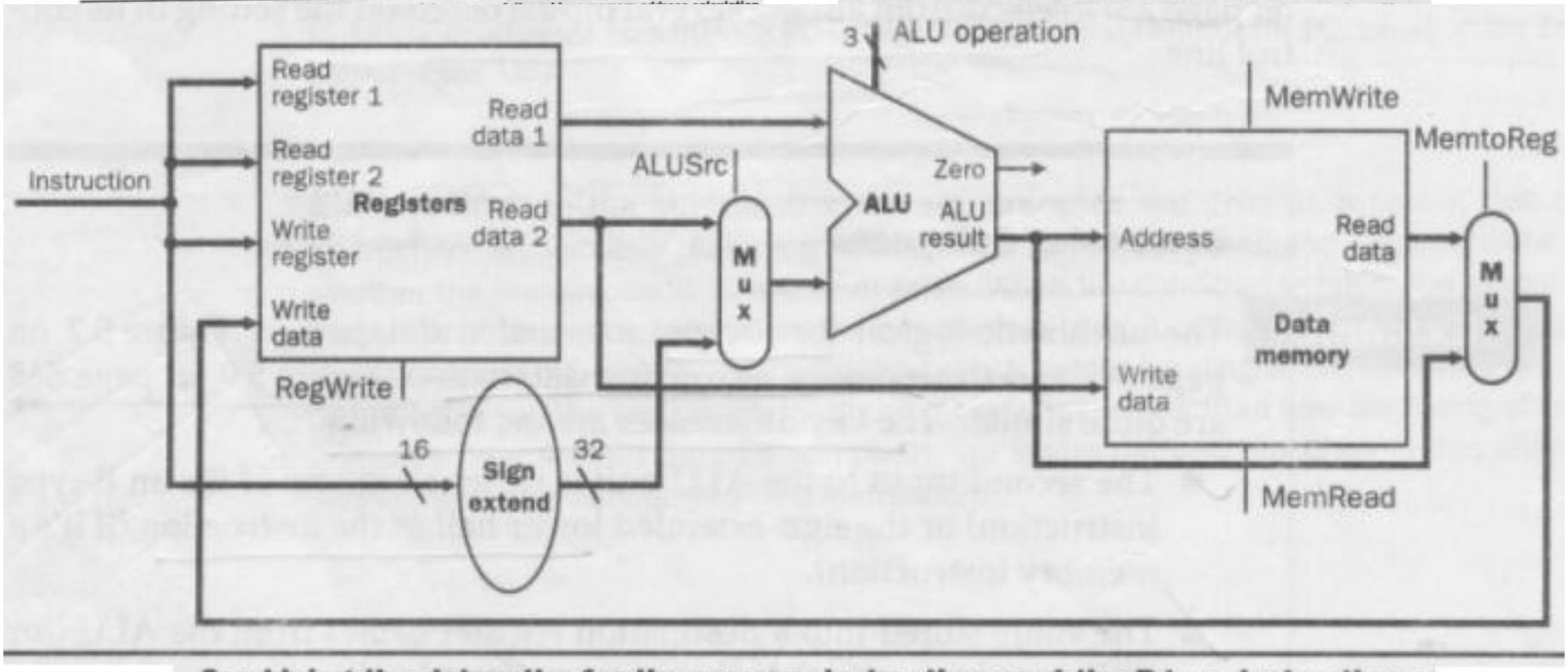
The datapath for a load or store does a register access, followed by a memory address calculation, then a read or write from memory, and a write into the register file if the instruction is a load.

A Simple Implementation Scheme

Datapath for memory instructions and R-type instructions

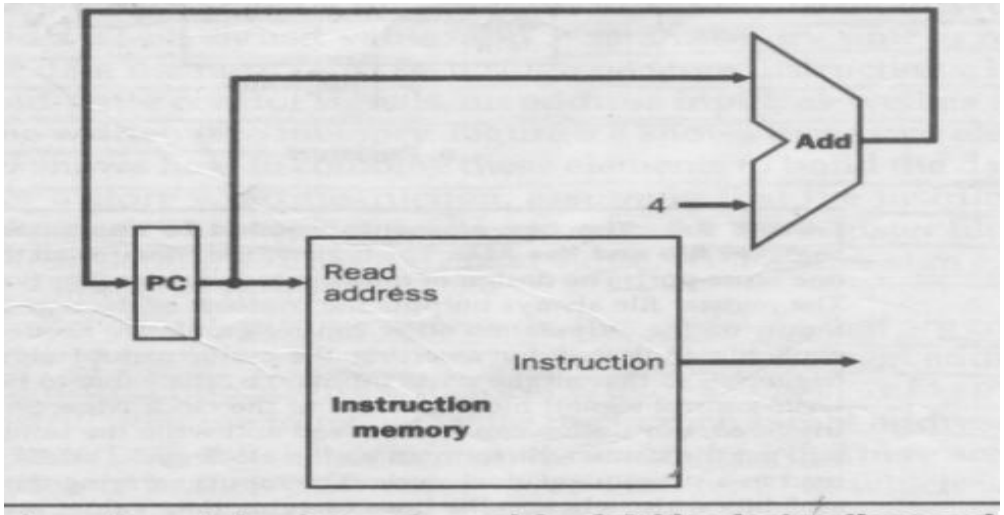




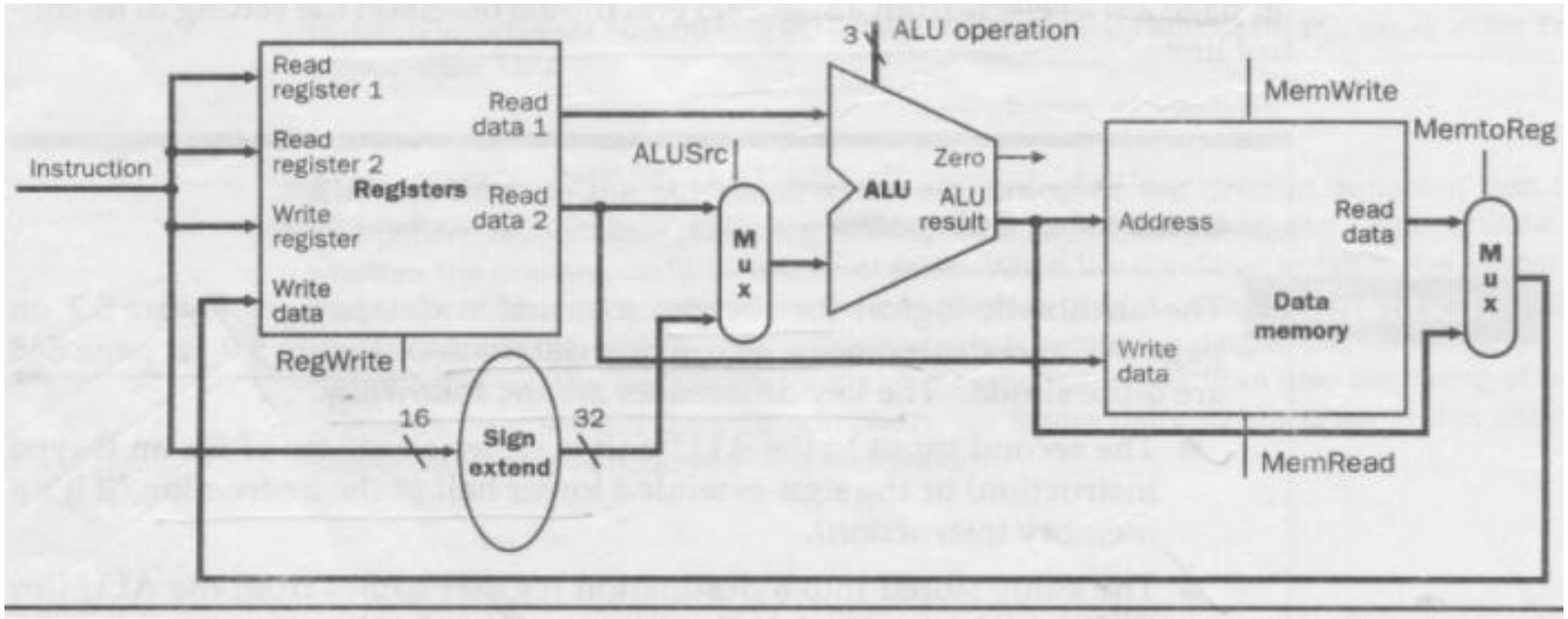


Combining the datapaths for the memory instructions and the R-type instructions.

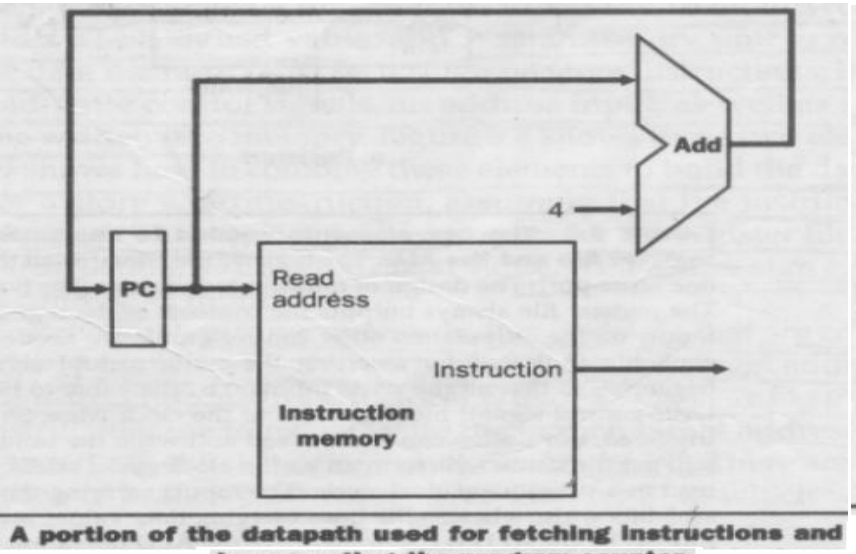
A Simple Implementation Scheme



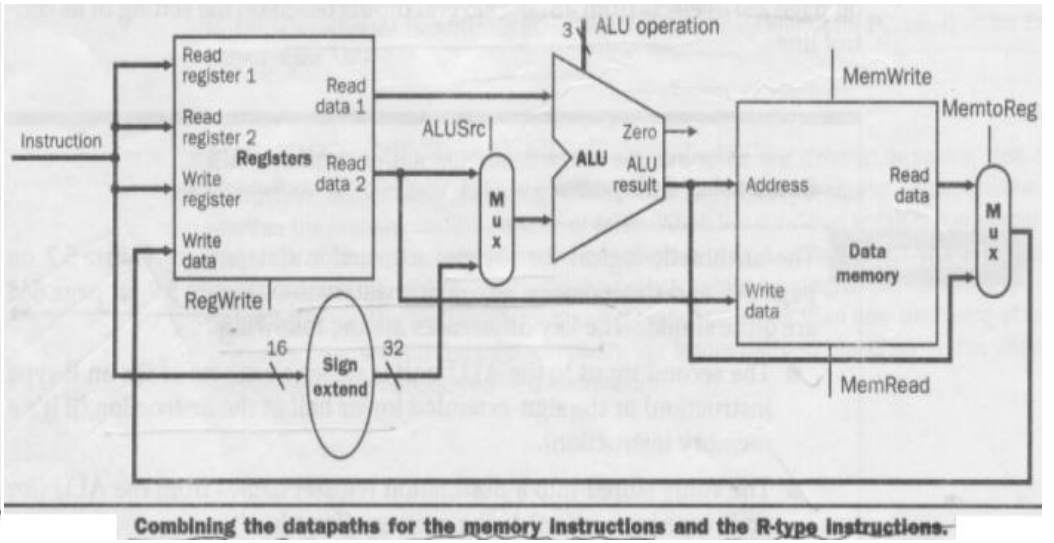
A portion of the datapath used for fetching instructions and incrementing the program counter.



Combining the datapaths for the memory instructions and the R-type instructions.



A portion of the datapath used for fetching instructions and incrementing the program counter.



Combining the datapaths for the memory instructions and the R-type instructions.

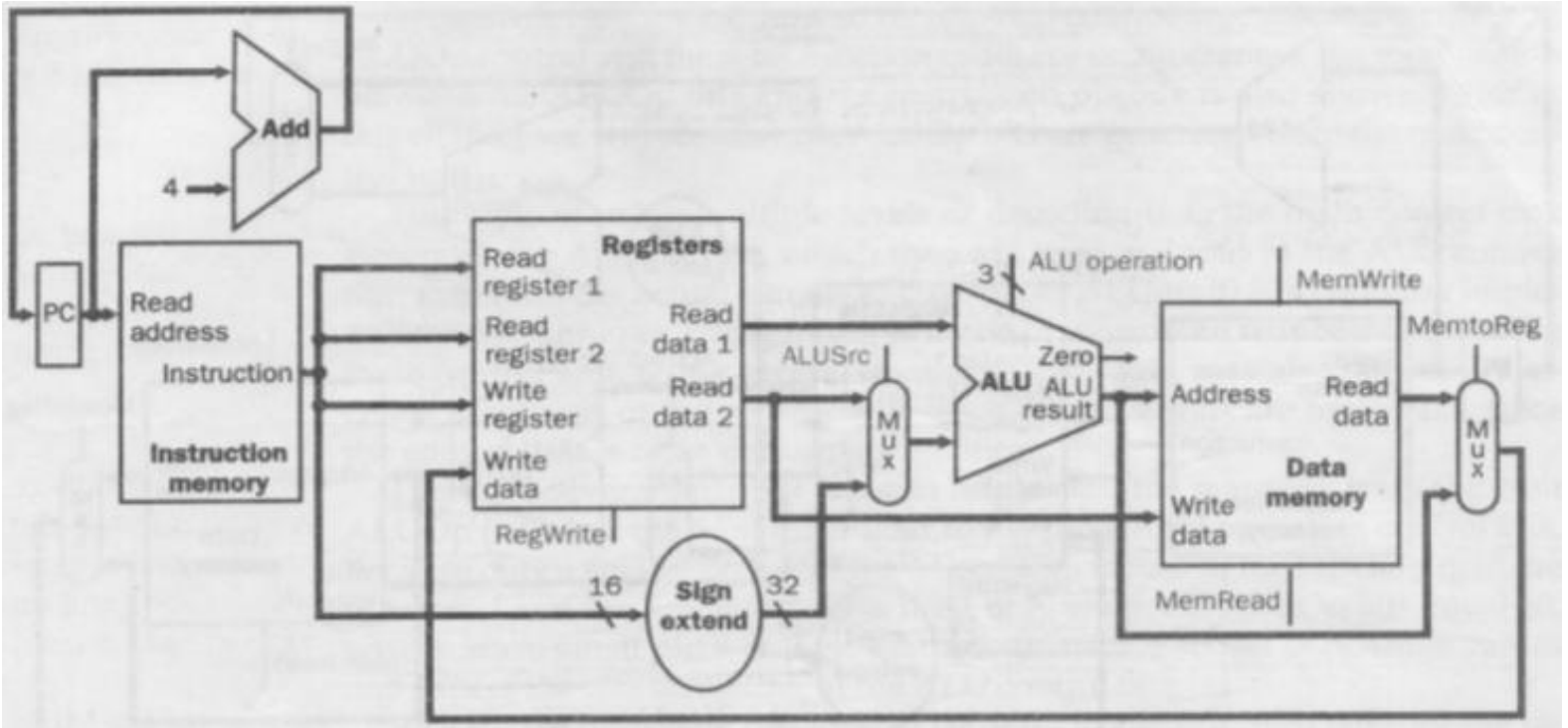


FIGURE 5.12 The instruction fetch portion of the datapath from Figure 5.5 is appended to the datapath of Figure 5.11 that handles memory and ALU instructions.

A Simple Implementation Scheme

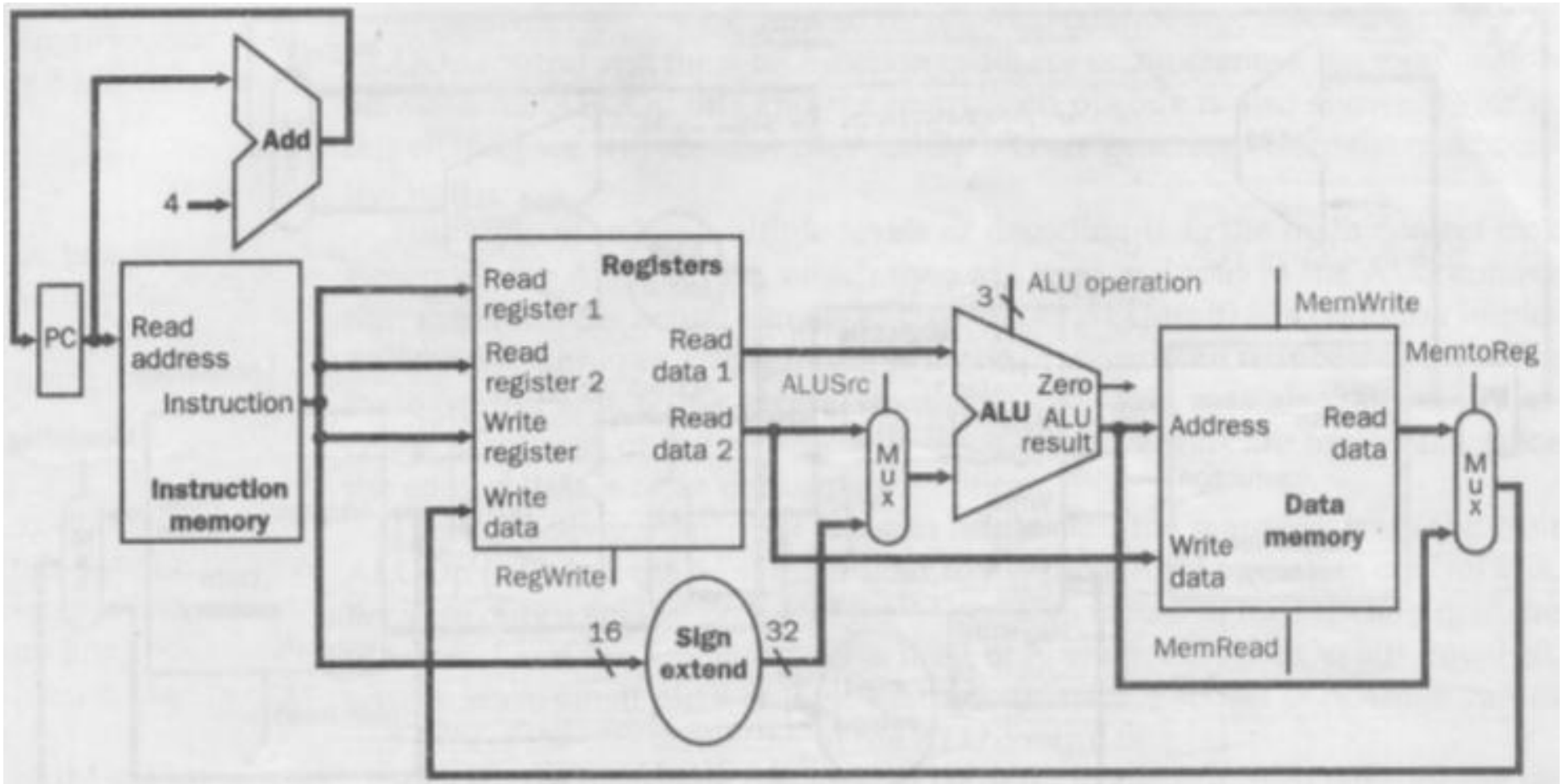


FIGURE 5.12 The instruction fetch portion of the datapath from Figure 5.5 is appended to the datapath of Figure 5.11 that handles memory and ALU instructions.

A Simple Implementation Scheme

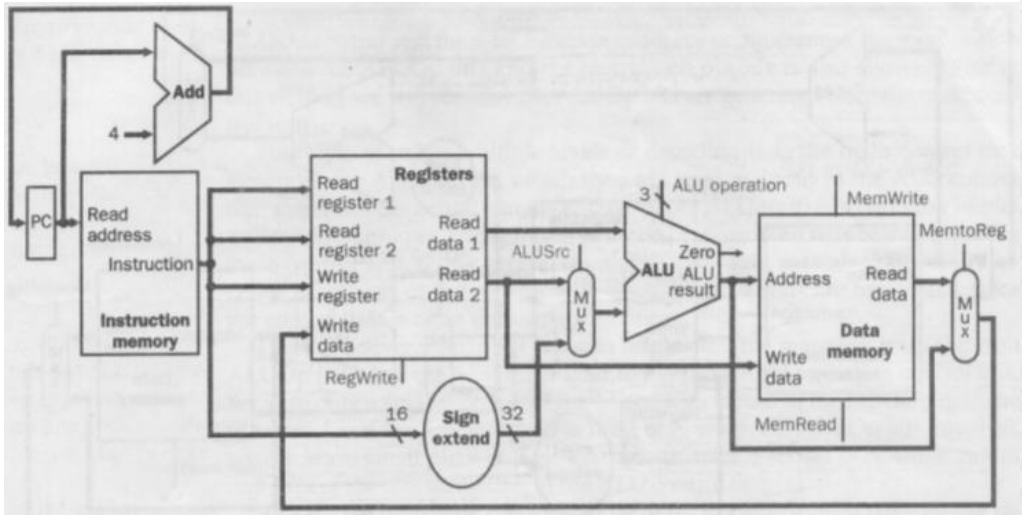


FIGURE 5.12 The instruction fetch portion of the datapath from Figure 5.5 is appended to the datapath of Figure 5.11 that handles memory and ALU instructions.

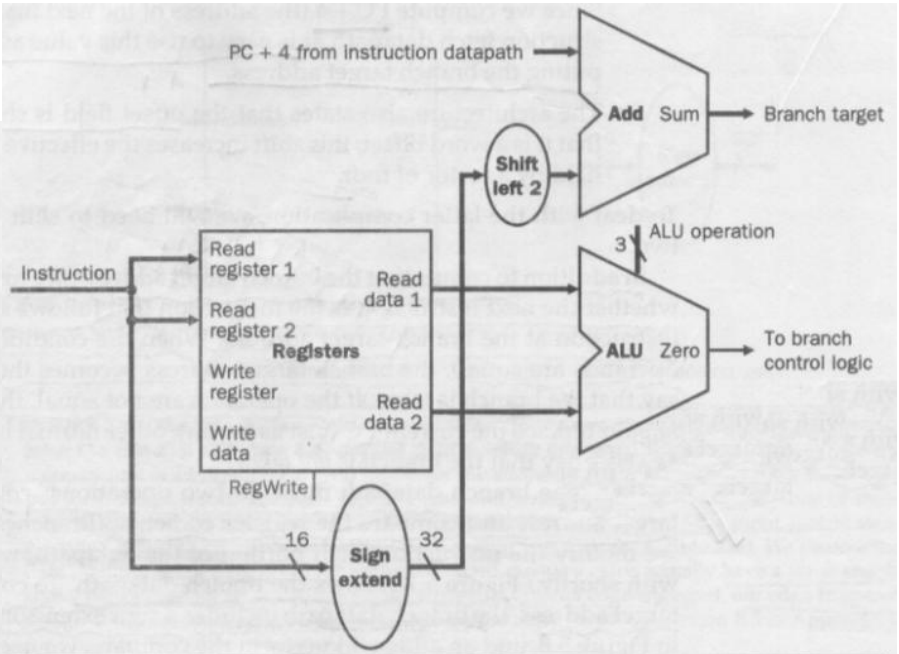


FIGURE 5.10 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.

A Simple Implementation Scheme

ALU operations, load/store word and branches

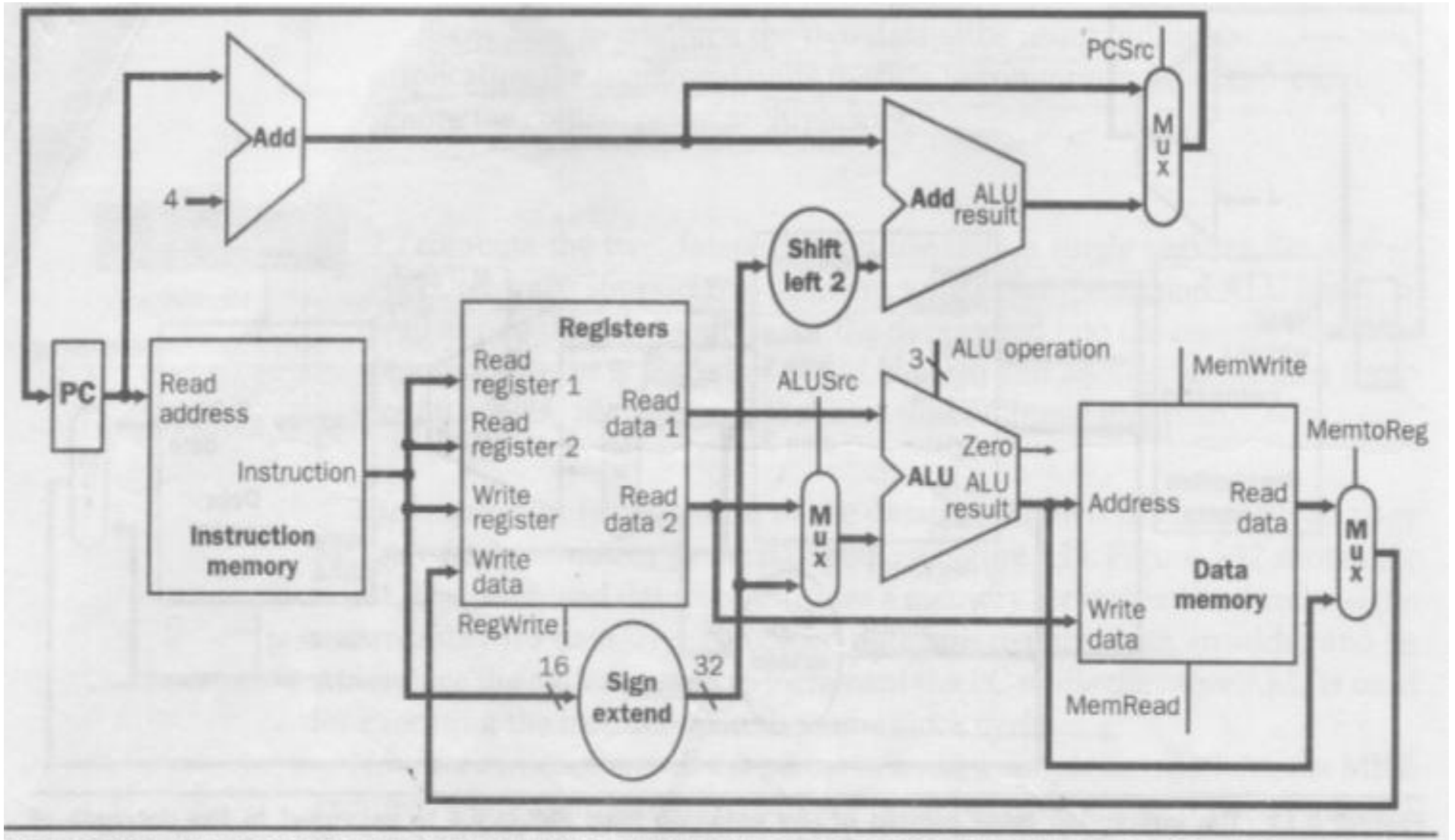


FIGURE 5.13 The simple datapath for the MIPS architecture combines the elements required by different instruction classes.

A Simple Implementation Scheme (Cont.)

- The ALU Control
- Depending on the instruction class, the ALU will need to perform one of these five functions.

ALU control input	Function
000	AND
001	OR
010	add
110	subtract
111	set on less than

lw, sw : compute the memory address by adding

R-type : add, sub, and, or, slt

Branch : ALU must perform a subtraction

A Simple Implementation Scheme (Cont.)

- The ALU Control

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

FIGURE 5.14 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction.

A Simple Implementation Scheme (Cont.)

- **The ALU Control (Truth table)**
- Only the entries for which the ALU control is asserted are shown.
- ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1 rather than 10 and 01
- When function field is used the first two bits (F5 and F4) are always 10 so they are don't care terms.

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

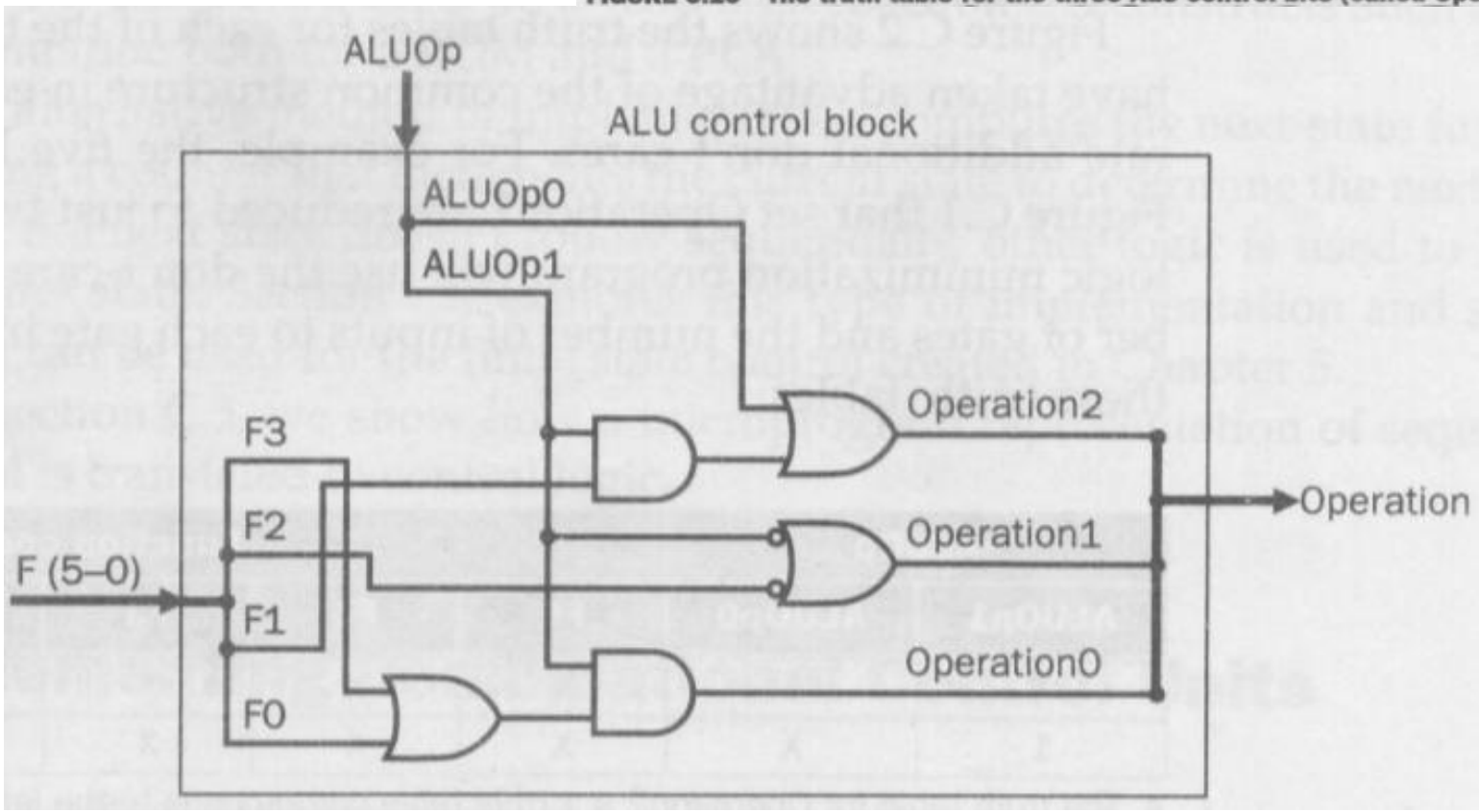
FIGURE 5.15 The truth table for the three ALU control bits (called Operation).

A Simple Implementation Scheme (Cont.)

- The ALU Control

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

FIGURE 5.15 The truth table for the three ALU control bits (called Operation).



A Simple Implementation Scheme (Cont.)

- **Designing the Main Control Unit**
- R-type

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

- lw, sw

31-26	25-21	20-16	15-0
35 or 43	rs	rt	address

- Branch

31-26	25-21	20-16	15-0
4	rs	rt	address

A Simple Implementation Scheme (Cont.)

- Designing the Main Control Unit
- Observations:
  - op field always contained in bits 31-26.
  - Two registers to be read are always specified at positions 25-21 and 20-16. (R-type, beq, sw)
  - Base register for lw and sw is always in 25-21
  - 16-bit offset (beq, lw, sw) are always in 15-0
  - Destination register is in 20-16 (lw) or 15-11 (R-type).

- Designing the Main Control Unit
- The different positions for the two destination registers implies a selector (i.e., a mux) to locate the appropriate field for each type of instruction.

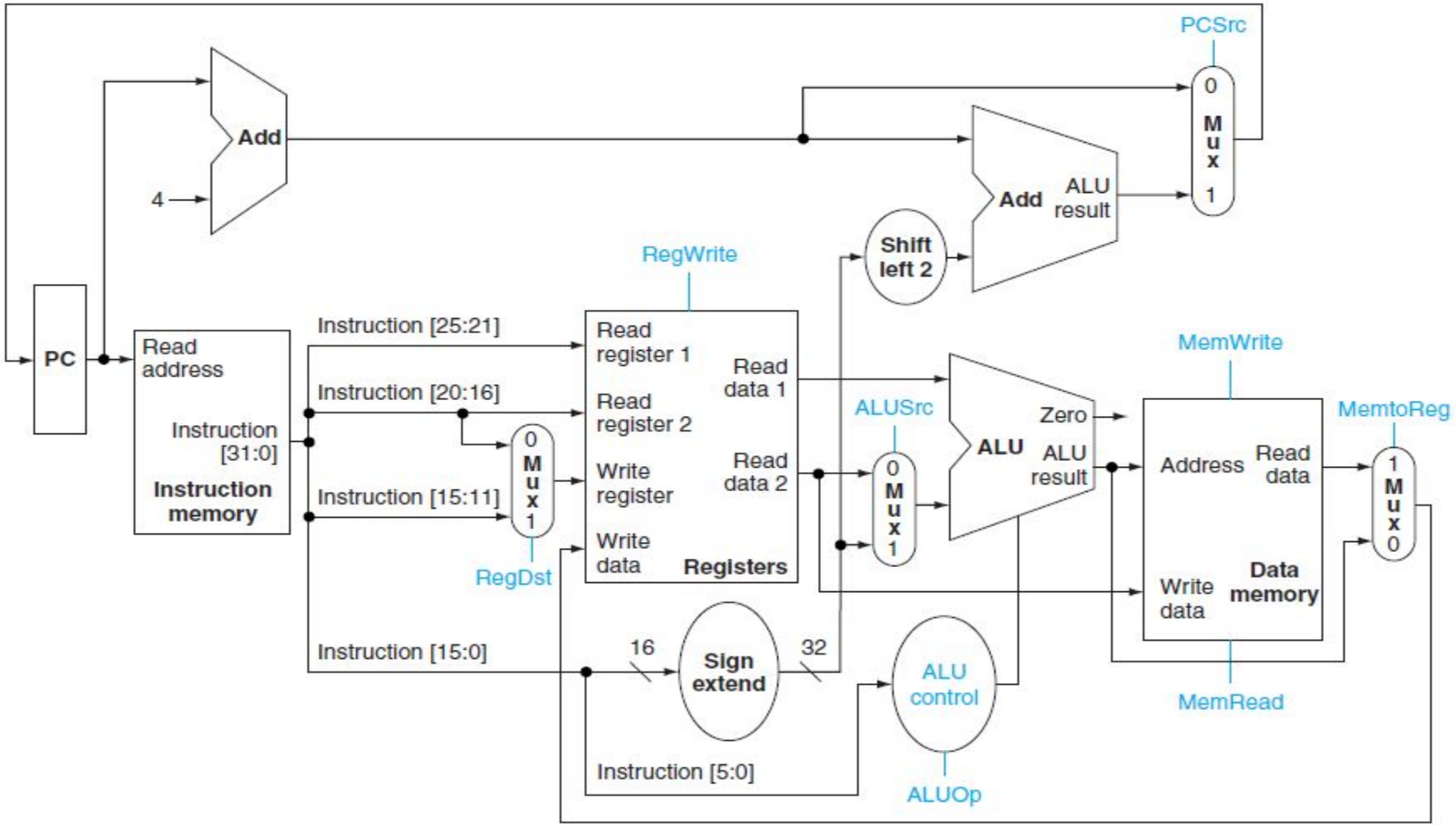


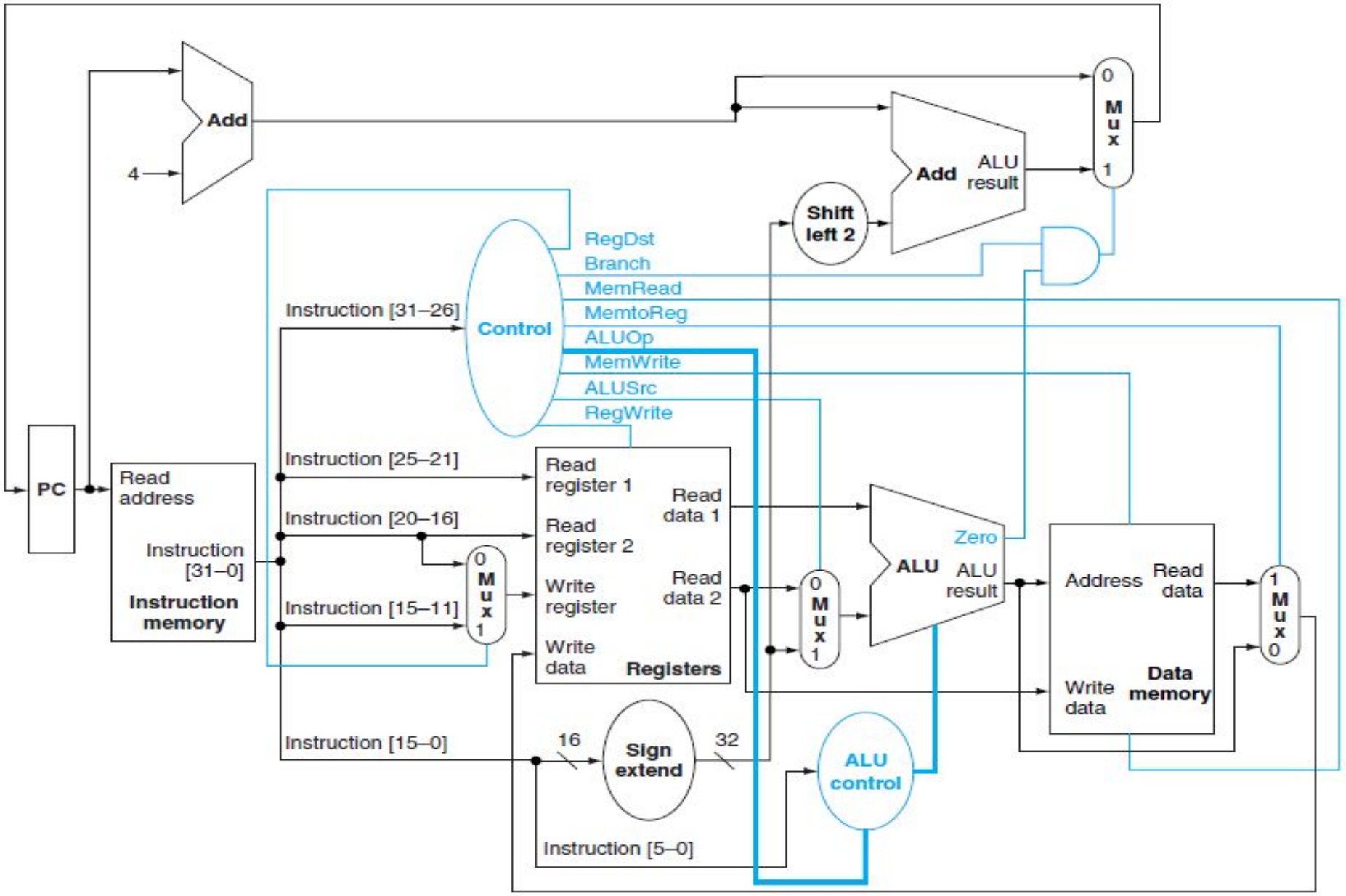
FIGURE 5.15 The datapath of Figure 5.12 with all necessary multiplexors and all control lines identified. The control lines are

A Simple Implementation Scheme (Cont.)

- The effect of each of the seven control lines

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

- The simple datapath with the control unit



Designing the Main Control Unit

FIGURE 5.17 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.

Designing the Main Control Unit

- All control signals can be set based on the opcode bits.

- Given only the opcode, the control unit can thus set all the control signals except PCSrc, which is only set if the instruction is beq and the Zero output of the ALU used for comparison is true.
- PCSrc is generated by **AND**-ing a Branch signal from the control unit with the Zero signal from the ALU.

A Simple Implementation Scheme (Cont.)

- **Operations of the Datapath**

- R-type instruction (add $t1, $t2, $t3)

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

- Operations of the Datapath R-type instruction (add $t1, $t2, $t3)

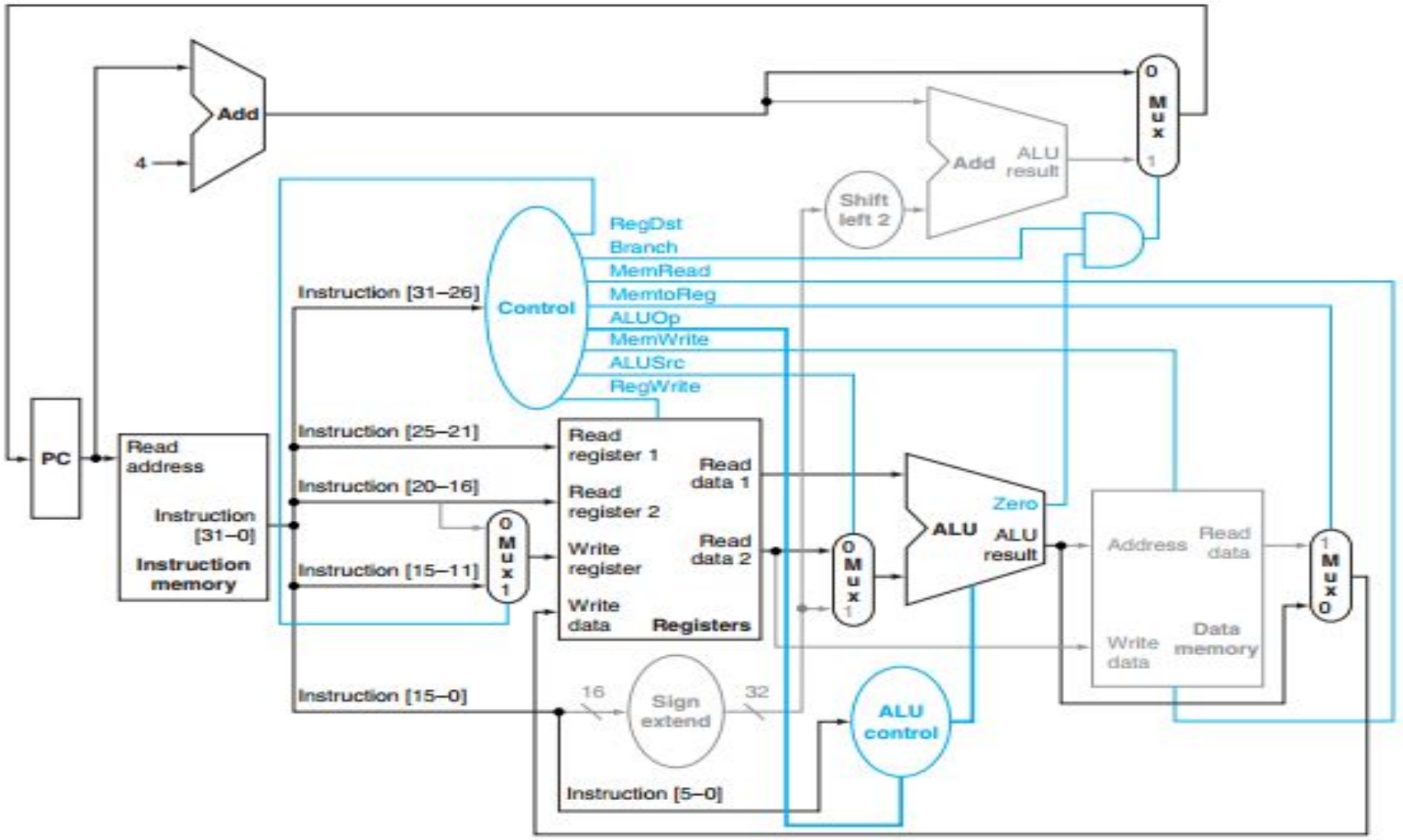


FIGURE 5.19 **The datapath in operation for an R-type instruction such as add** `$t1,$t2,$t3`. The control lines, datapath units, and connections that are active are highlighted.

A Simple Implementation Scheme (Cont.)

- Operations of the Datapath

- lw $t1, offset($t2) instruction
  - Instruction is fetched
  - $t2 value is read
  - ALU operates
  - Sum of the ALU is used as the address for the data memory
  - Data from the memory is written into the register

- Operations of the Datapath lw $t1, offset($t2) instruction

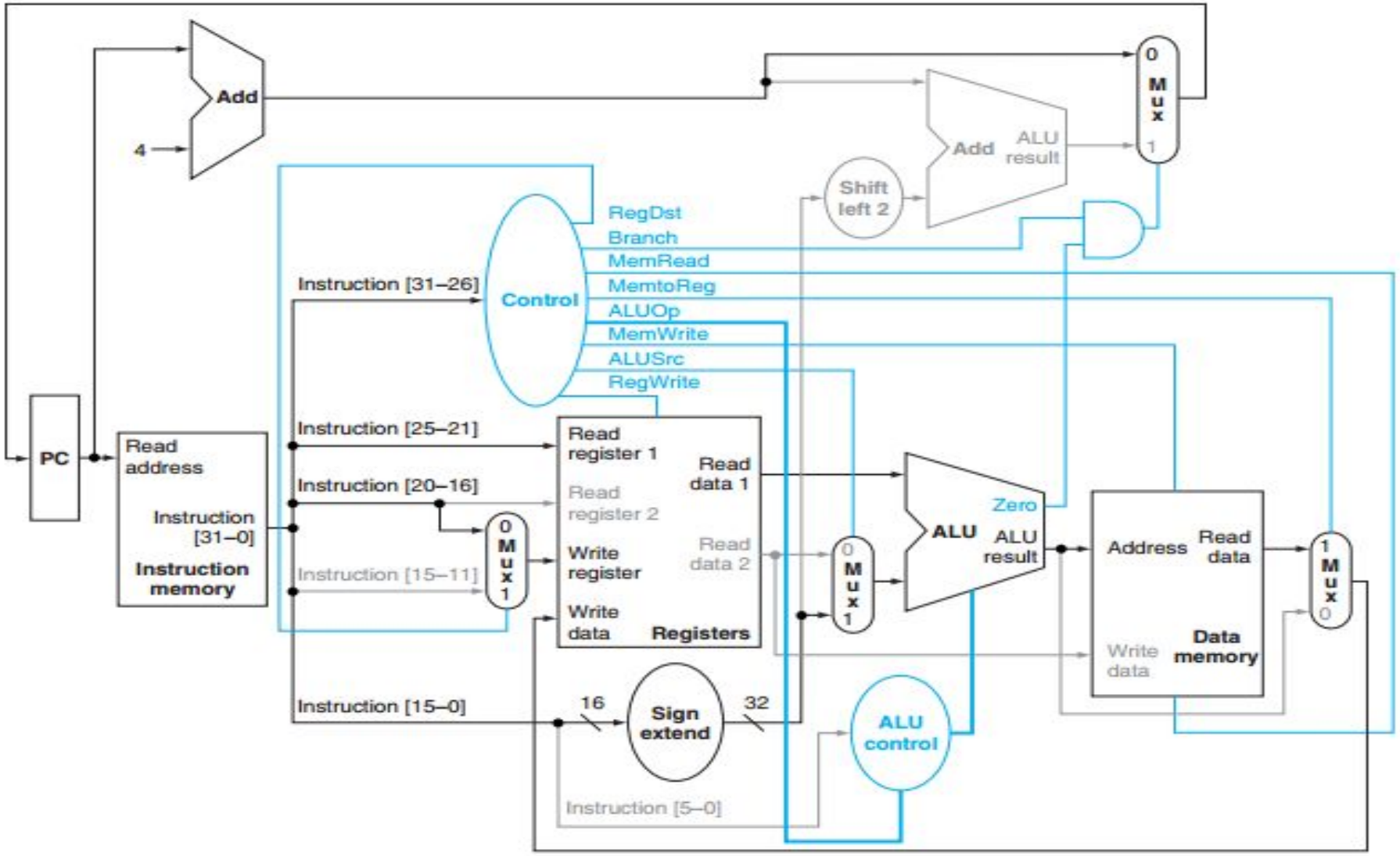


FIGURE 5.20 The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are high-

A Simple Implementation Scheme (Cont.)

- Operations of the Datapath

- beq $t1, $t2, offset instruction

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, `$t1` and `$t2`, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (`offset`) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

- Operations of the Datapath beq $t1, $t2, offset instruction

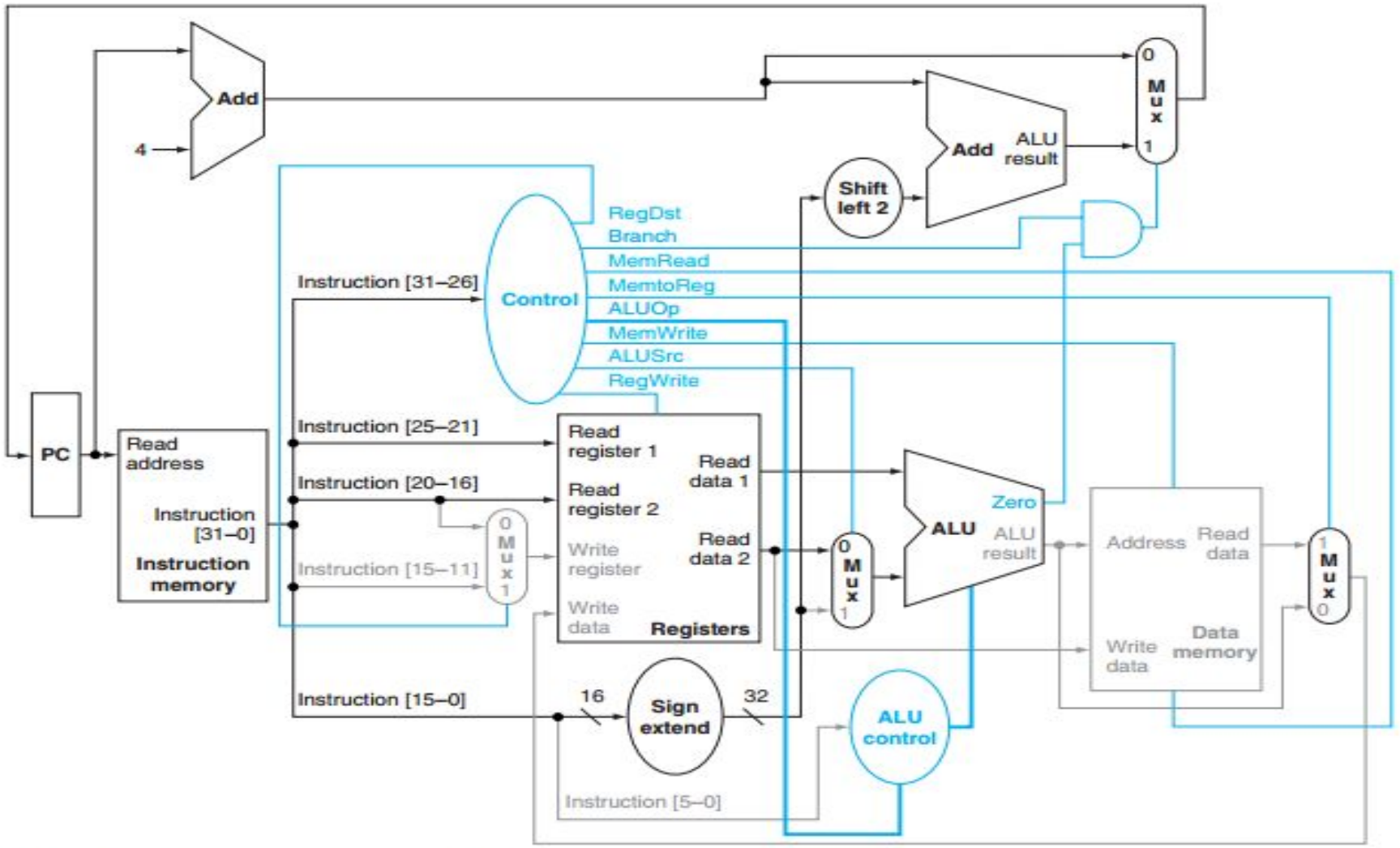


FIGURE 5.21 The datapath in operation for a branch equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates

A Simple Implementation Scheme (Cont.)

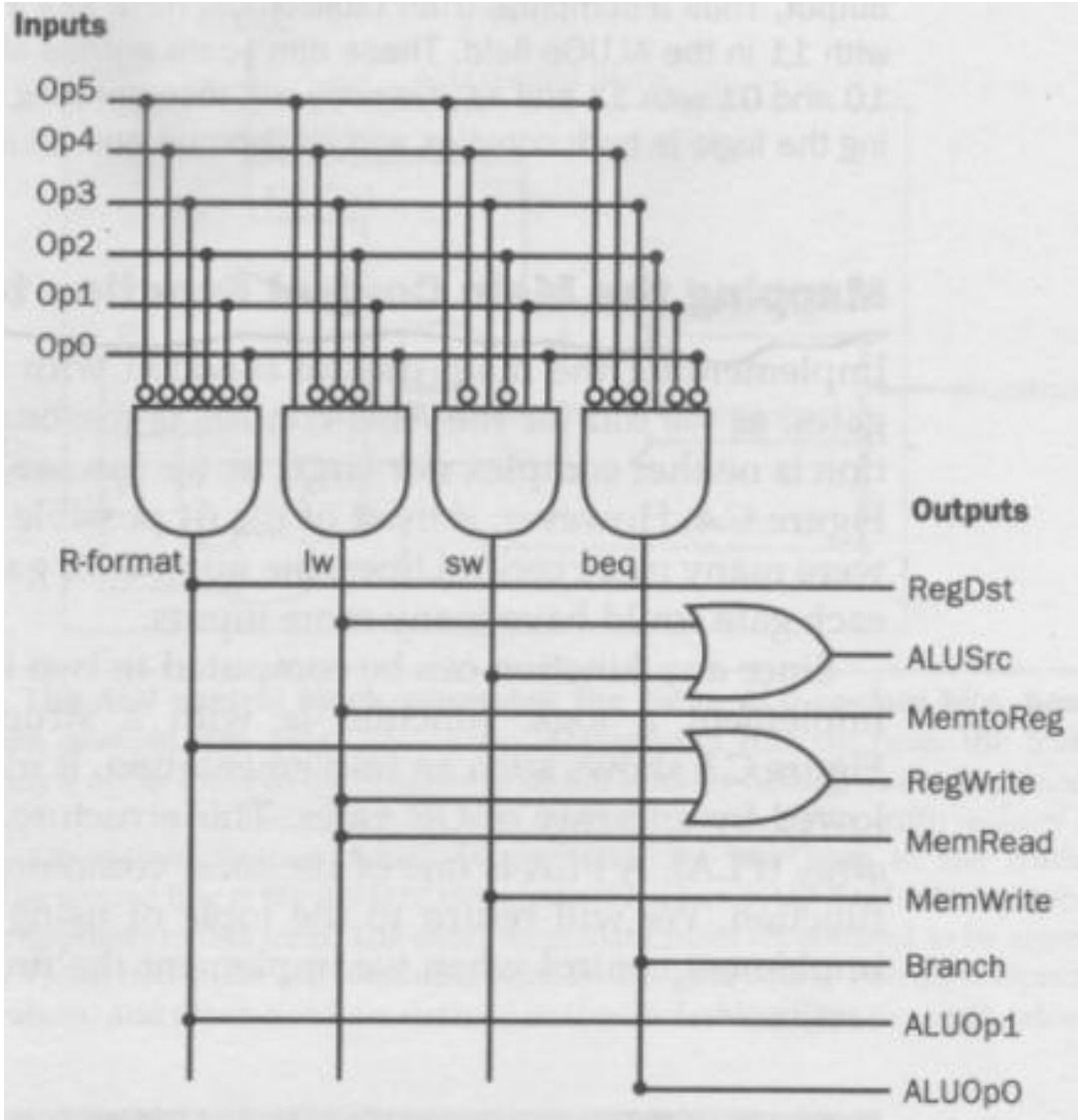
- Finalizing the Control

Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE C.4 The control function for the simple one-clock implementation is completely specified by this truth table.

A Simple Implementation Scheme (Cont.)

- Finalizing the Control



A Simple Implementation Scheme (Cont.)

- Performance of single cycle machines
- Example (P – 373) (SELF)
  - Memory units: 2 ns
  - ALU & Adder: 2 ns
  - Register file: 1 ns
  - Other units have no delay
- Which implementation is faster?
  - Every instruction operates in 1 clock cycle of fixed length.
  - Every instruction executes in 1 clock cycle of variable length.

- **Why a Single-Cycle Implementation is not used?**

- It is inefficient
- Clock cycle must have the same length for every instruction.
- CPI will be 1, therefore, the overall performance of a single cycle implementation is not likely to be very good

A Multicycle Implementation

- Each step in the execution will take 1 clock cycle.
- Multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles.
- This sharing can help reduce the amount of hardware required.

A Multicycle Implementation

- **Advantages:**

- The ability to allow instructions to take different numbers of clock cycles.

- The ability to share functional units within the execution of a single instruction.

A Multicycle Implementation

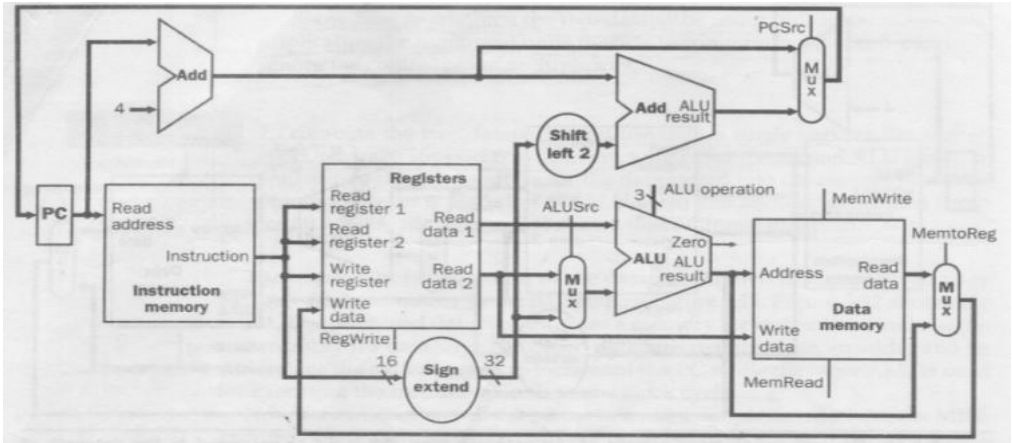


FIGURE 5.13 The simple datapath for the MIPS architecture combines the elements required by different instruction classes.

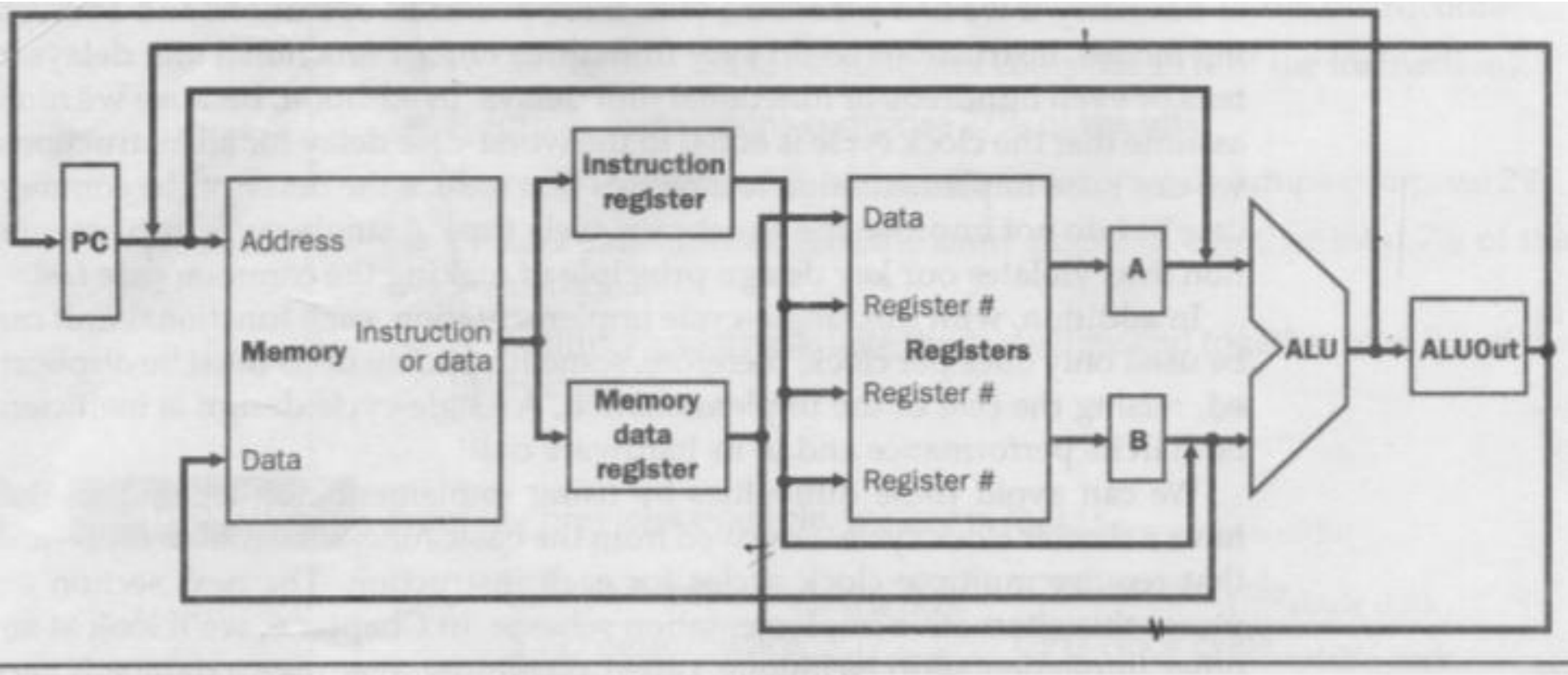


FIGURE 5.30 The high-level view of the multicycle datapath.

Differences between a single-cycle and multi-cycle datapath

- In the multicycle datapath, one memory unit stores both instructions and data, whereas the single-cycle datapath requires separate instruction and data memories.

- The multicycle datapath uses on ALU, versus an ALU and two adders in the single-cycle datapath.

- In the single-cycle implementation, the instruction executes in one cycle (by design) and the outputs of all functional units must stabilize within one cycle. In contrast, the multicycle implementation uses one or more registers to temporarily store (buffer) the ALU or functional unit outputs. This *buffering* action stores a value in a temporary register until it is needed or used in a subsequent clock cycle.

A Multicycle datapath for basic instructions

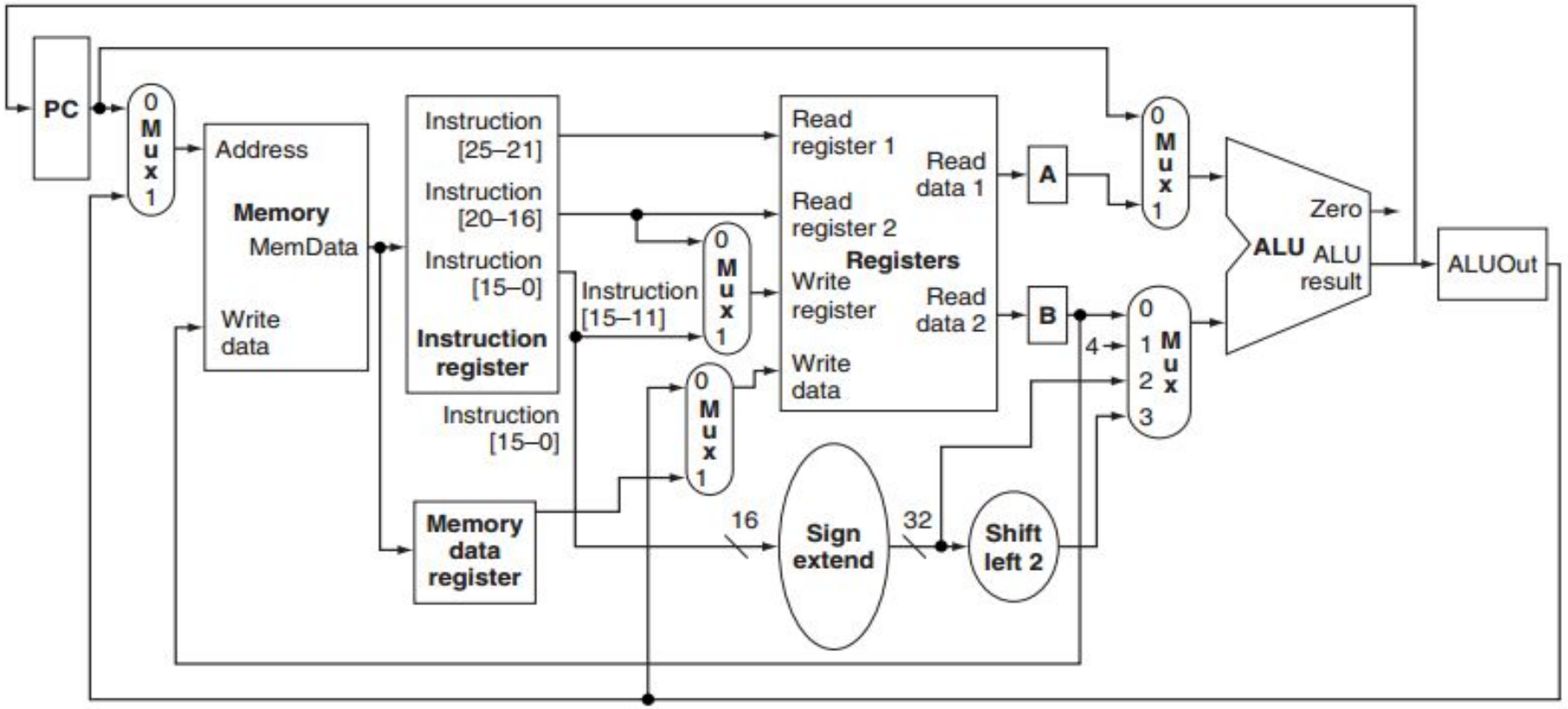


FIGURE 5.26 Multicycle datapath for MIPS handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

A Multicycle Implementation

- **New Registers:**

The following temporary registers are important to the multicycle datapath implementation discussed in this section:

- *Instruction Register* (IR) saves the data output of memory for a subsequent instruction read
- *Memory Data Register* (MDR) saves memory output for a data read operation;
- *A and B Registers* (A,B) store ALU operand values read from the register file and
- *ALU Output Register* (ALUout) contains the result produced by the ALU.

• New Muxes.

- we also need to add new multiplexers and expand existing ones, to implement sharing of functional units. For example, we need to select between memory address as PC (for a load instruction) or ALUout (for load/store instructions). The muxes also route to one ALU the many inputs and outputs that were distributed among the several ALUs of the single-cycle datapath. Thus, we make the following additional changes to the single-cycle datapath:

- Add a multiplexer to the first ALU input, to choose between **(a)** the A register as input (for R- and I-format instructions) , or **(b)** the PC as input (for branch instructions).

- On the second ALU, the input is selected by a ***four-way mux*** (two control bits). The two additional inputs to the mux are **(a) the immediate (constant) value 4 for incrementing the PC** and **(b) the sign-extended offset, shifted two bits to preserve alignment, which is used in computing the branch target address**.

A Multicycle datapath with control lines

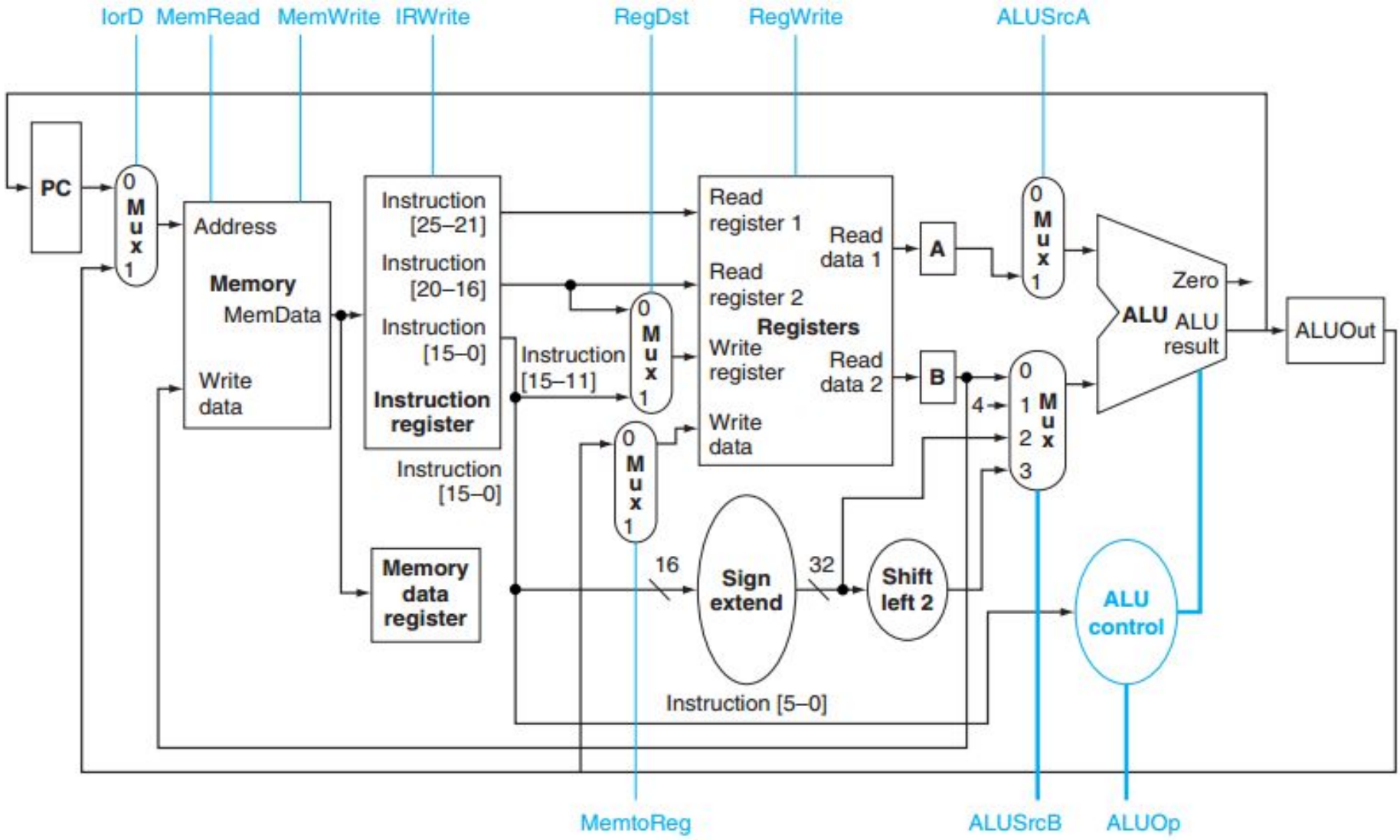


FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit

- **New Control Signals.**

- *Write Control Signals* for the IR and programmer-visible state units
- *Read Control Signal* for the memory; and
- *Control Lines* for the muxes.

FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

- **Branch and Jump Instruction Support.**

- To implement branch and jump instructions, one of three possible values is written to the PC:
- *ALU output* = PC + 4, to get the next instruction during the instruction fetch step (to do this, PC + 4 is written directly to the PC)
- *Register ALUout*, which stores the computed branch target address.
- *Lower 26 bits (offset) of the IR*, shifted left by two bits (to preserve alginment) and concatenated with the upper four bits of PC+4, to form the jump target address.

The complete datapath for the multicycle implementation for the necessary control lines

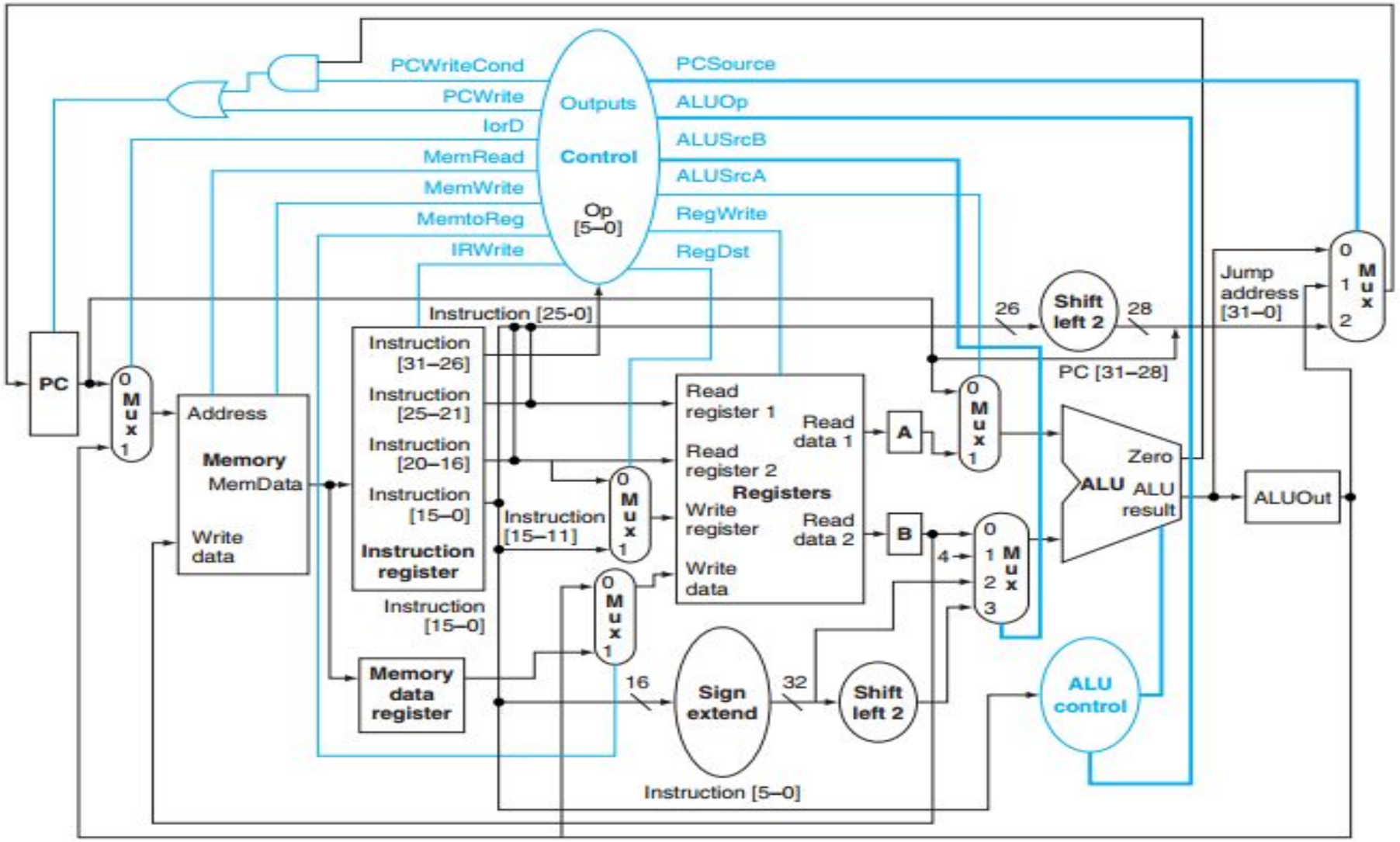


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The con-

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

Microprogramming: Simplifying Control Design

- Each *microinstruction* defines the set of datapath control signals that must be asserted in a given state.
- Designing the control as a program that implements the machine instructions in terms of simpler microinstructions is called *microprogramming*.

Microprogramming:

Implementing the Microprogram

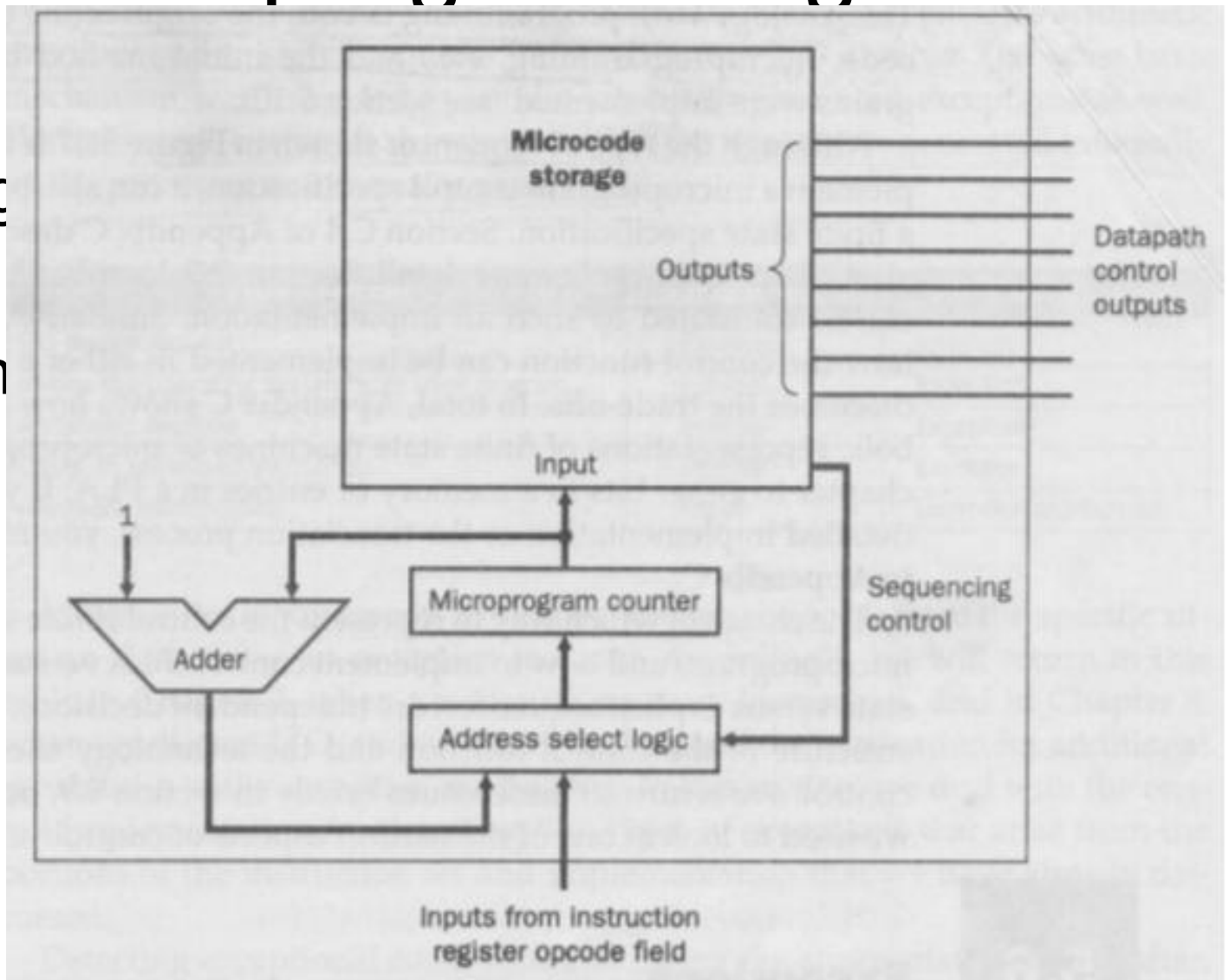


FIGURE 5.47 A typical implementation of a microcode controller would use an explicit incrementer to compute the default sequential next state and would place the microcode in a read-only memory.

Implementing the Microprogram

- The microinstruction ,used to set the datapath control, are assembled directly from the microprogram,
- The microprogram counter determines how the next instruction is chosen
- The address select logic contains dispatch tables as well as the logic to select among the alternative next states.
- The selection of next microinstruction is controlled by the sequencing control.
- The microcode storage may either a ROM or may be implemented by a PLA

Exceptions (self)

- An **exception** is an unexpected event within the processor.
  - Example: arithmetic overflow
- An **interrupt** is an event that also causes an unexpected change in control flow but comes from outside of the processor.
  - Example: I/O device request

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Exceptions

- How Exceptions Are Handled?-from book

- Two types of exceptions –
  - Execution of an undefined instruction
  - An arithmetic overflow
- Basic Actions
  - Save the address of the offending instruction in the exception program counter (EPC)
  - Transfer control to the OS at some specified address

Exceptions

- How Control Checks(detect) for Exceptions

- Undefined instruction
  - This is detected when no next state is defined from state-1 for the op value.
- Arithmetic overflow
  - A signal Overflow is provided as an output from the ALU.