The Processor: Datapath and Control

Rui Wang, Assistant professor
Dept. of Information and Communication
Tongji University

Email: ruiwang@tongji.edu.cn

- Building a datapath
- Simple implementation scheme
- Multicycle implementation
- Exceptions

A Basic MIPS Implementation

- Implementation of a MIPS subset
  - The memory-reference instructions load word (lw) and store word (sw)
  - The arithmetic-logical instructions add, sub, and, or, and slt
  - The instructions branch equal (beq) and jump (j), which we add last

Implementation overview

- For each instruction processing, two steps are the same
  - Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory
  - Read one or two registers, using fields of the instruction to select the registers to read. Lw (one register), most other instructions (two registers)

Instruction processing

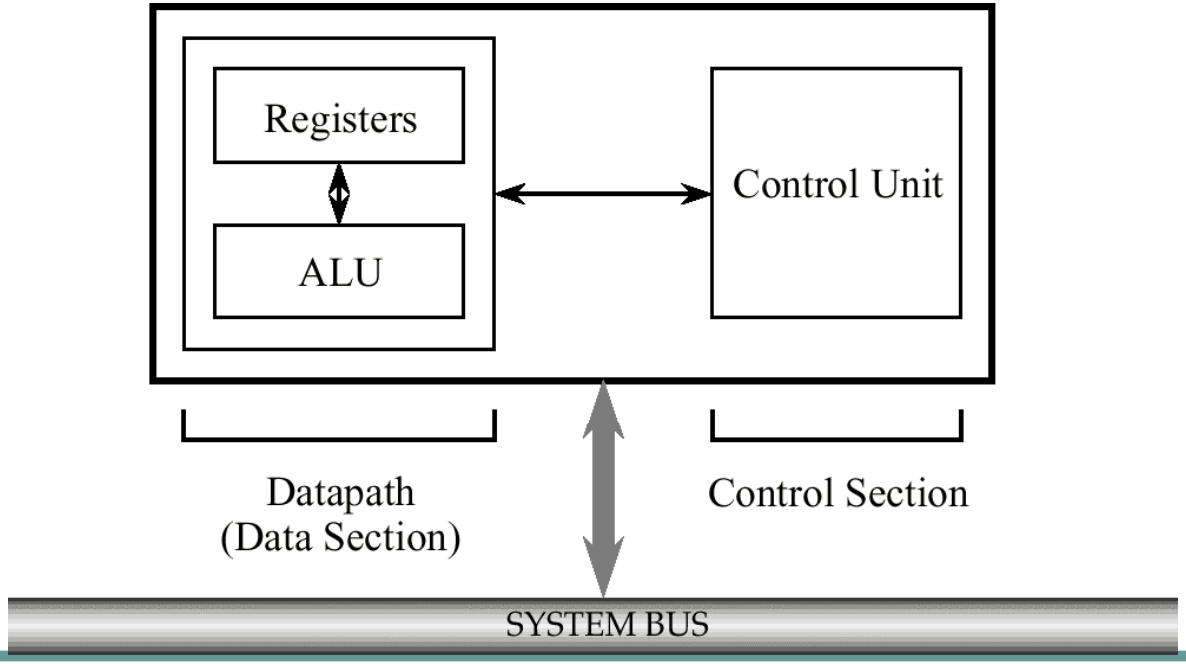
The Fetch-Execute Cycle

The steps that the control unit carries out in executing a program are:

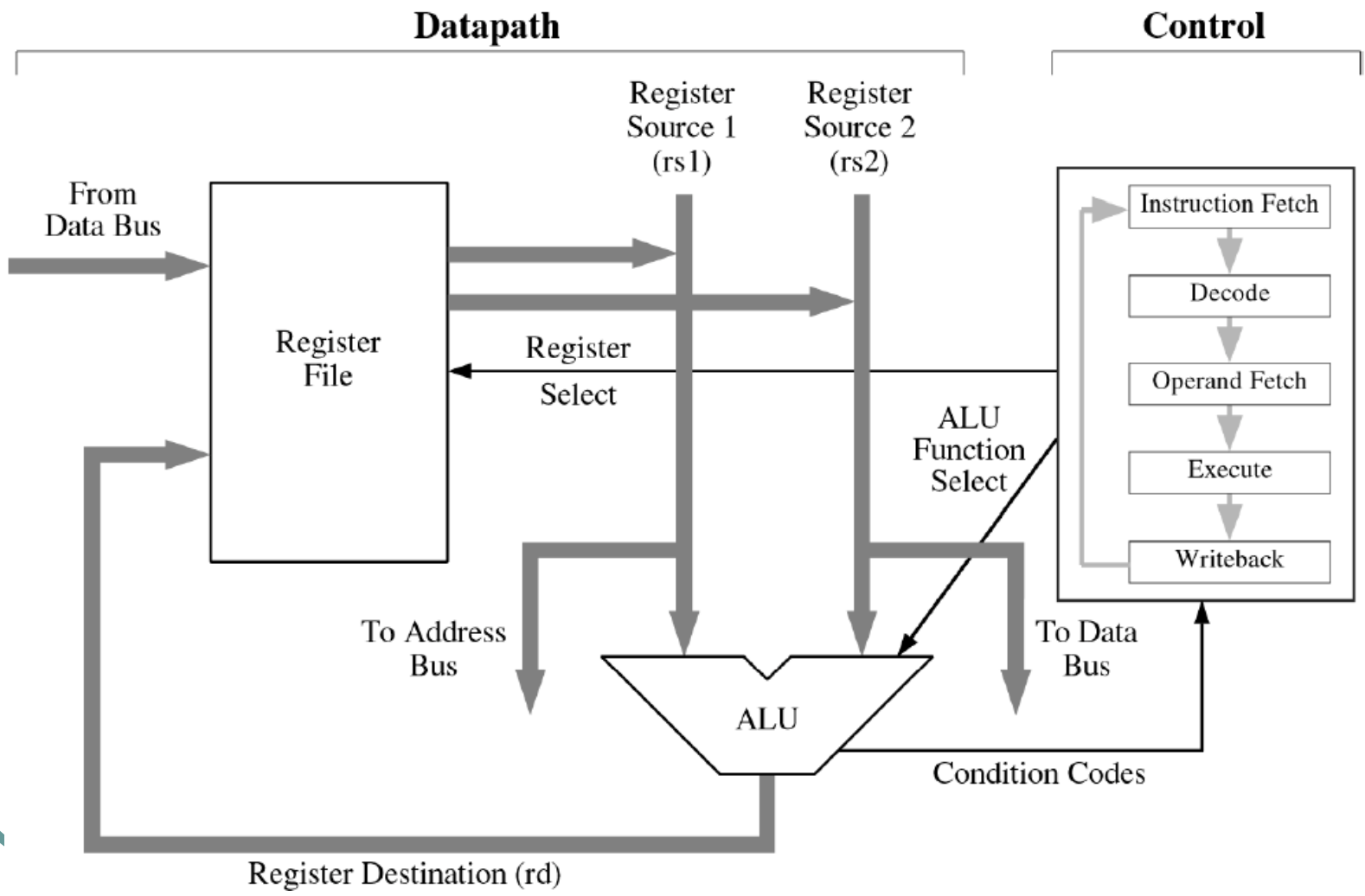
- (1) Fetch the next instruction to be executed from memory.
- (2) Decode the opcode.
- (3) Read operand(s) from main memory, if any.
- (4) Execute the instruction and store results, if any.
- (5) Go to step 1.

High Level View of Microarchitecture

- The microarchitecture consists of the control unit and the programmer-visible registers, functional units such as the ALU, and any additional registers that may be required by the control unit.



A More Detailed View



Datapath

Datapath

A *datapath* is a collection of functional units, such as arithmetic logic units or multipliers, that perform data processing operations.

Datapath

Datapath is a part of the microarchitecture

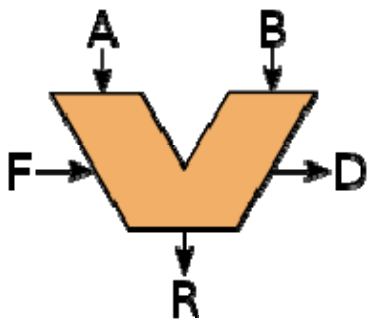
- Architecture
  - The ISA – the programmer's view of the machine
  - Implementation independent, an interface
- Microarchitecture
  - The lower-level implementation of the ISA
  - Design specific, and implementation
- Example:
  - Architecture state: Register $t5
  - Microarchitecture state: Carry bit on the ALU

Datapath Elements

- Computation Elements
  - Combination Circuits
  - Outputs follow inputs
  - e.g., ALU
- State Elements
  - Sequential Circuits
  - Output change on clock edge
  - e.g., A Register

Computation Element: ALU

- Arithmetic Logic Unit
  - a digital circuit that performs arithmetic and logical operations
  - loads data from input registers, (an external Control Unit then tells the ALU what operation to perform on that data,) and then the ALU stores its result into an output register.
  - a fundamental building block of the central processing unit (CPU) of a computer



Computation Element: ALU

- Basic operations
  - Integer arithmetic operations
    - addition, subtraction
    - multiplication
    - Division
  - Bitwise logic operations
    - AND, NOT, OR, XOR
  - Bit-shifting operations
    - shifting or rotating a word by a specified number of bits to the left or right, with or without sign extension
    - can be interpreted as multiplications by 2 and divisions by 2.

Computation Element: ALU

- Complex operations
  - any operations, say, square root
  - different methods
    - calculation in a single clock
    - calculation pipeline
    - Interactive calculation
    - co-processor
    - ...
  - speed vs. cost
    - tradeoff

Computation Element: Others

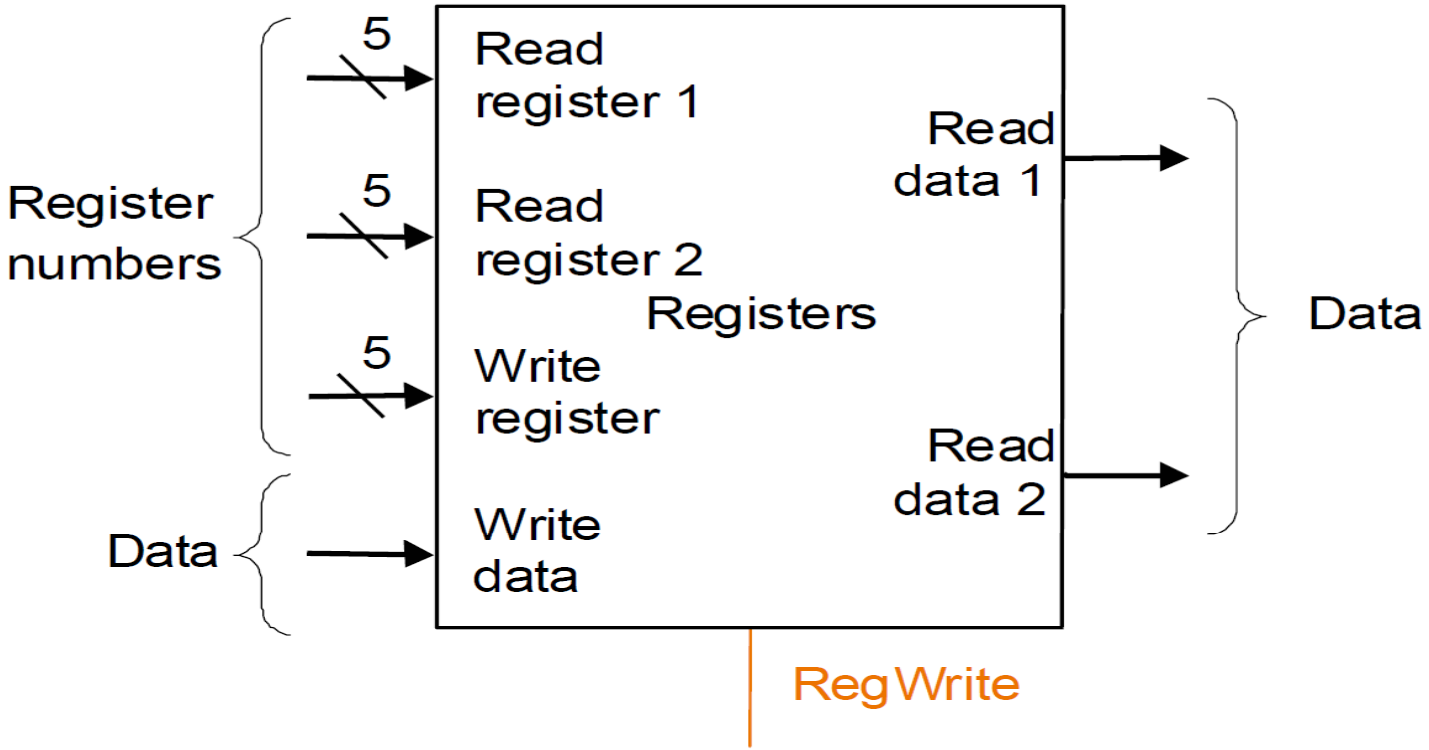
- FPU
  - Float Point Unit
  - performs arithmetic operations between two floating point values
- Mux
  - Multiplexer
  - *n* input lines, m control wires to select
  - $n = 2^m$
- …

State Element: Register File

- Register File
  - not a "*file*"
  - an array of processor registers in a central processing unit (CPU)
  - microarchitecture to implement architectural state
  - built using D flip-flops
  - multiple read and write ports

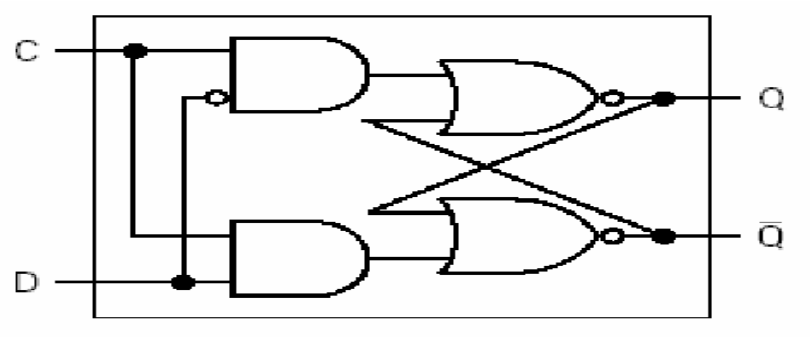
State Element: Register File

- Read two operands at once
- 2 source operands per instruction

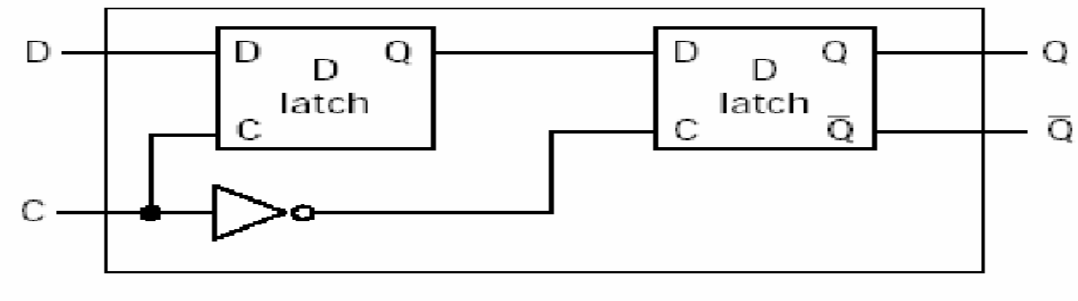


State Element: Register File

- Register Implementation



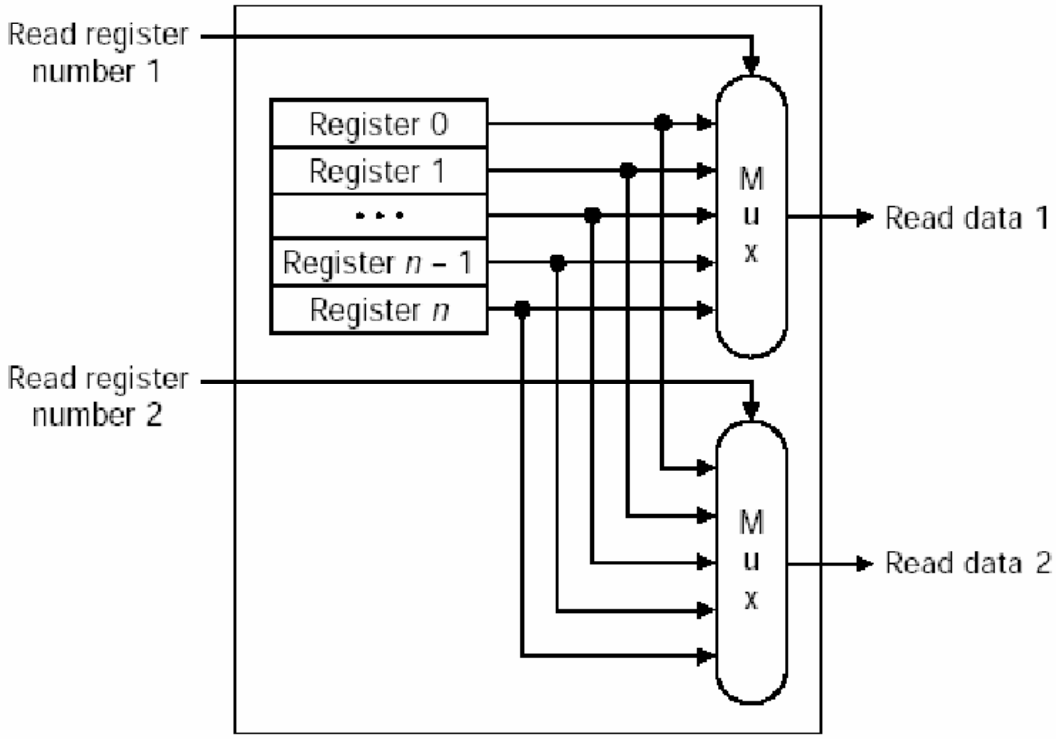
D latch

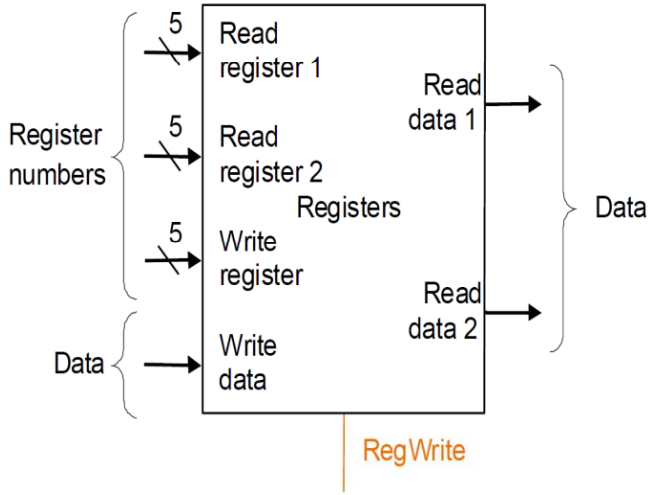


Falling edge triggered D flip-flop

State Element: Register File

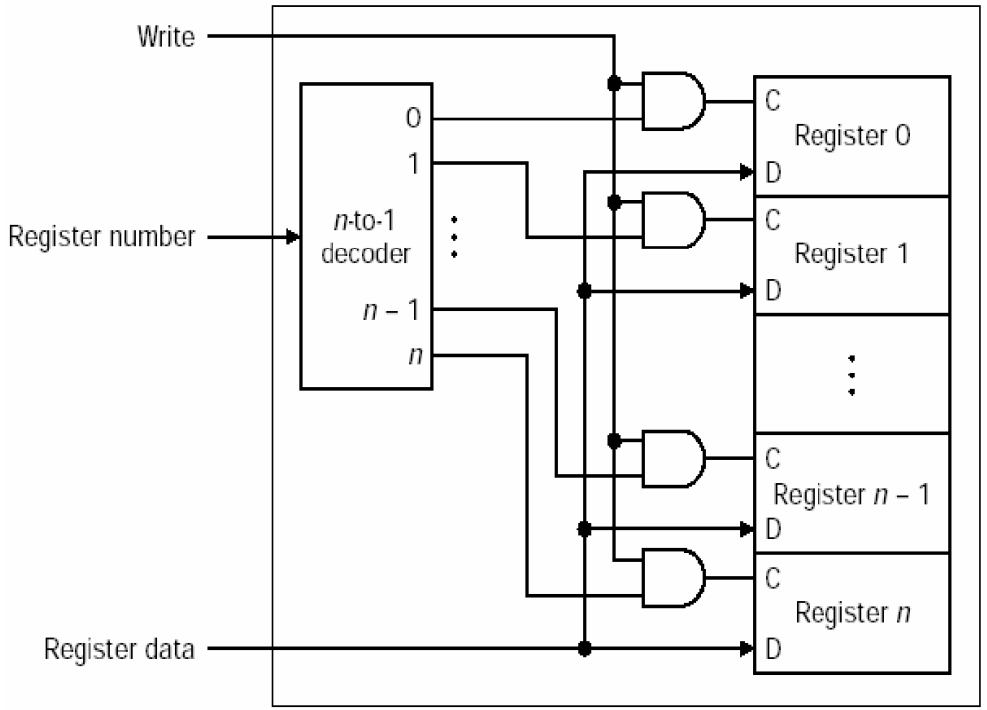
- Read Implementation

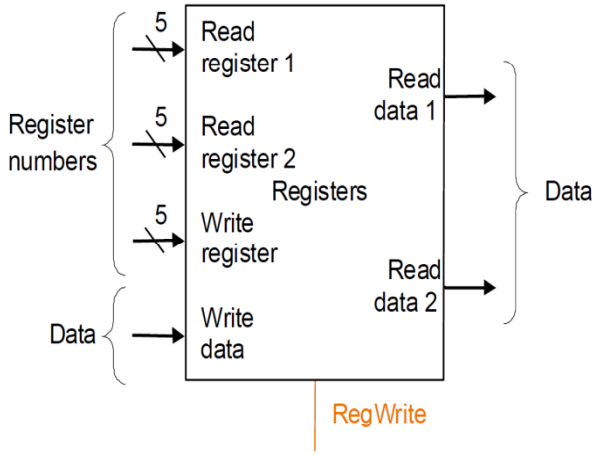




State Element: Register File

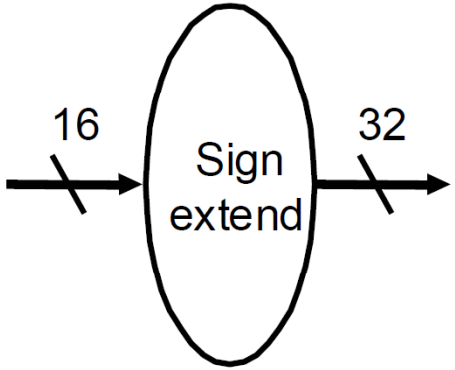
- Write Implementation



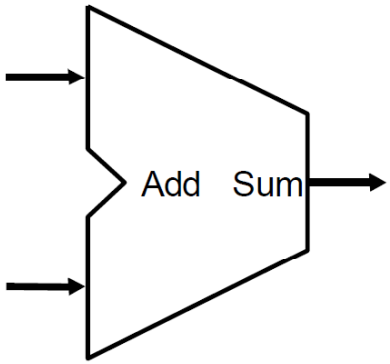


Putting them all together

- more elements
  - computation elements



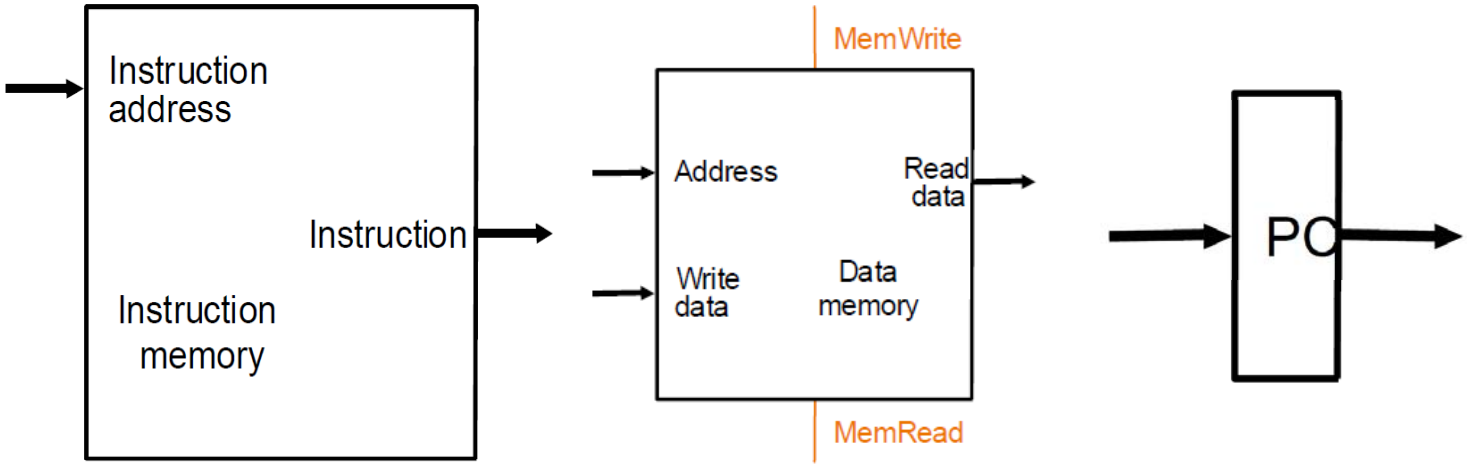
Sign Extend



Adder

Putting them all together

- more elements
  - state element



Instruction memory | Data memory | Program counter

Putting them all together

- A list
  - Computation elements:
    - ALU, Adder
    - Mux
    - Sign extend
  - State elements:
    - Register file, Instruction memory, Data memory
    - PC

Abstract View of MIPS Subset Implementation

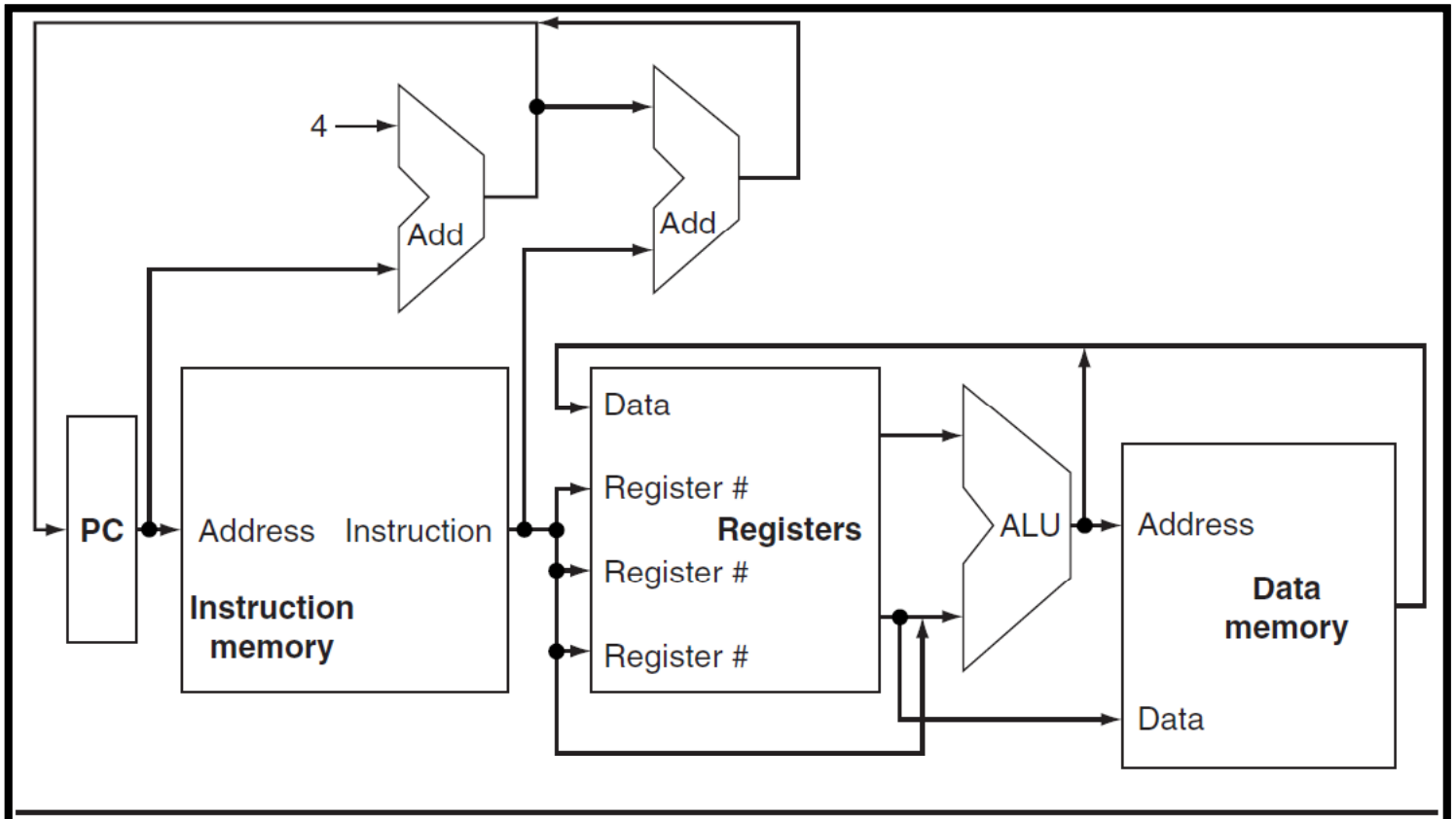
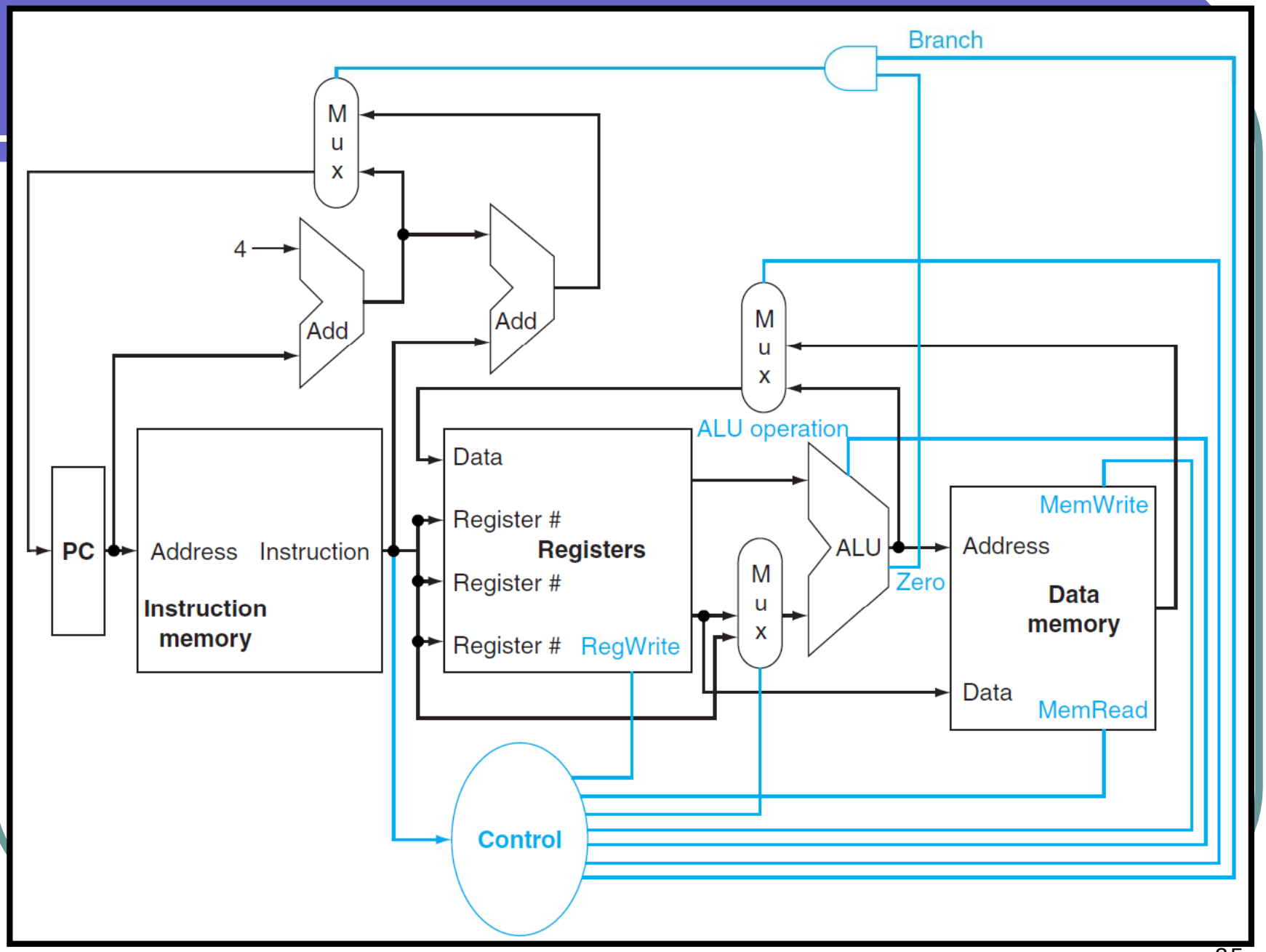


FIGURE 5.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the pro-

Abstract View of MIPS Subset Implementation

FIGURE 5.1 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 from either 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.


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

Building a Datapath

- R-format instruction
  - perform arithmetic or logical operations
  - add, sub, and, or, and slt

B_{31-26}	B_{25-21}	B_{20-16}	B_{15-11}	B_{10-6}	B_{5-0}
opcode	register s	register t	register d	shift amount	function

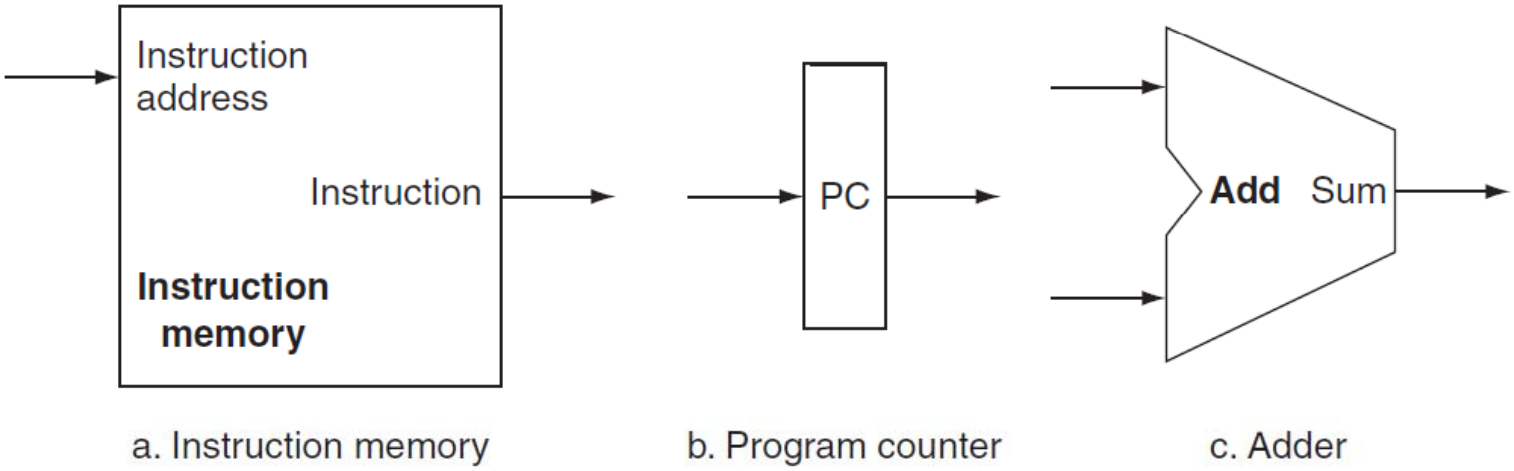
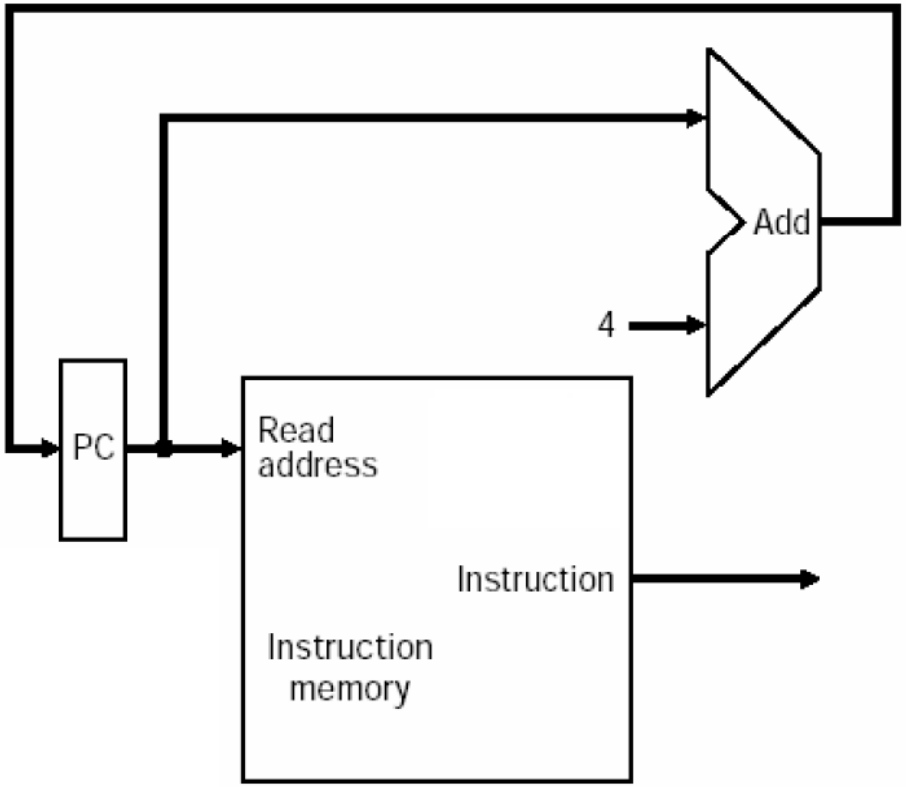


FIGURE 5.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction

FIGURE 5.5 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 will be 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 perform an add of its two 32-bit inputs and place the result on its output.

Building a Datapath

- Fetching Instructions (no branching)



- R-type instruction: four inputs (three for register numbers and one for data) and two outputs (both for data)

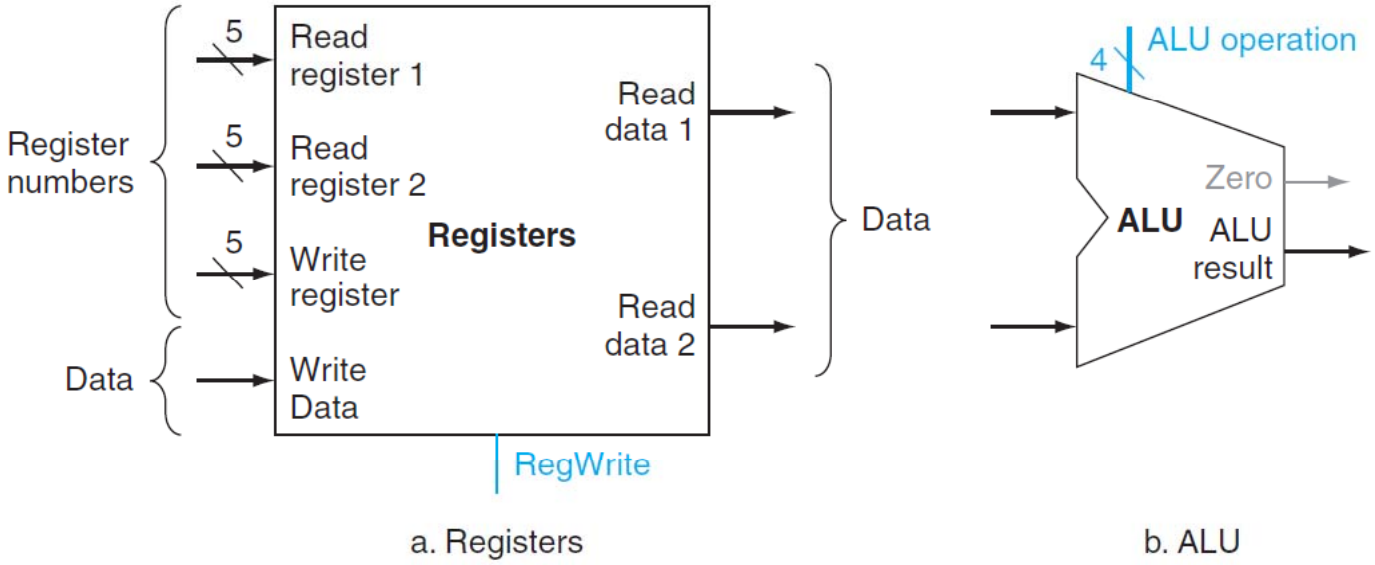
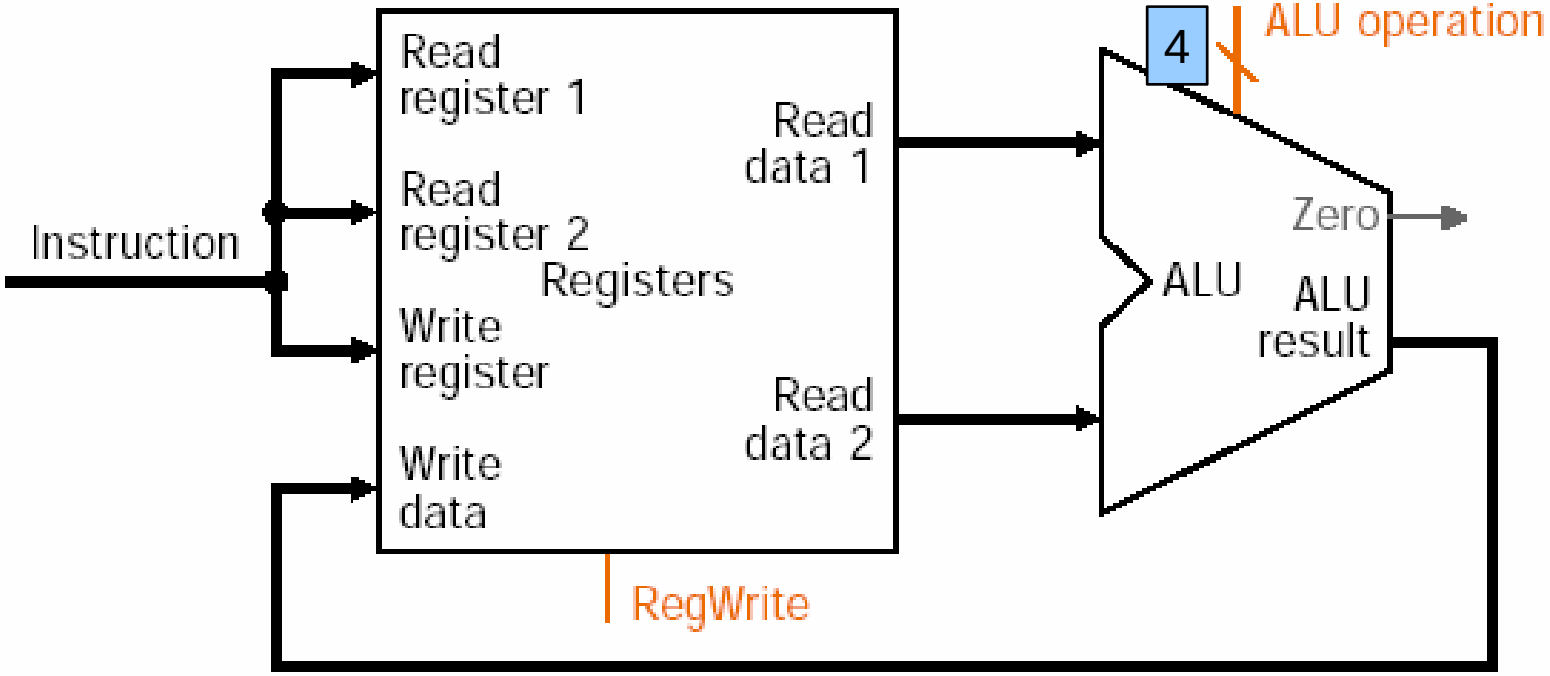


FIGURE 5.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port.

FIGURE 5.7 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 B.8 of Appendix B. 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 B. We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 5.6, when we discuss exceptions; we omit it until then.

- The ALU instructions

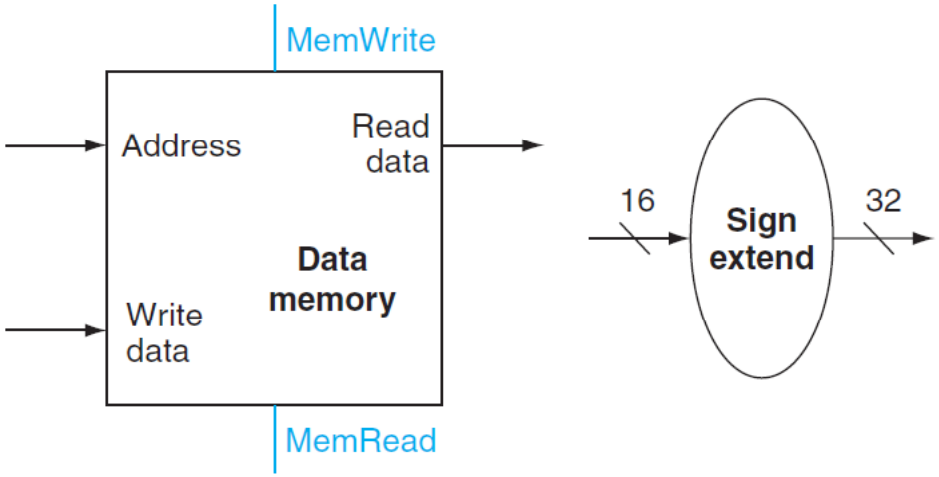


REG3=REG1-REG2 SUB $3, $1, $2

Datapath

16 bit

- Load and store word instructions (MEM)
  - lw / sw $t1,offset_value($t2)
  - Need a unit to **sign-extend the 16-bit offset field** to a 32-bit signed value
  - Need a data memory unit to read from or write to



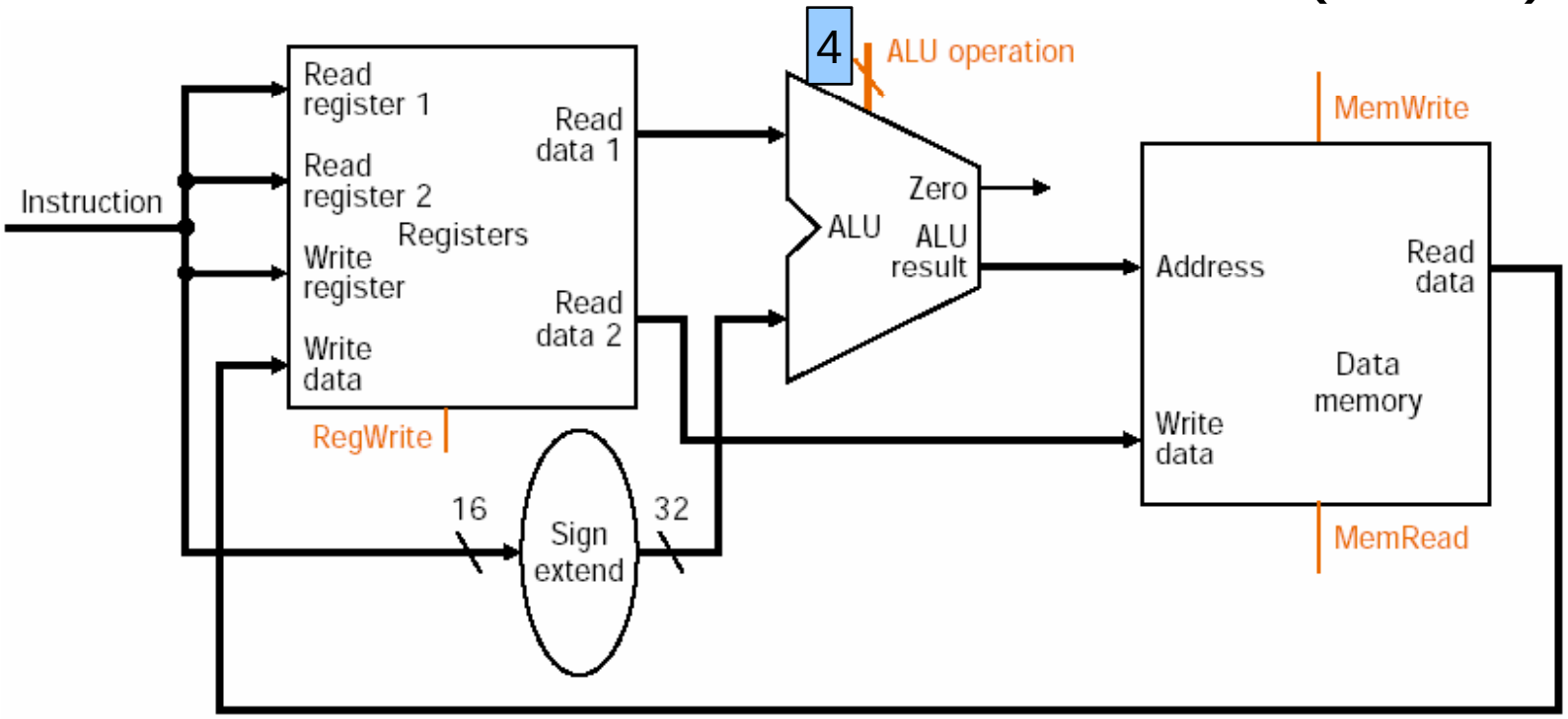
a. Data memory unit b. Sign-extension unit

FIGURE 5.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 5.7, are the data memory unit and the sign extension unit. The

FIGURE 5.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 5.7, 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 7. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 3). 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 B.8 of Appendix B for a further discussion of how real memory chips work.

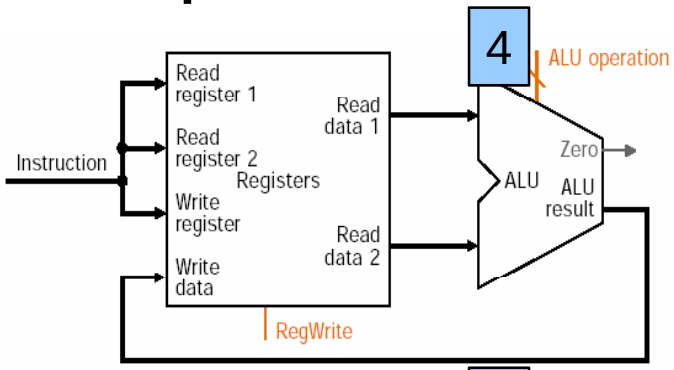
Datapath

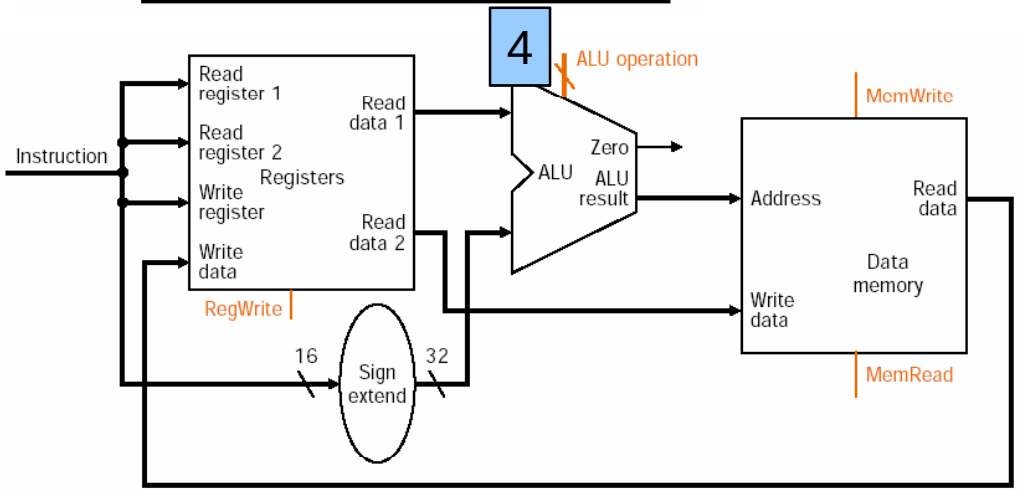
- Load and store word instructions (MEM)

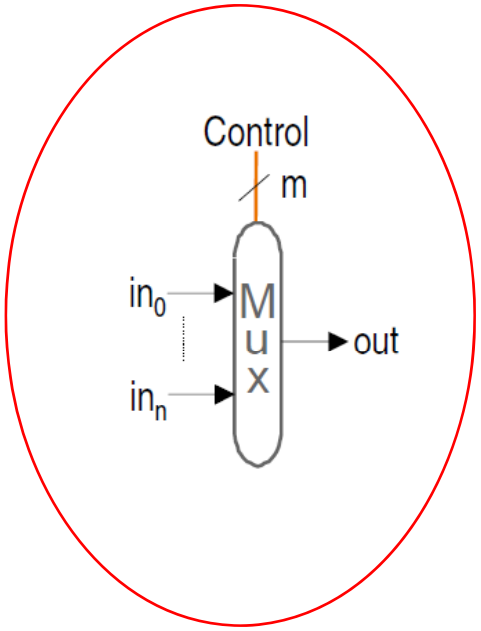


Datapath

- Composition of Memory and ALU

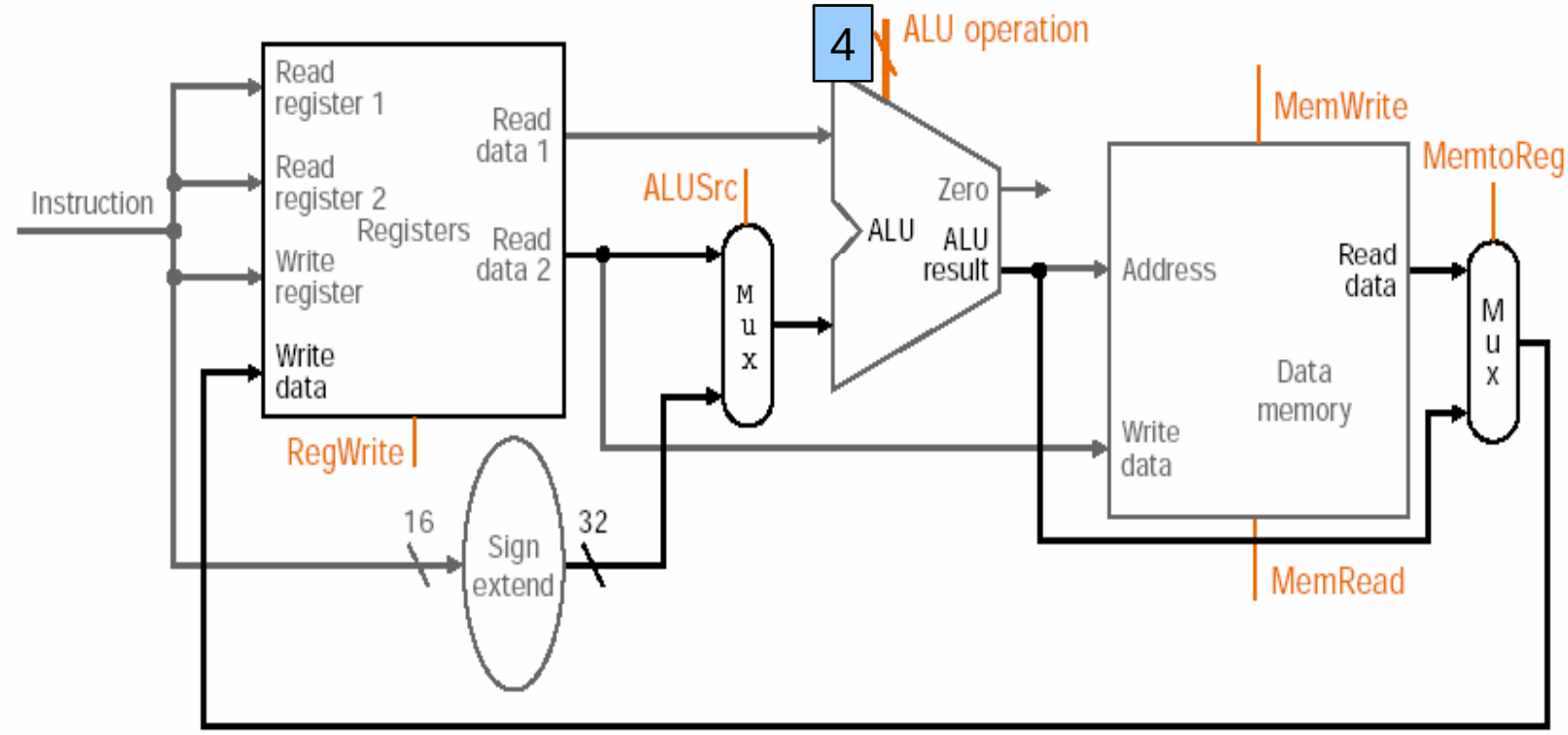






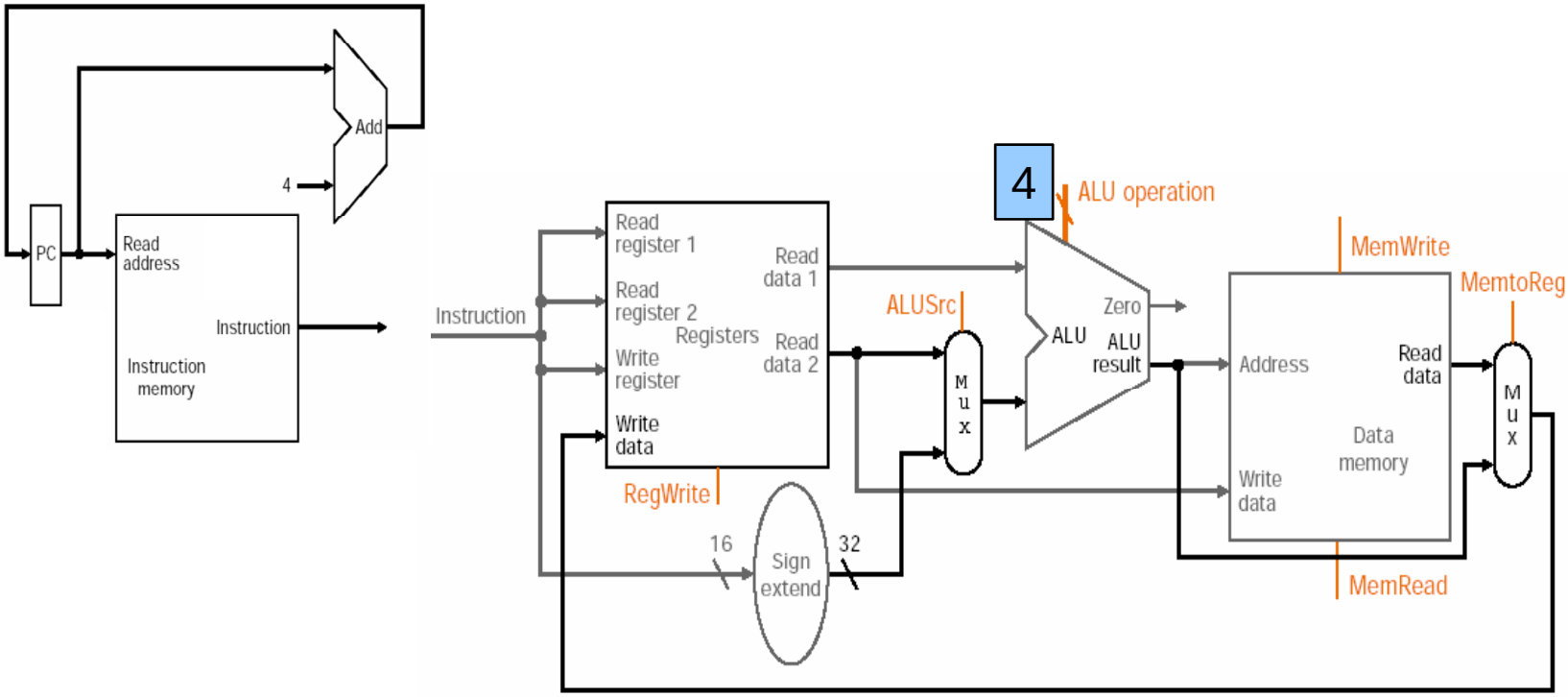
Datapath

- ALU+MEM



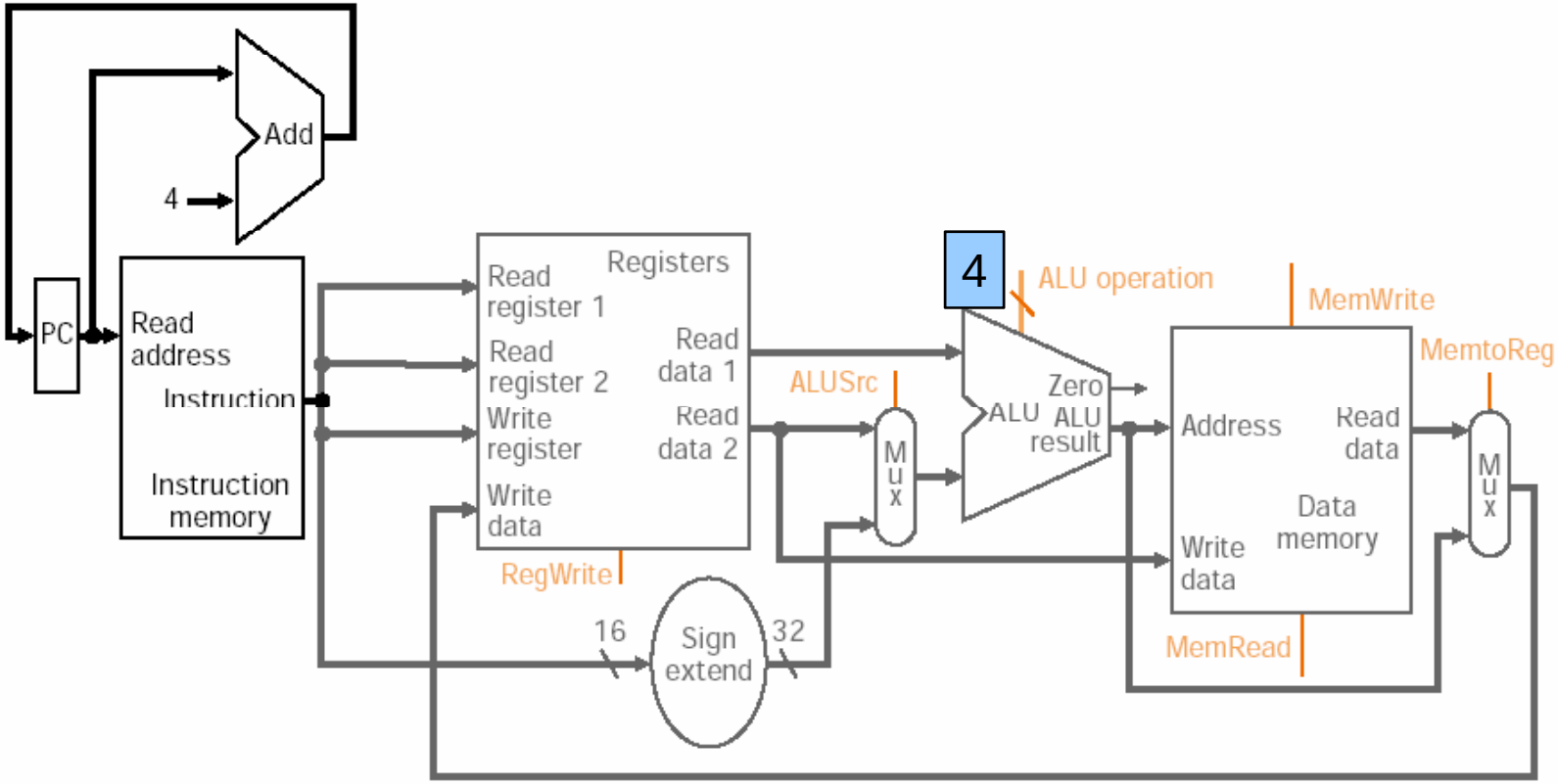
Datapath

- …and Fetch?



Datapath

- ALU+MEM+Fetch



Datapath

- Branch: beq $t1,$t2,offset
- Branch target address (add the sign-extended offset to the PC)
- the offset field is shifted left 2 bits so that it is a word offset
- branch datapath must do two operations: compute the branch target address and compare the register contents

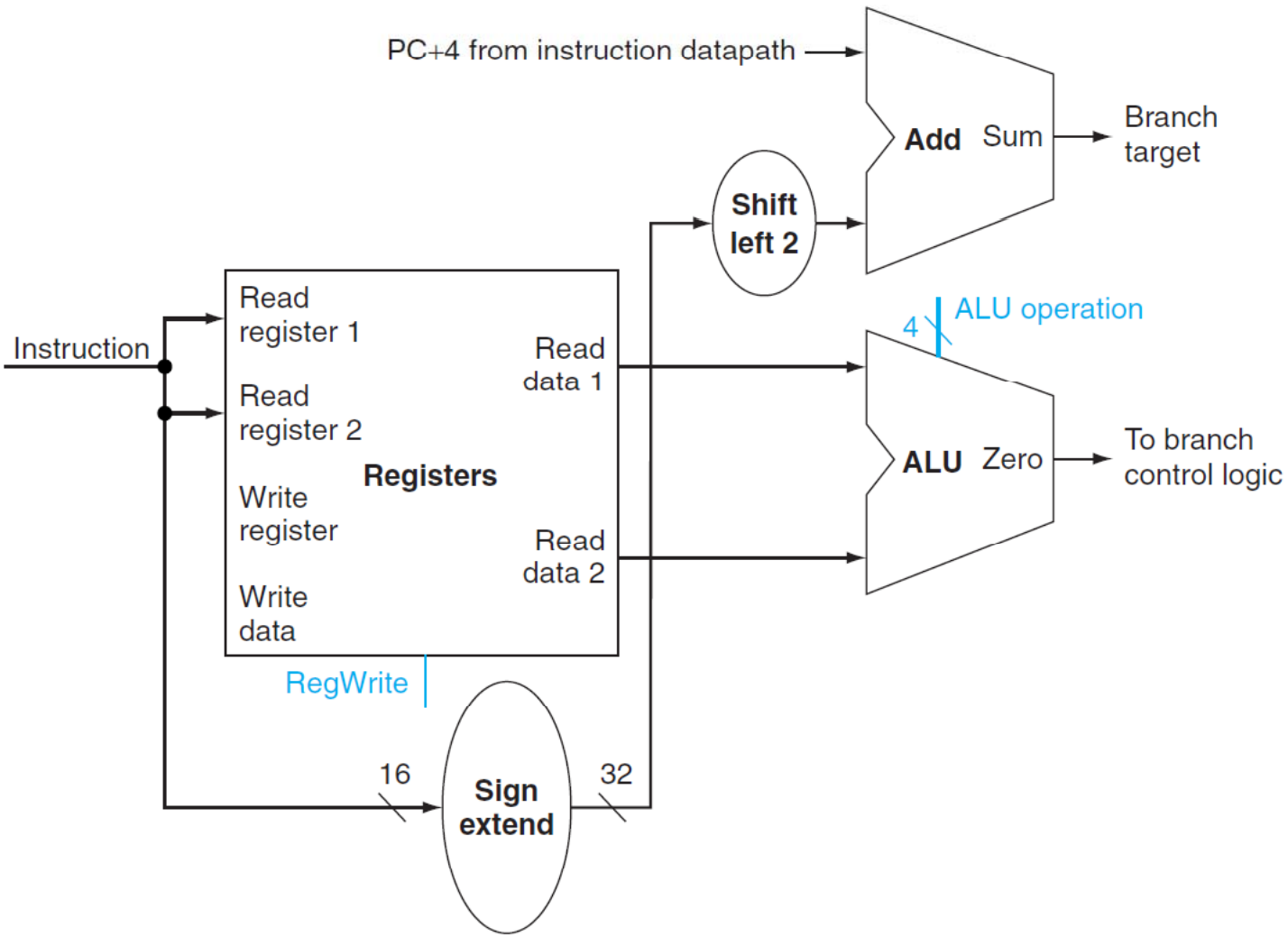


FIGURE 5.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to

Datapath

- Jump instruction
- Replace the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits
- This shift is accomplished simply by concatenating 00 to the jump offset

Creating a single datapath

- To share a datapath, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs
- The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are:
  - The arithmetic-logical instructions use the ALU with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 16-bit offset field from the instruction
  - The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

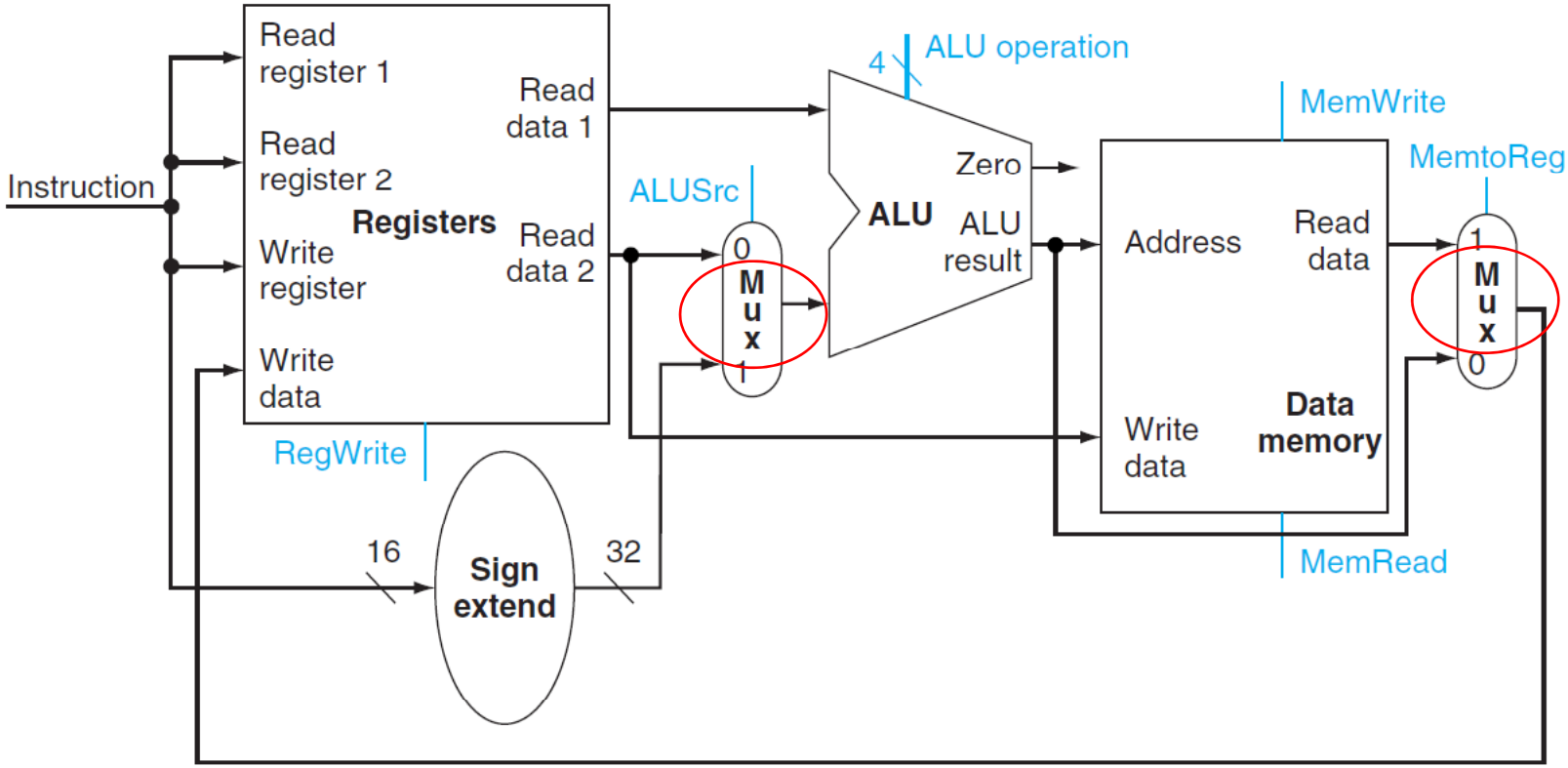


FIGURE 5.10 The datapath for the memory instructions and the R-type instructions. This example shows hc

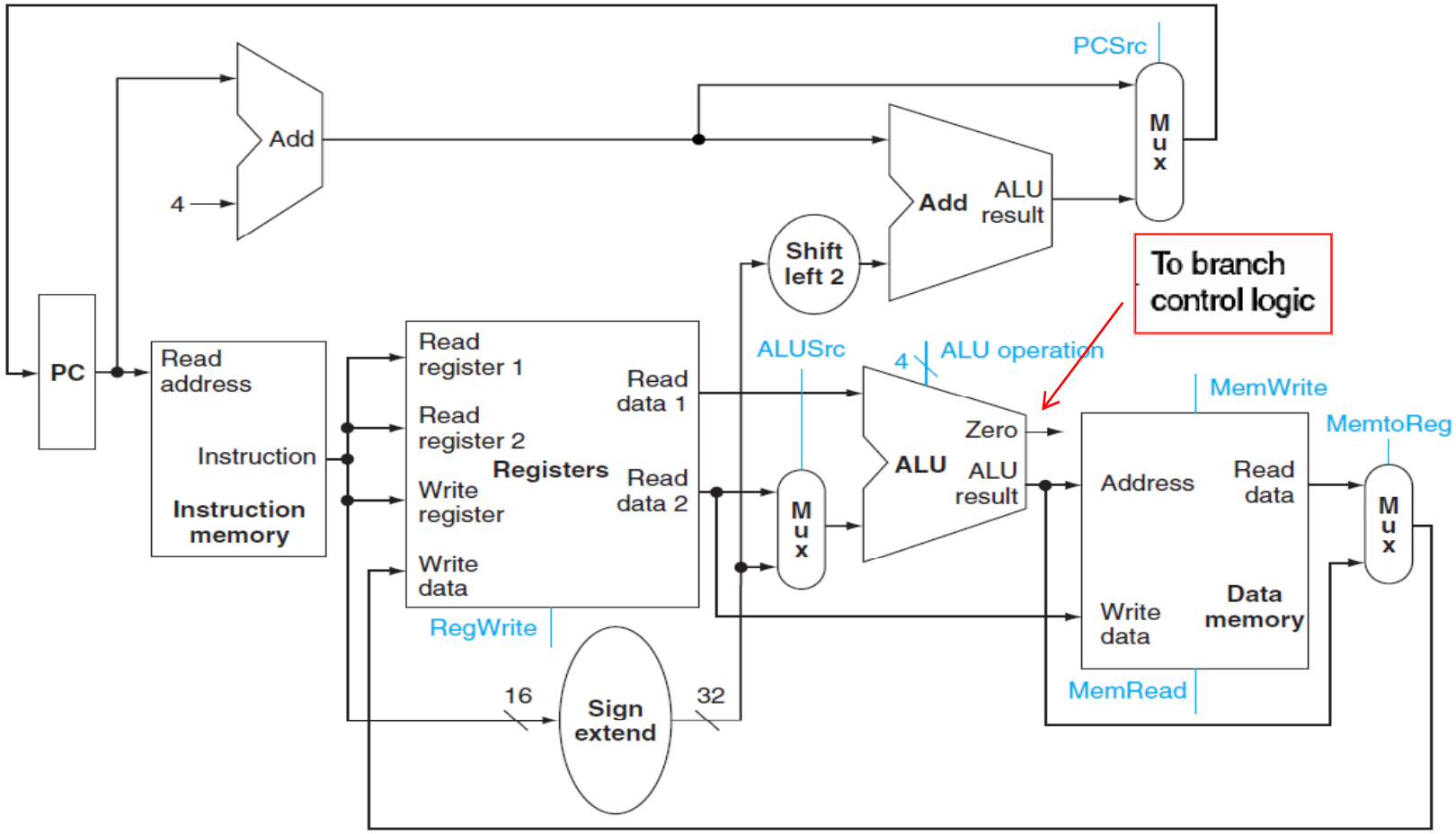


FIGURE 5.11 The simple datapath for the MIPS architecture combines the elements required by different instruction classes. This datapath can execute the basic instructions (load/store word, ALU operations, and branches) in a single clock cycle. An additional multiplexor is needed to integrate branches. The support for jumps will be added later.

Control

- Datapath
  - A *datapath* is a collection of functional units, such as arithmetic logic units or multipliers, that perform data processing operations.
- Control
  - The control commands the datapath, memory, and I/O devices according to the instructions of the program.

ALU Control

- ALU has four control inputs. These bits were not encoded; hence, only 6 of the possible 16 possible input combinations are used in this subset

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

- We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp
- ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for beq, or determined by the operation encoded in the funct field (10)
- The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 5.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we "don't care" about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

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

FIGURE 5.13 The truth table for the three 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. Also, when the function field is used, the first two 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.

don't-care term: An element of a logical function in which the output does not depend on the values of all the inputs.

Three type of instructions

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

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

FIGURE 5.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats. The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `add`, `sub`, `and`, `or`, and `slt`. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC to compute the branch target address.

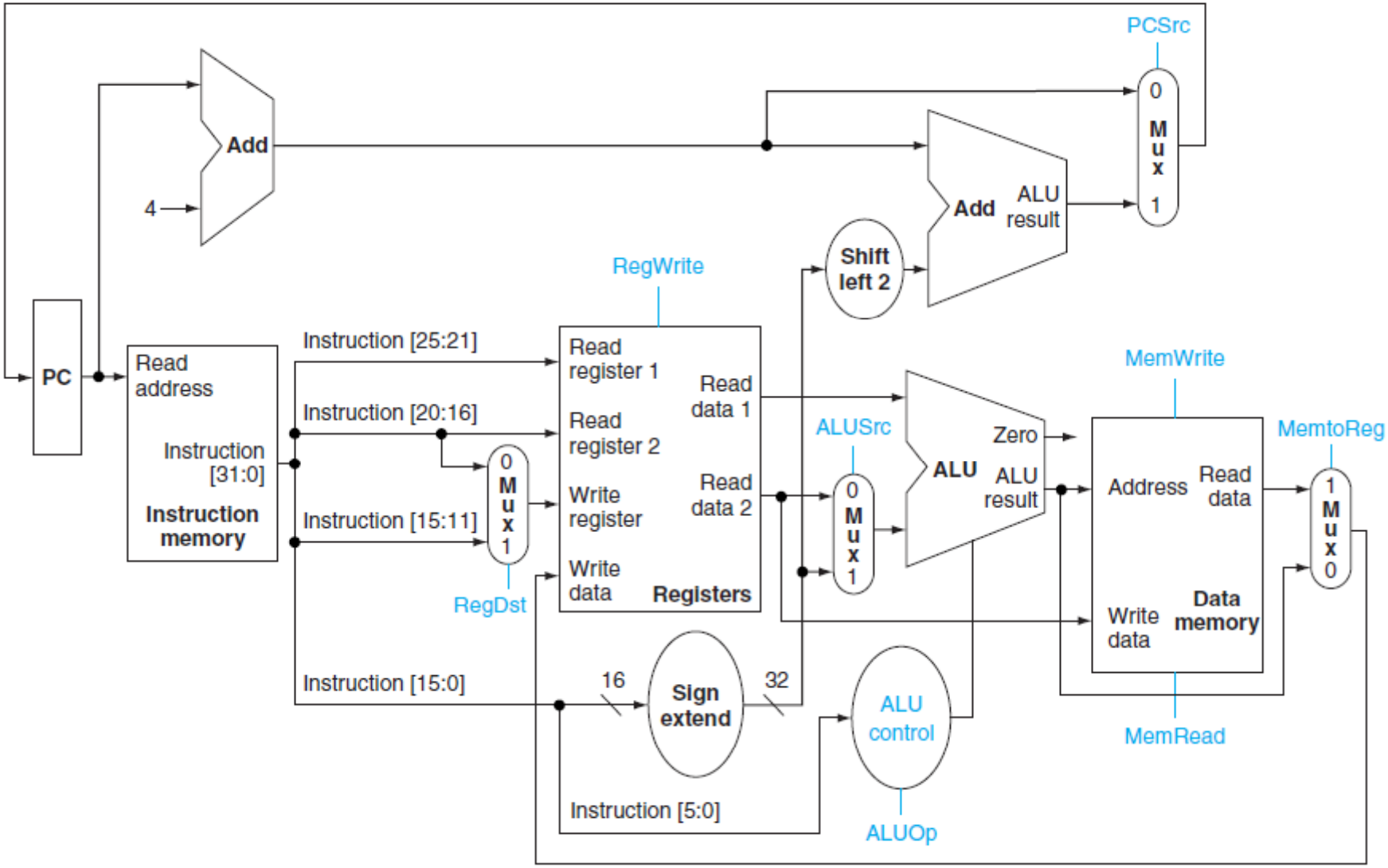


FIGURE 5.15 The datapath of Figure 5.12 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

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.

FIGURE 5.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. The clock is never gated externally to a state element, since this can create timing problems. (See Appendix B for further discussion of this problem.)

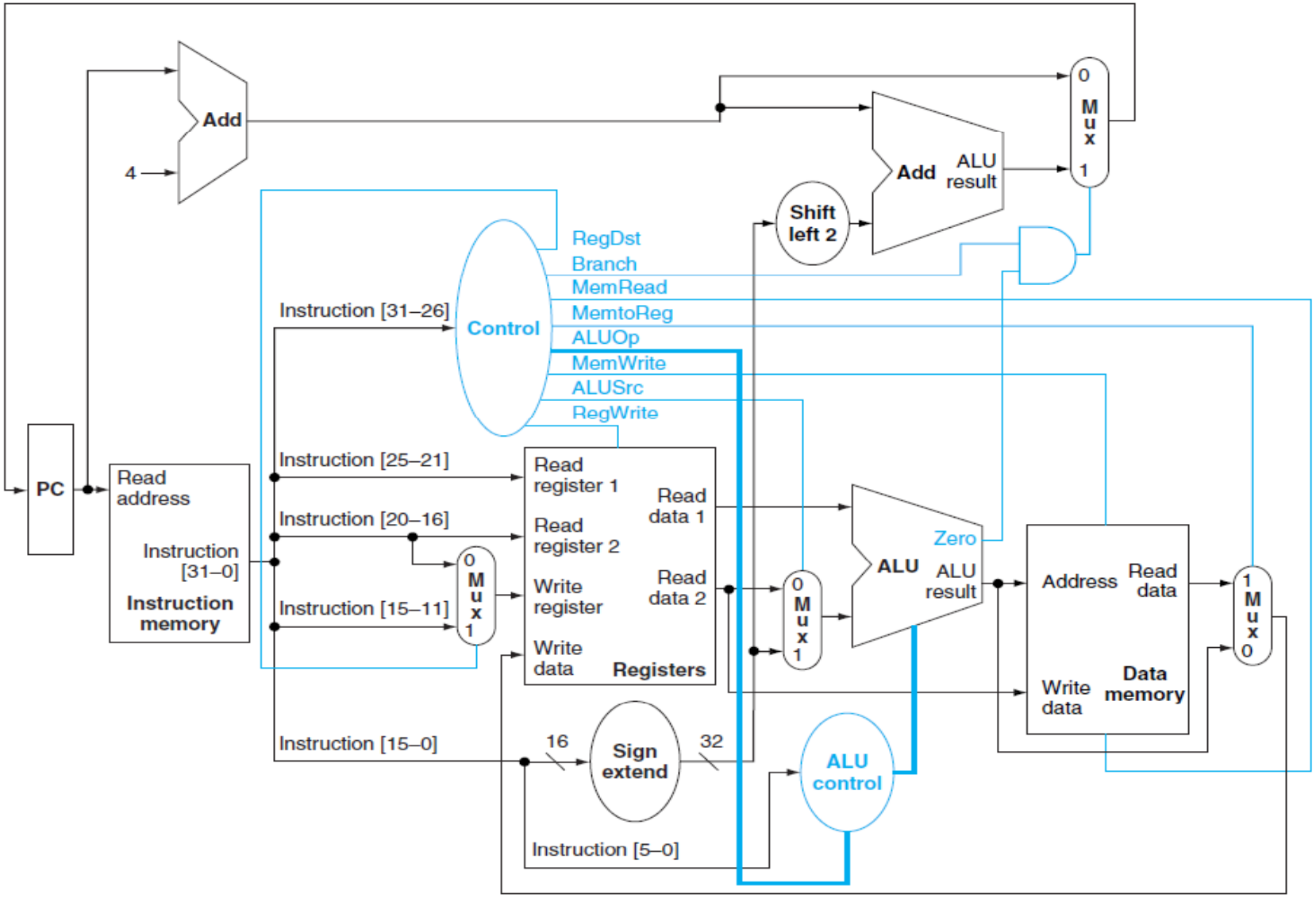


FIGURE 5.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads 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.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

FIGURE 5.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of

FIGURE 5.18 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.

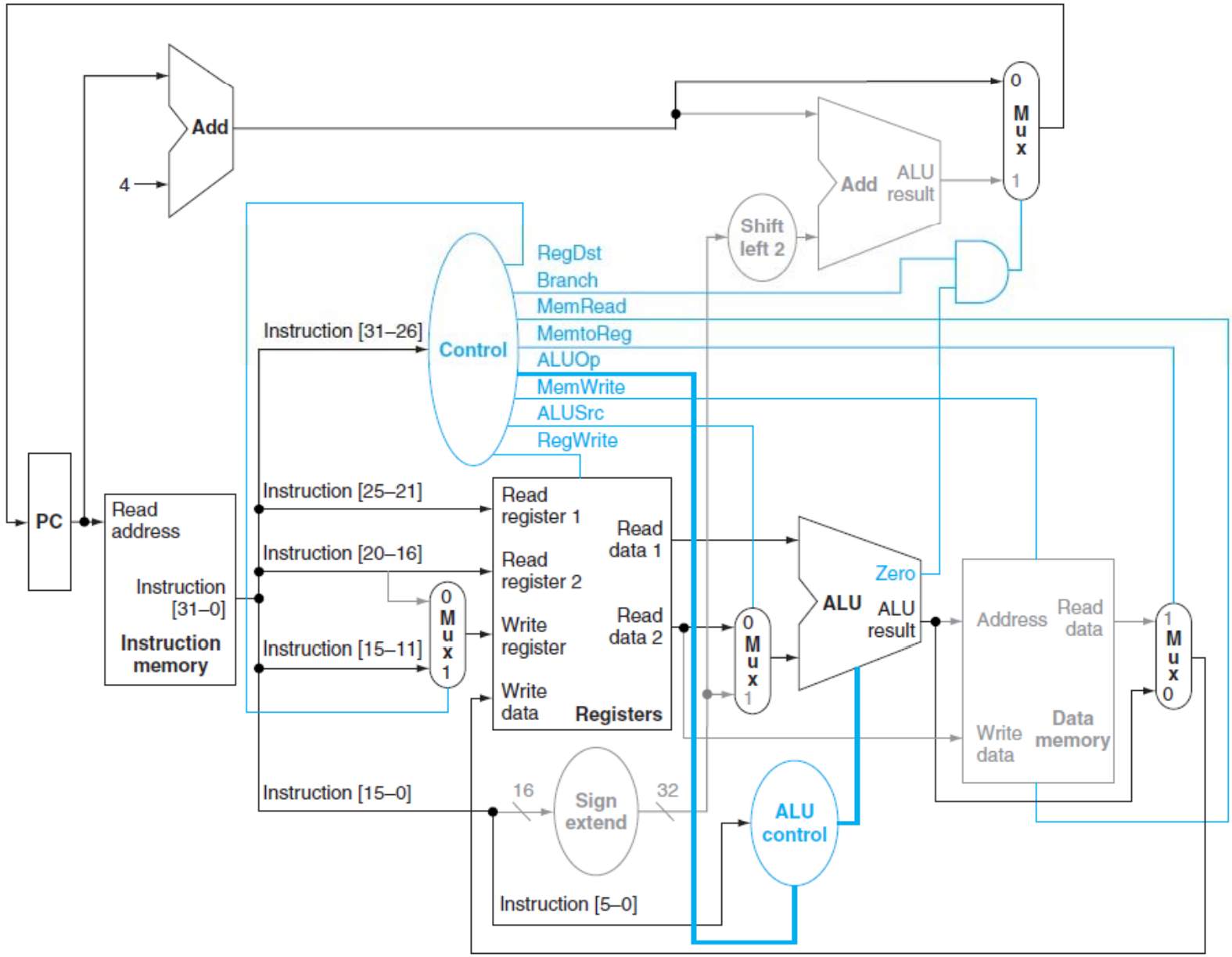


FIGURE 5.19 The datapath in operation for an R-type instruction such as add `$t1,$t2,$t3`. The control lines, datapath units, and connections that are active are highlighted.

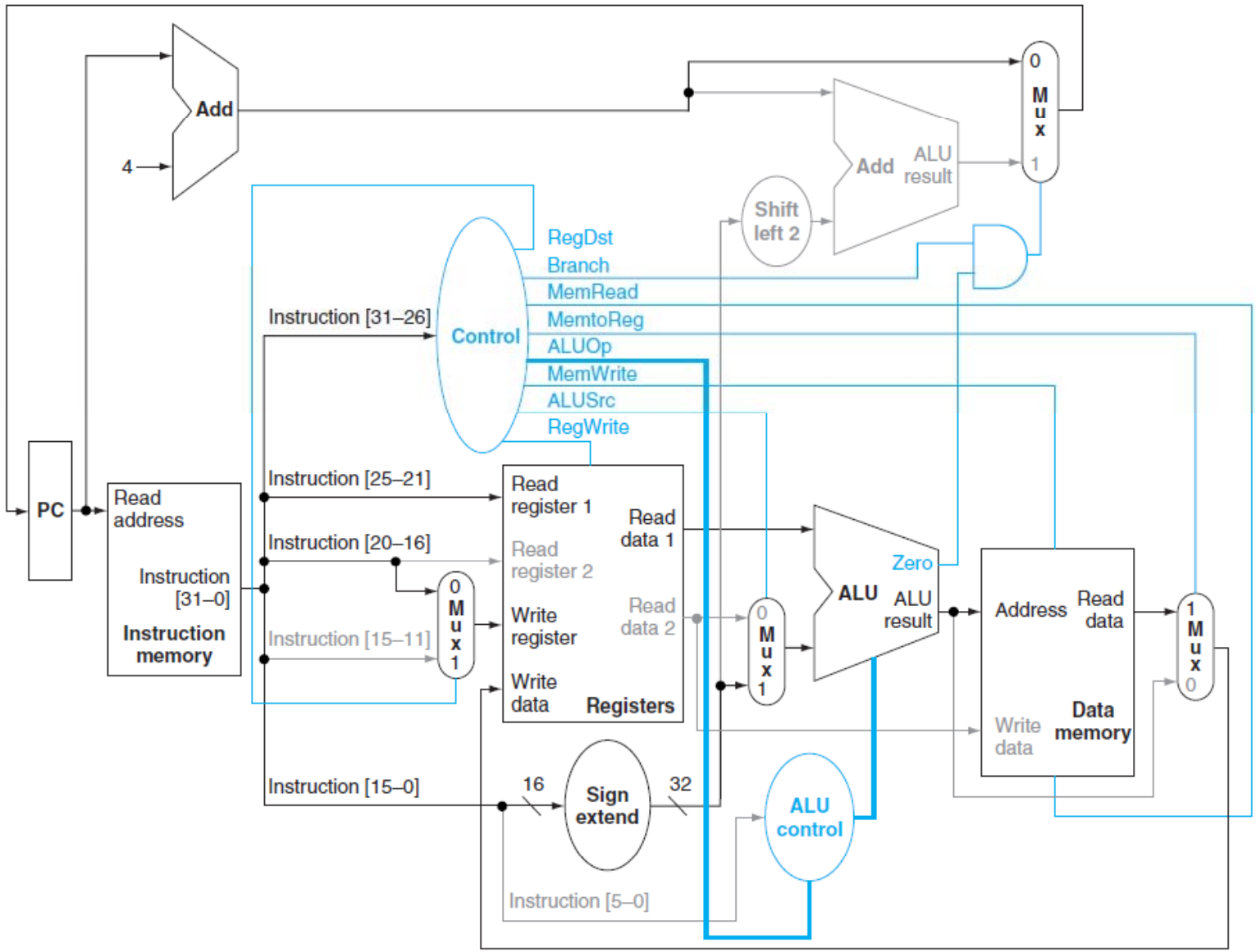


FIGURE 5.20 The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

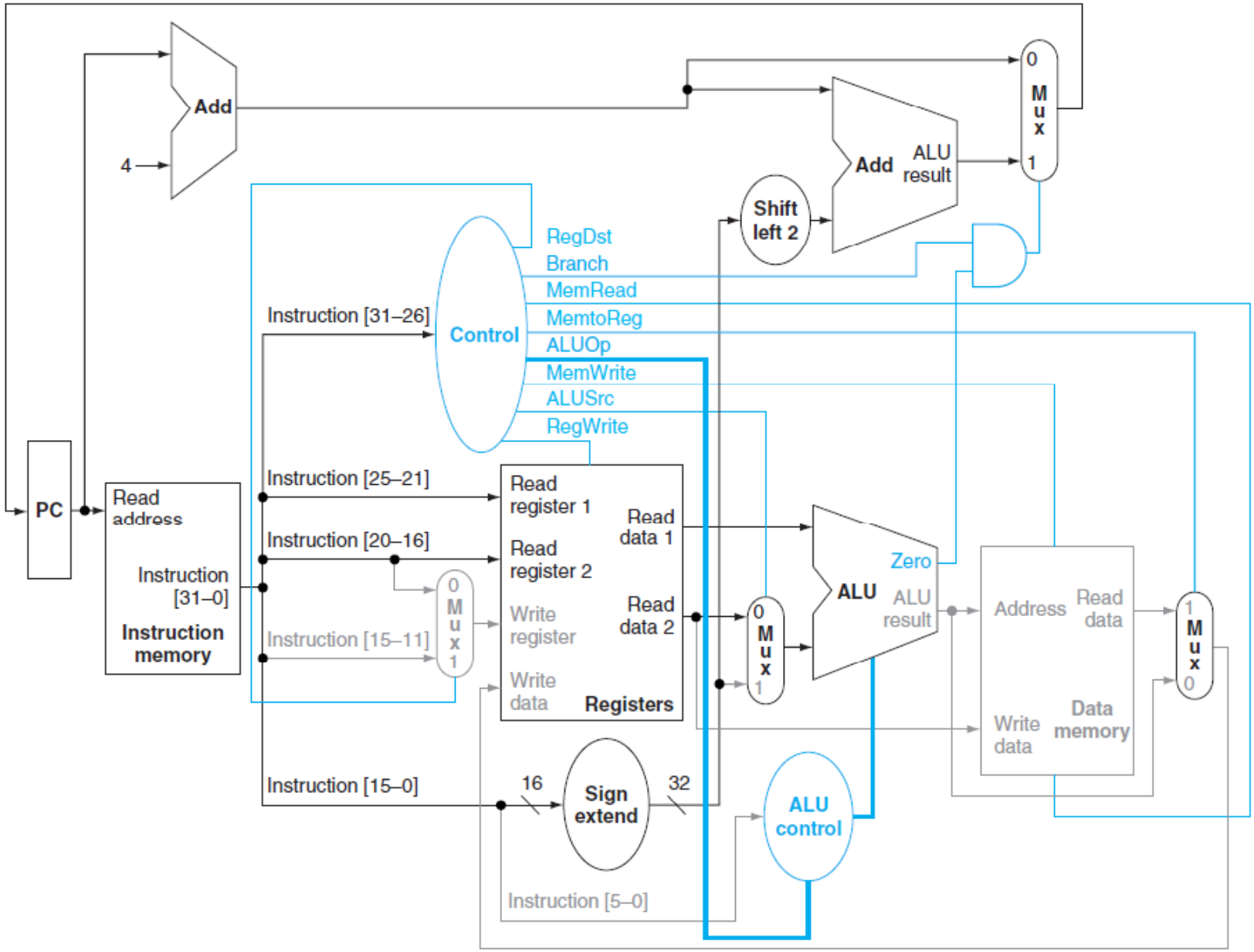


FIGURE 5.21 The datapath in operation for a branch equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

Input or output	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 5.22 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes that determine the control output settings. (Remember that Op [5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{Op5} \cdot \overline{Op2}$, since this is sufficient to distinguish the R-format instructions from lw, sw, and beq. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

Implementing Jumps

- we can implement jump by storing into the PC the concatenation of
  - the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)
  - the 26-bit immediate field of the jump instruction
  - the bits 00B

Field	000010	address
Bit positions	31:26	25:0

FIGURE 5.23 Instruction format for the jump instruction (opcode = 2). The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC + 4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.

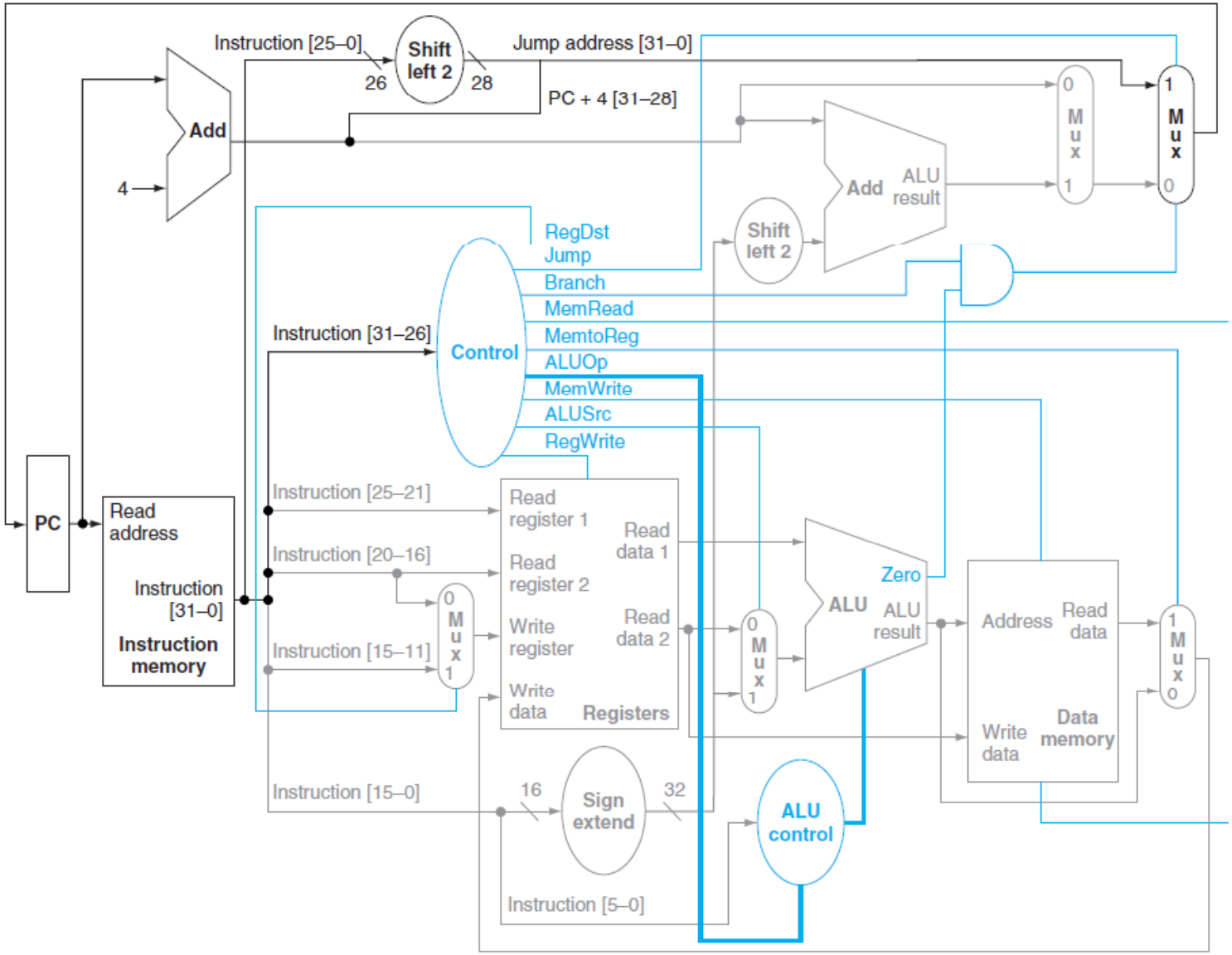
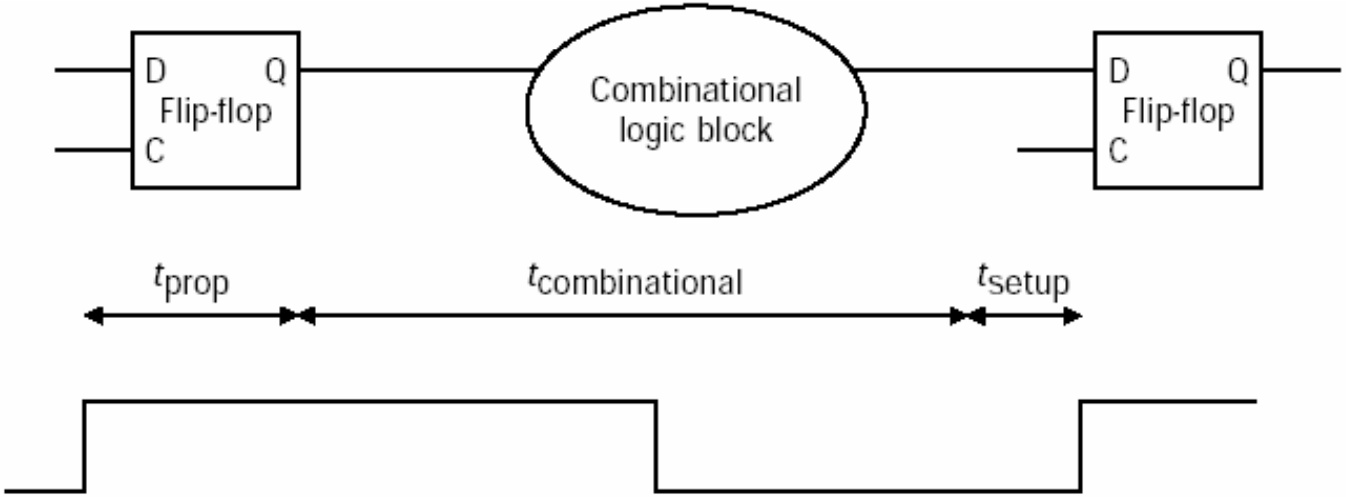


FIGURE 5.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Single cycle implementation

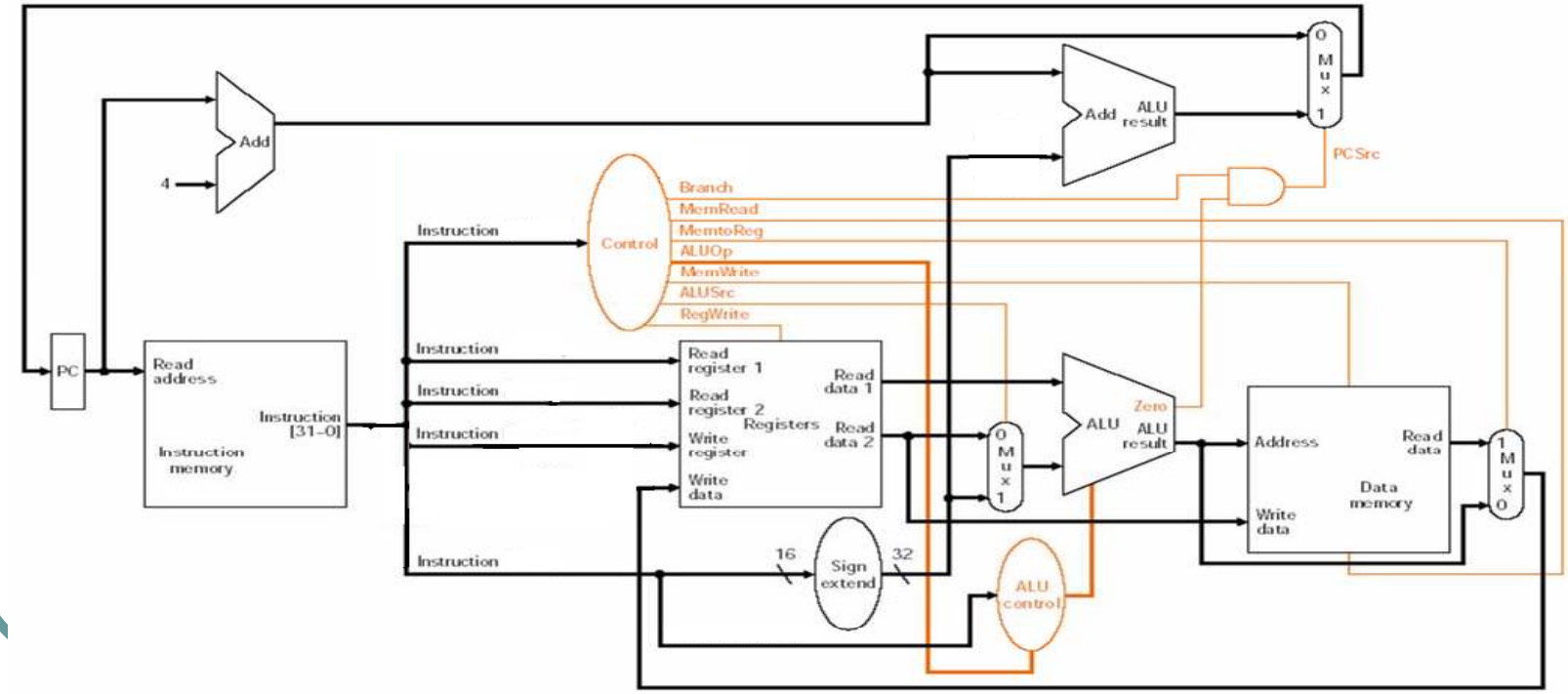
- **single-cycle implementation** Also called single clock cycle implementation.
- An implementation in which an instruction is executed in one clock cycle
- Although the single-cycle design will work correctly, it would not be used in modern designs because it is inefficient



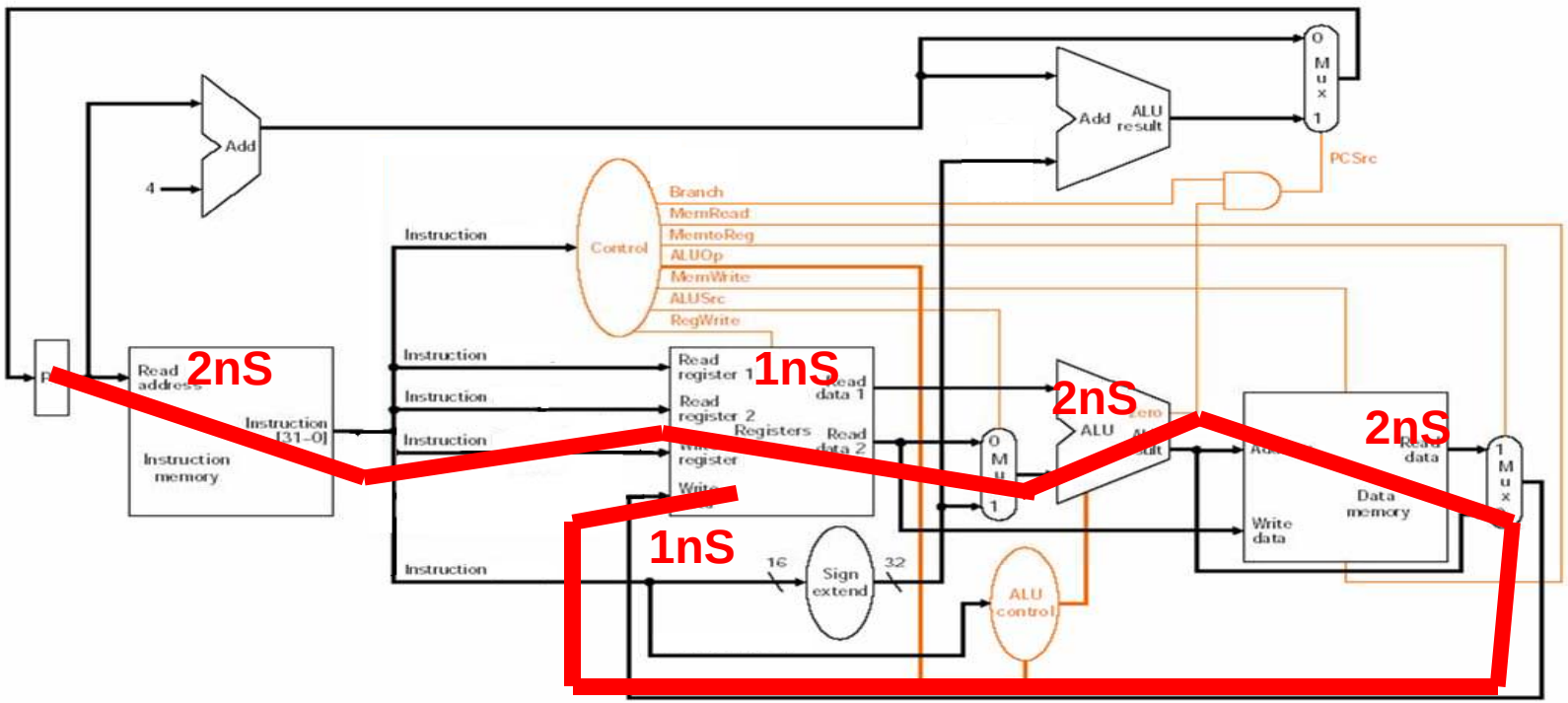
- *Single Cycle Implementation*
- We wait for everything to settle down
  - ALU might not produce “right answer” right away
- Every instruction has the same length clock cycle
- Cycle time determined by length of the longest path
- CPI = 1

Single Cycle Implementation?

- Calculate cycle time assuming negligible delays except: memory (2ns), ALU/adders (2ns), register file access (1ns)



Single Cycle Implementation?



- Load is longest running instruction: memory (2ns), ALU/adders (2ns), register file access (1ns)
- 2ns + 1ns + 2ns + 2ns + 1ns = 8ns

Multicycle implementation

- **Multicycle implementation Also called** multiple clock cycle implementation
- An implementation in which an instruction is executed in multiple clock cycles

Multicycle Approach

- Break up the instructions into steps, one per cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers to hold values between cycles

Multicycle Approach

- Reuse datapath components
  - ALU is used to compute address and to increment PC
  - Memory is used for instruction and data
- Control signals are not determined solely by instructions

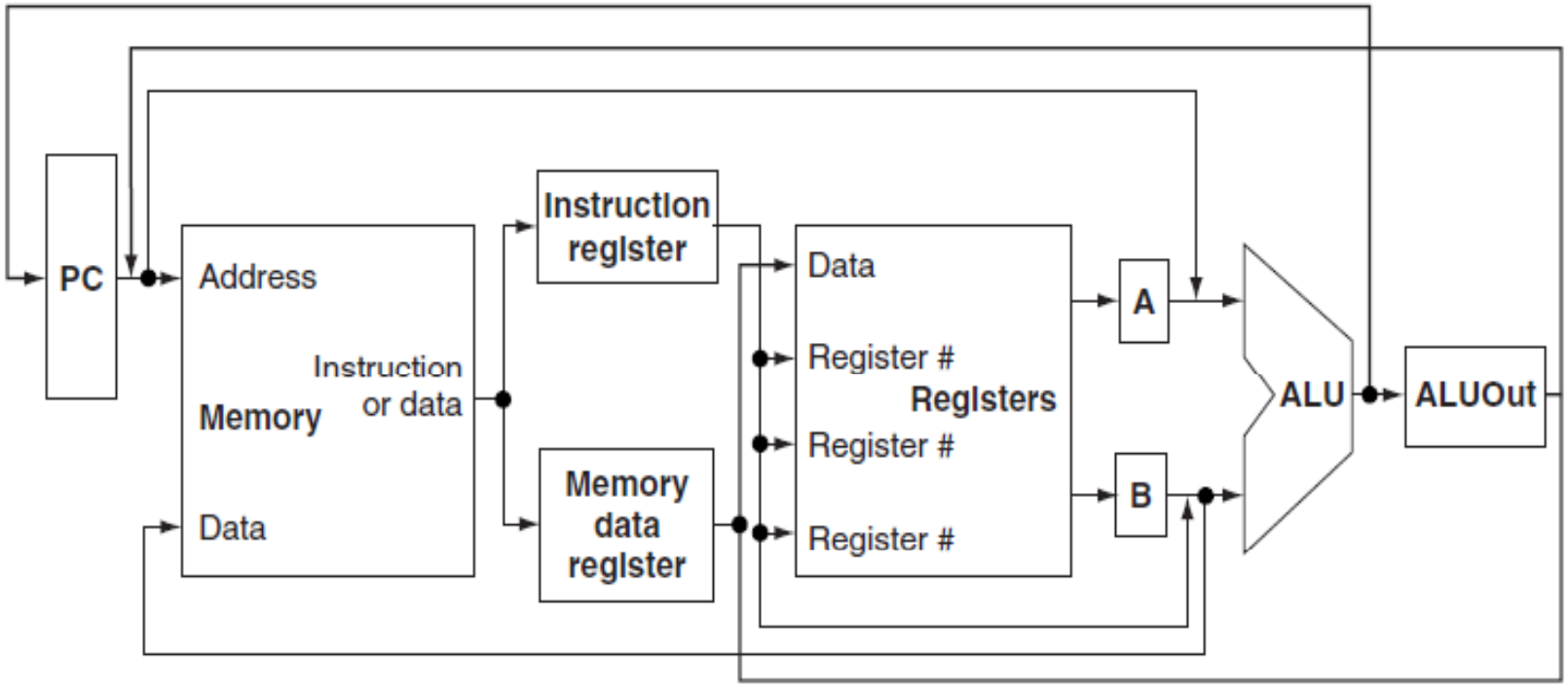


FIGURE 5.25 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

At the end of a clock cycle, all data that is used in subsequent clock cycles must be stored in a state element. Data used by *subsequent instructions* in a later clock cycle is stored into one of the programmer-visible state elements: the register file, the PC, or the memory. In contrast, data used by the *same instruction* in a later cycle must be stored into one of these additional registers.

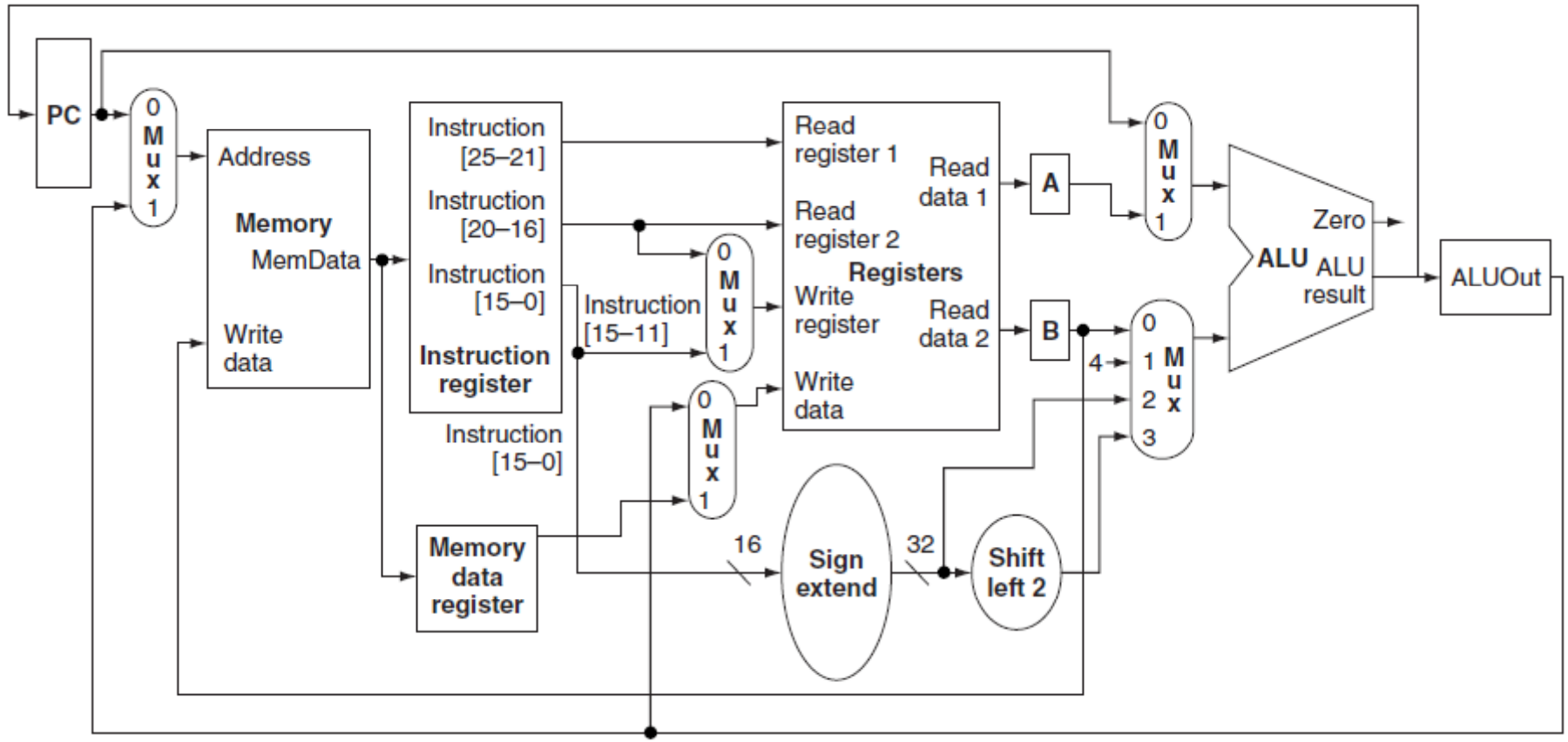


FIGURE 5.26 Multicycle datapath for MIPS handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several 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.

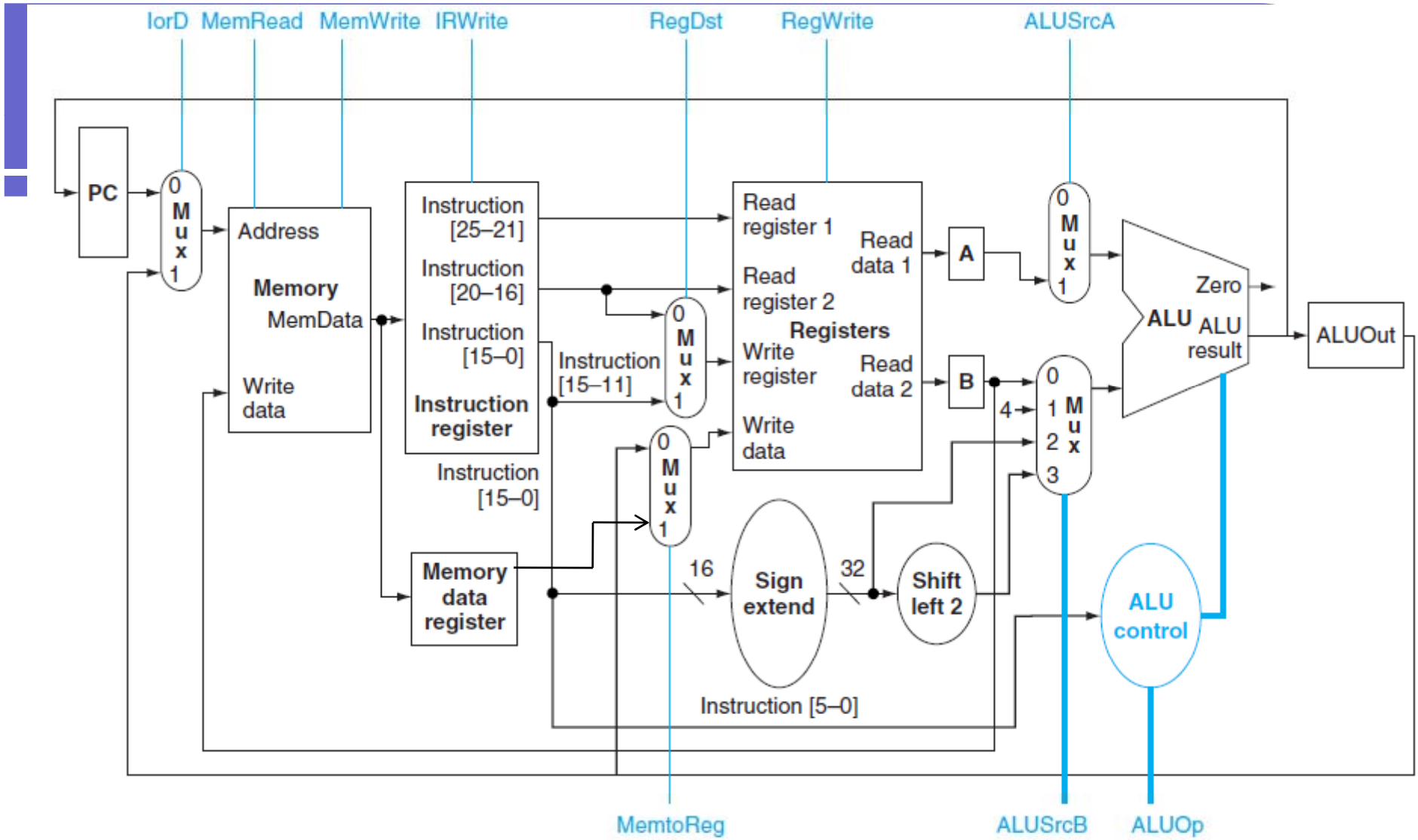


FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately 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.

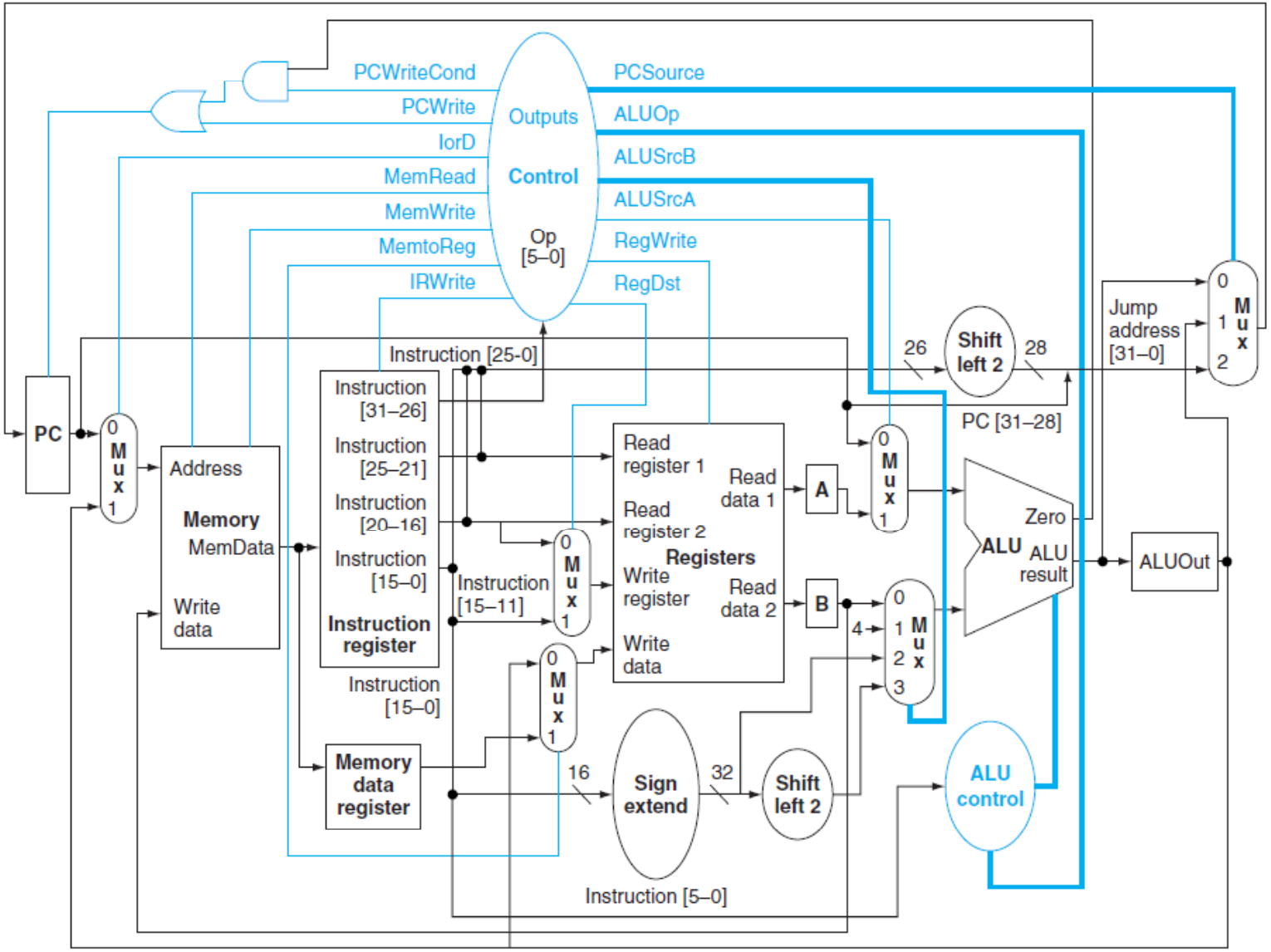


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

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.

FIGURE 5.29 The action caused by the setting of each control signal in Figure 5.28 on page 323. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSource) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

Breaking the Instruction Execution into Clock Cycles

- Instruction fetch step

IR <= Memory[PC];
PC <= PC + 4;

- Assert the control signals MemRead and IRWrite
- set IorD to 0 to select the PC as the source of the address
- Set the ALUSrcA signal to 0 (PC+4)
- Set the ALUSrcB signal to 01 (sending 4 to the ALU)
- Set ALUOp to 00 (to make the ALU add).
- Set PC source to 00 and set PCWrite (update PC).
- The increment of the PC and the instruction memory access can occur in parallel.
- The new value of the PC is not visible until the next clock cycle

- Instruction decode and register fetch

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend (IR[15-0]) << 2);
```

- Access the register file to read registers rs and rt and store the results into the registers A and B
  - A,B are overwritten on each cycle
- Compute the branch target address and stores the address in ALUOut
  - Set ALUSrcA to 0 (so that the PC is sent to the ALU),
  - Set ALUSrcB to the value 11 (so that the sign-extended and shifted offset field is sent to the ALU)
  - Set ALUOp as 00 (so the ALU adds).
  - The register file accesses and computation of branch target occur in parallel

- Execution, memory address computation, or branch completion
  - Memory reference:

ALUOut <= A + sign-extend (IR[15:0])

- ALUSrcA =1 (the first ALU input is register A)
- ALUSrcB = 10 (the output of the sign extension unit is used for the second ALU input)
- ALUOp =00 (causing the ALU to add)

  - Arithmetic-logical instruction (R type)

ALUOut <= A op B

- The ALU is performing the operation specified by the function code on the two values read from the register file in the previous cycle.
- ALUSrcA = 1 and ALUSrcB = 00, A and B to be used as the ALU inputs.
- ALUOp=10 (the funct field is used to determine the ALU control signal settings)

- Branch

if (A == B) PC <= ALUOut

- The ALU is used to do the equal comparison between the two registers read in the previous step.
- The Zero signal out of the ALU is used to determine whether or not to branch.
- ALUSrcA = 1 and ALUSrcB = 00 (the register file outputs are the ALU inputs).
- ALUOp=01 (causing the ALU to subtract) for equality testing.
- The PCWriteCond is asserted to update the PC if the Zero output of the ALU is asserted.
- PCSource =01(the value written into the PC will come from ALUOut, which holds the branch target address computed in the previous cycle).
- For conditional branches that are taken, we actually write the PC twice: once from the output of the ALU (during the Instruction decode/register fetch) and once from ALUOut (during the Branch completion step).
- The value written into the PC last is the one used for the next instruction fetch.

- Jump

PC <= {PC [31:28], (IR[25:0]],2'b00)}

- The PC is replaced by the jump address.
- PCSource is set to direct the jump address to the PC
- PCWrite is asserted to write the jump address into the PC

- Memory access or R-type instruction completion step
  - Memory reference:

MDR <= Memory [ALUOut] or

Memory [ALUOut] <= B

- If the instruction is a load, a data word is retrieved from memory and is written into the MDR. If the instruction is a store, then the data is written into memory.
- For a store, the source operand is saved in B
- The signal MemRead (for a load) or MemWrite (for store) will need to be asserted.
- For loads and stores, IorD=1 (force the memory address to come from the ALU)
- MDR is written on every clock cycle, no explicit control signal need be asserted.

- Arithmetic-logical instruction (R-type):

Reg[IR[15:11]] <= ALUOut

- Place the contents of ALUOut, which corresponds to the output of the ALU operation in the previous cycle, into the Result register
- RegDst=1 (force the rd field (bits 15:11) to be used to select the register file entry to write)
- RegWrite must be asserted, and MemtoReg must be set to 0 so that the output of the ALU is written, as opposed to the memory data output.

- Memory read completion step
  - Load:

Reg[IR[20:16]] <= MDR

- Write the load data, which was stored into MDR in the previous cycle, into the register file.
- MemtoReg = 1 (to write the result from memory), assert RegWrite (to cause a write),
- RegDst = 0 (choose the rt (bits 20:16) field as the register number)

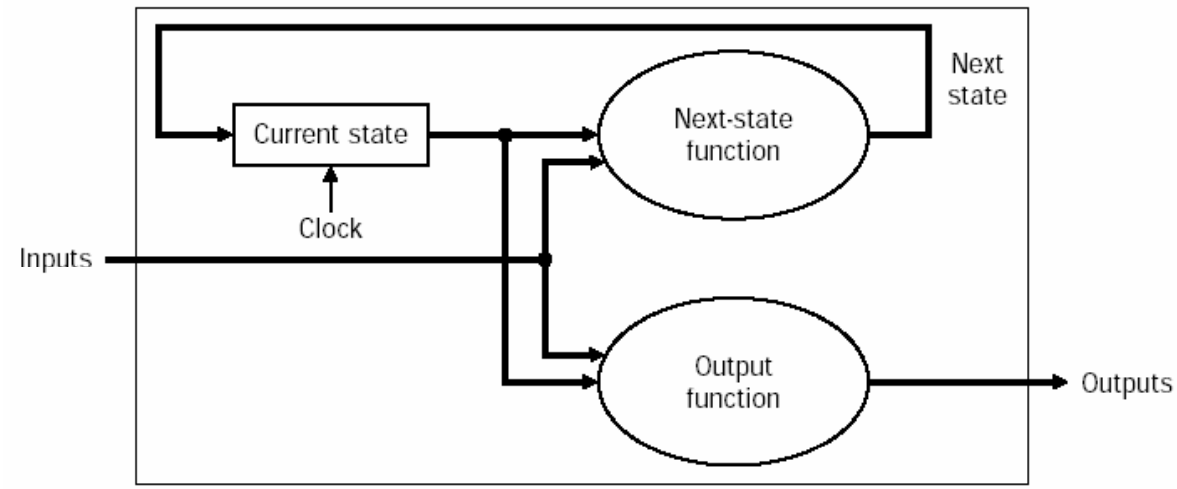
Breaking the Instruction Execution into Clock Cycles

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Multicycle Approach

- Use Finite State Machine For Control
  - A set of states
  - Next state function (determined by current state and the input)
  - Output function (determined by current state and possibly input)



  - Use a Moore machine (output based only on current state)
  - To derive FSM, need steps of the instruction!

Multicycle Approach

- Implementing the Control
  - Values of control signals are dependent upon:
    - 1. instruction is being executed
    - 2. step is being performed
  - Using steps, specify a finite state machine
    - specify the finite state machine graphically, or
    - use “microprogramming” (symbolic representation of control in the form of instructions)
  - Implementation can be derived from specification
    - Graphical Specification of FSM

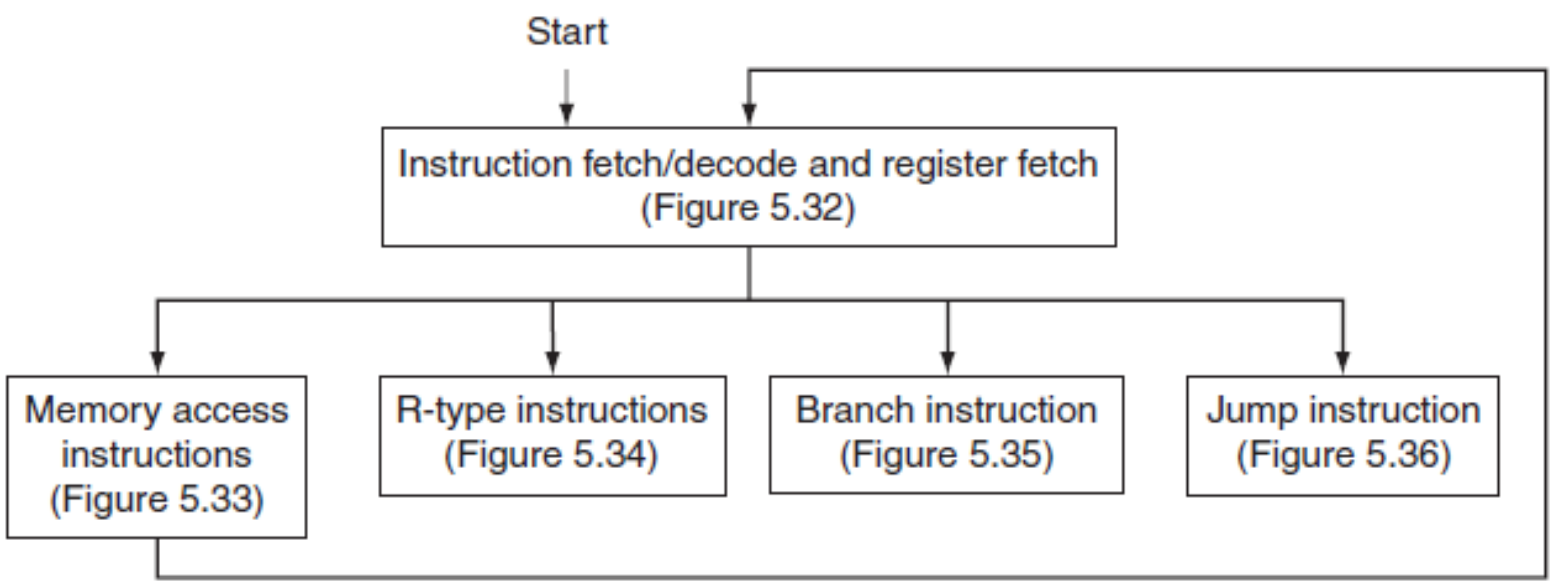


FIGURE 5.31 The high-level view of the finite state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

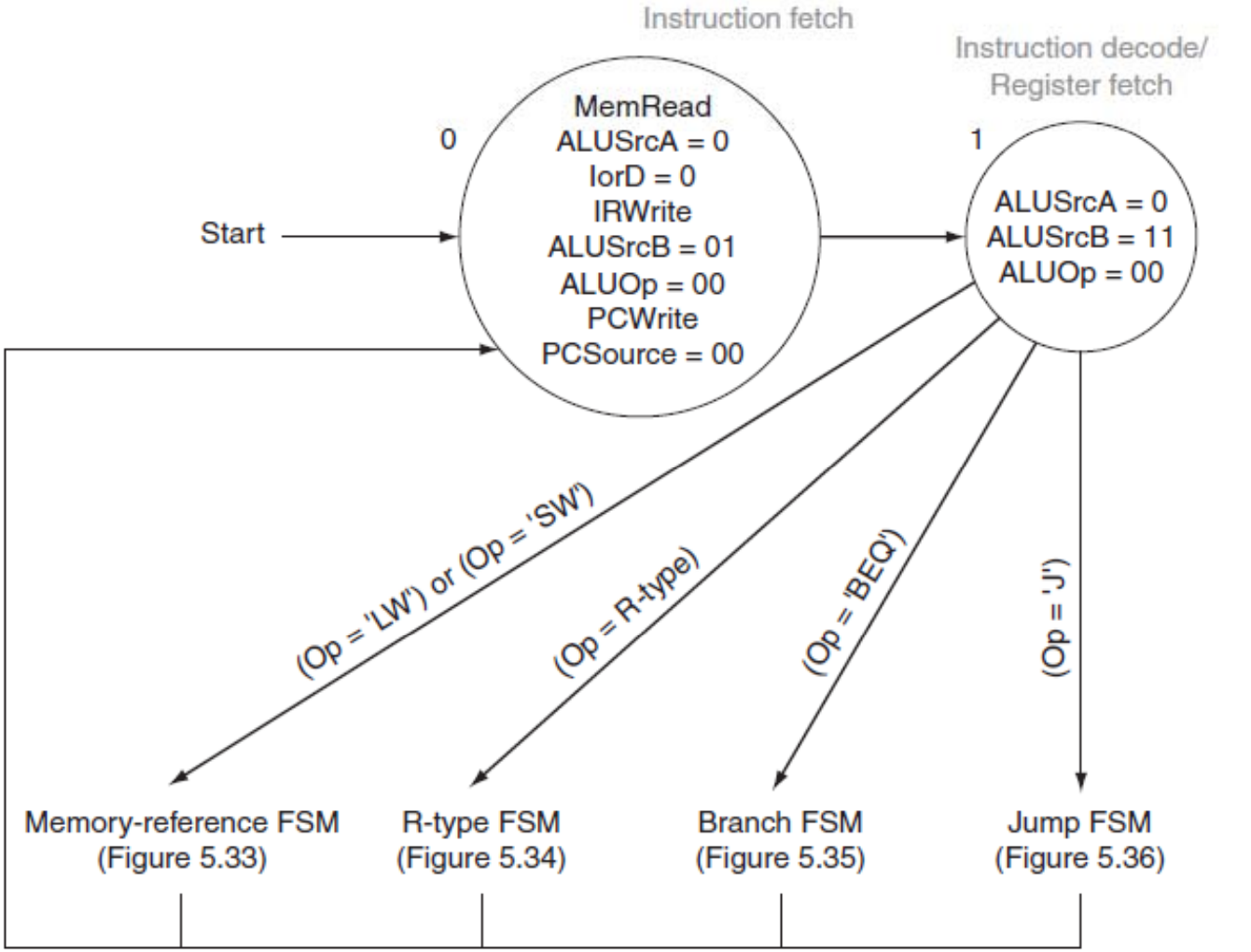


FIGURE 5.32 The instruction fetch and decode portion of every instruction is identical. These states correspond to the top box in the abstract finite state machine in Figure 5.31. In the first

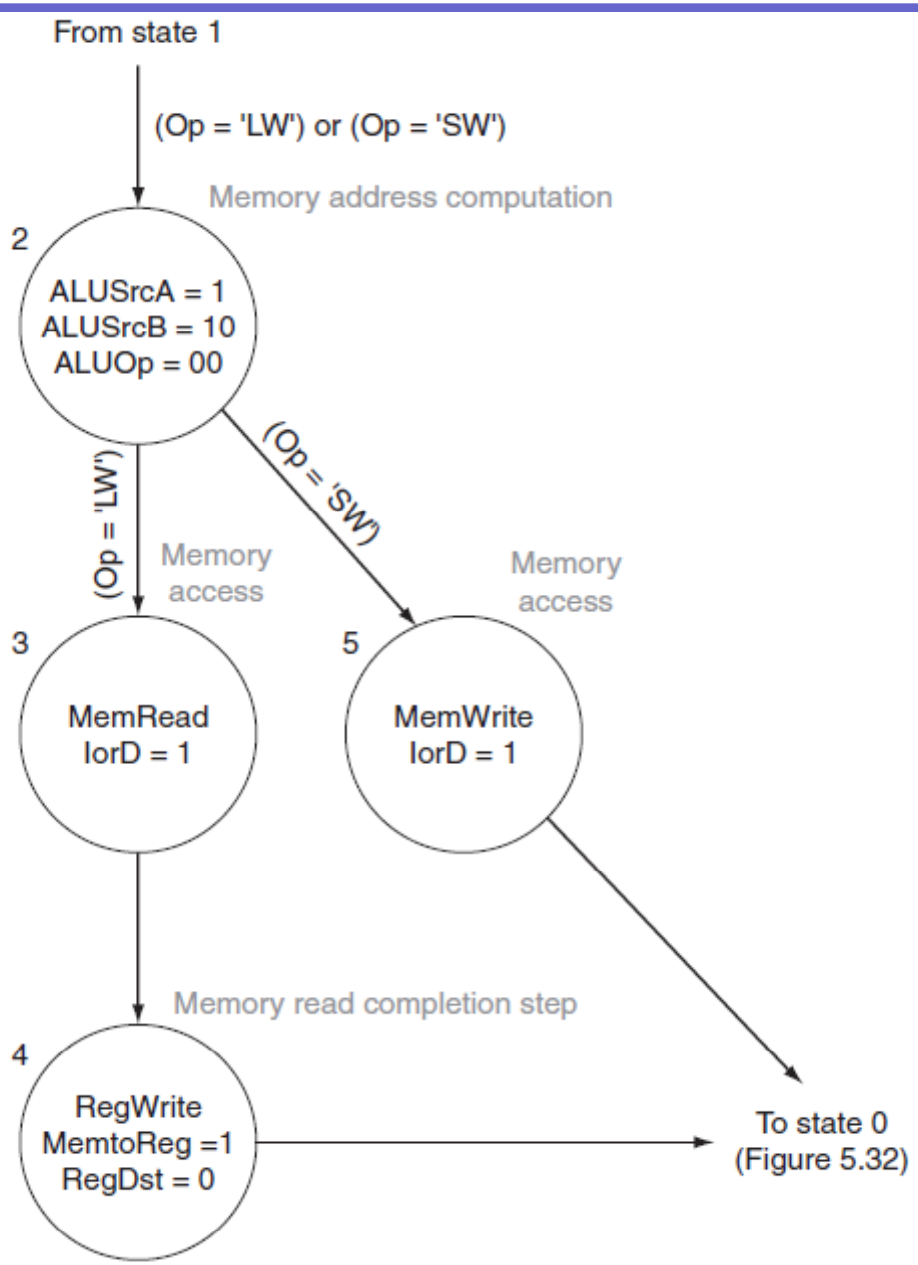


FIGURE 5.33 The finite state machine for controlling memory-reference instructions has four states. These states correspond to the box labeled "Memory access instructions" in Figure 5.31.

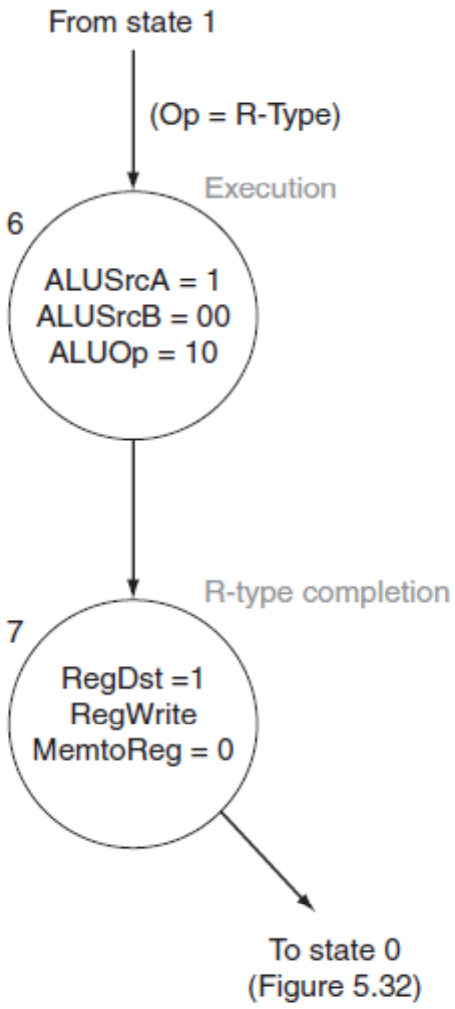


FIGURE 5.34 R-type instructions can be implemented with a simple two-state finite state machine. These states correspond to the box labeled "R-type instructions" in Figure 5.31. The first

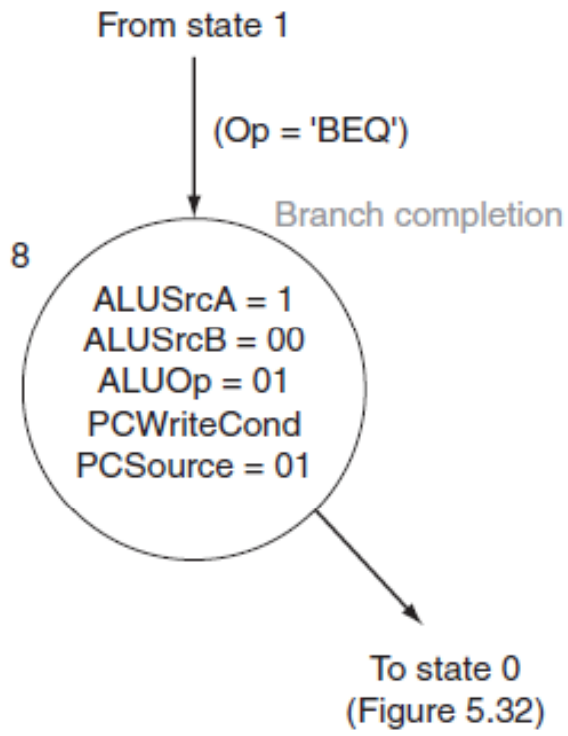


FIGURE 5.35 The branch instruction requires a single state. The first three outputs that are

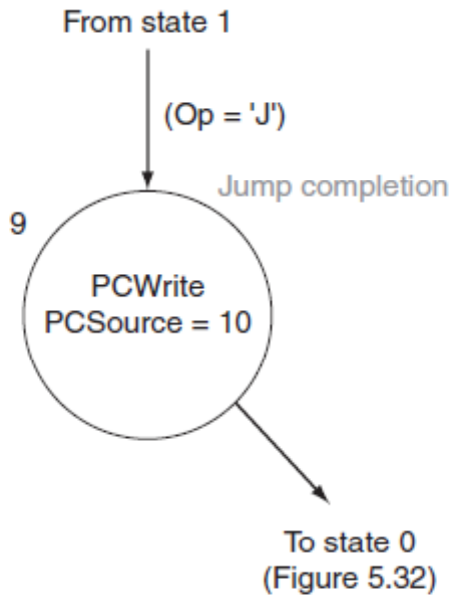


FIGURE 5.36 The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the Instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.

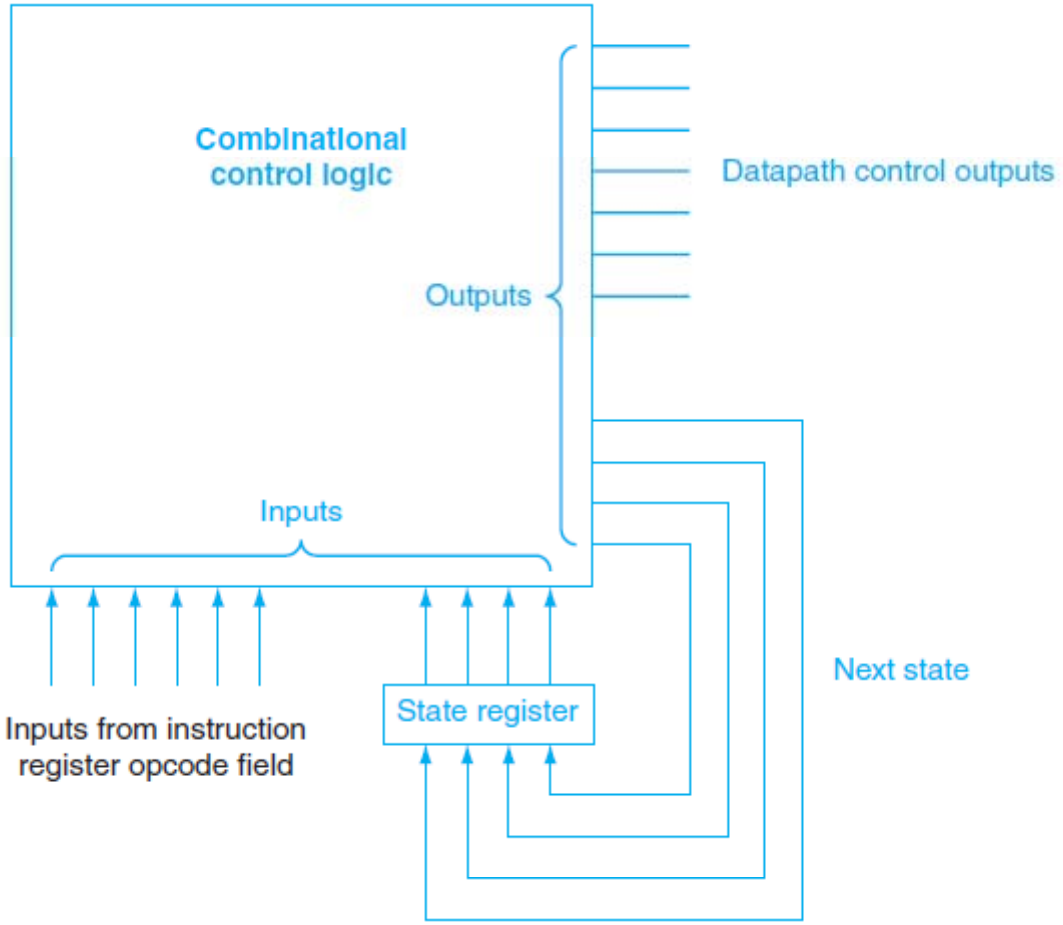


FIGURE 5.37 Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational

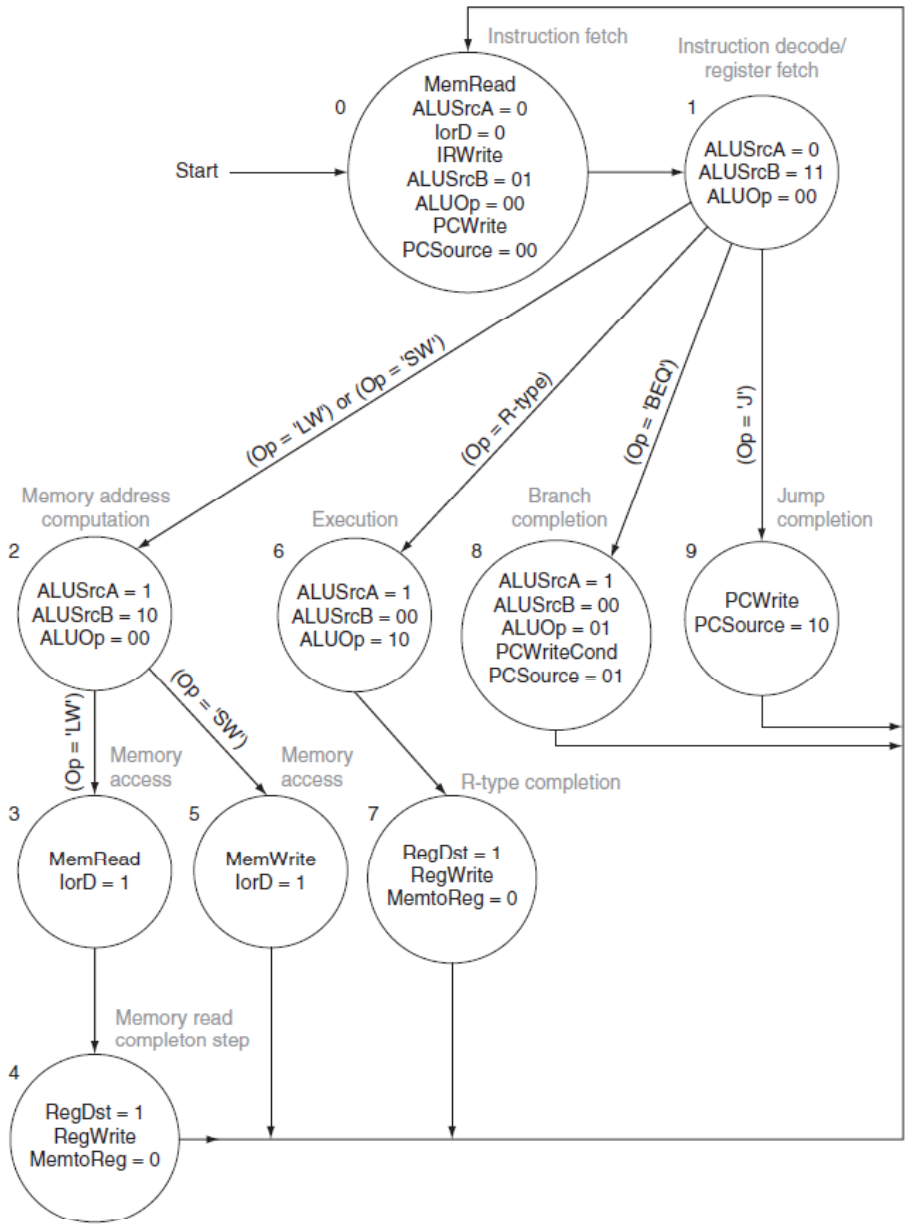


FIGURE 5.38 The complete finite state machine control for the datapath shown in Figure 5.28. The labels on the arcs are conditions that are tested to determine which state is the next state;

Exceptions

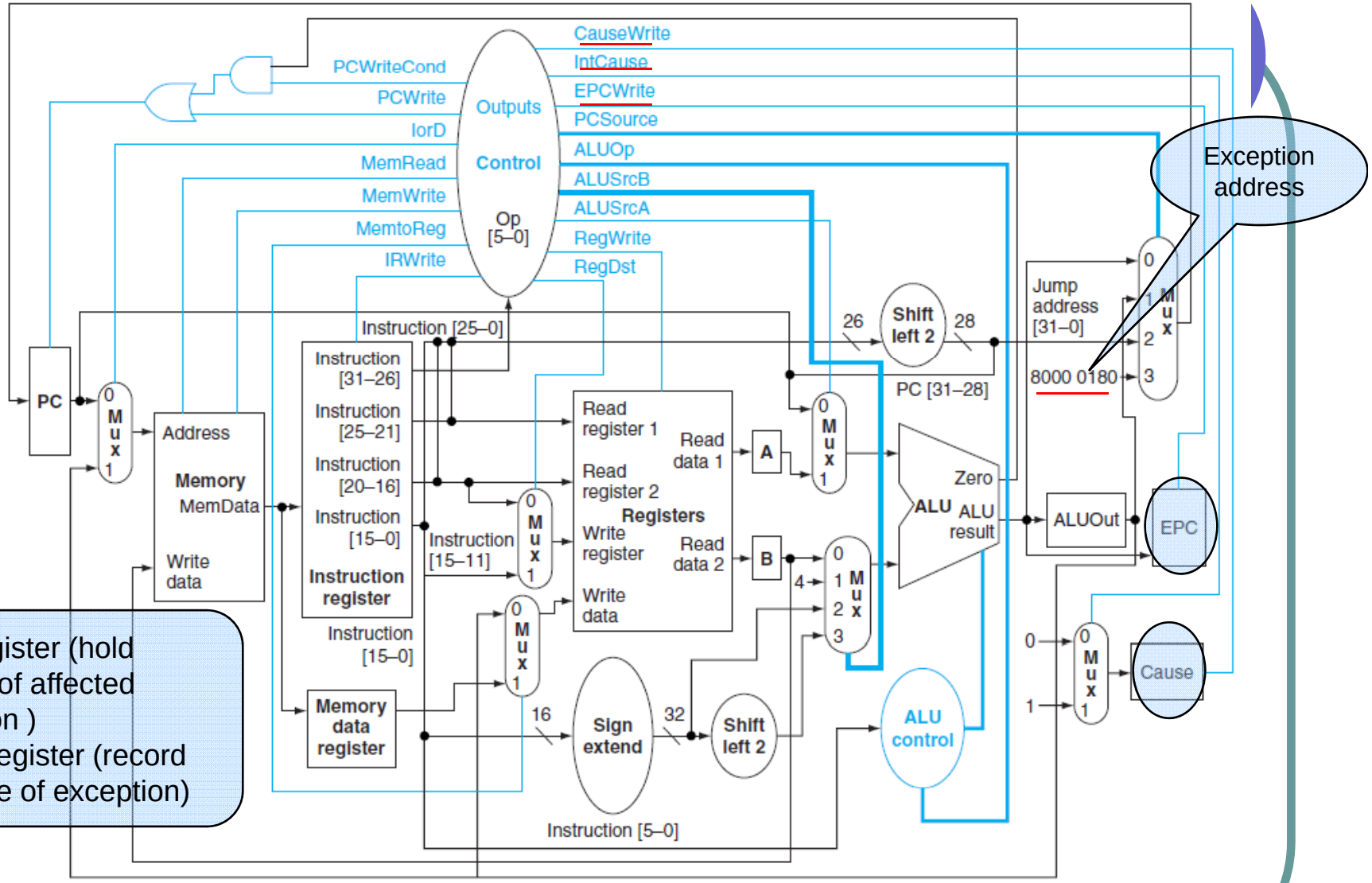
- **Exception:** An unscheduled event that disrupts program execution; used to detect overflow
- **Interrupt: An exception that** comes from outside of the processor

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

How Exceptions Are Handled

- The method used in the MIPS architecture is to include a status register which holds a field that indicates the reason for the exception.
- A second method is to use **vectored interrupts. In a vectored interrupt, the** address to which control is transferred is determined by the cause of the exception
- Need to add a few extra registers and control signals

Exception type	Exception vector address (in hex)
Undefined instruction	$C000\ 0000_{hex}$
Arithmetic overflow	$C000\ 0020_{hex}$



EPC: register (hold address of affected instruction)
Cause: register (record the cause of exception)

FIGURE 5.39 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers. For simplicity, this figure does not show the ALU overflow signal, which would need to be stored in a one-bit register and delivered as an additional input to the control unit (see Figure 5.40 to see how it is used).

How Control Checks for Exceptions

- *Undefined instruction: This is detected when no next state is defined from* state 1 for the op value.
- *Arithmetic overflow: The ALU includs logic to* detect overflow, and a signal called *Overflow is provided as an output from the* ALU.

How Exce

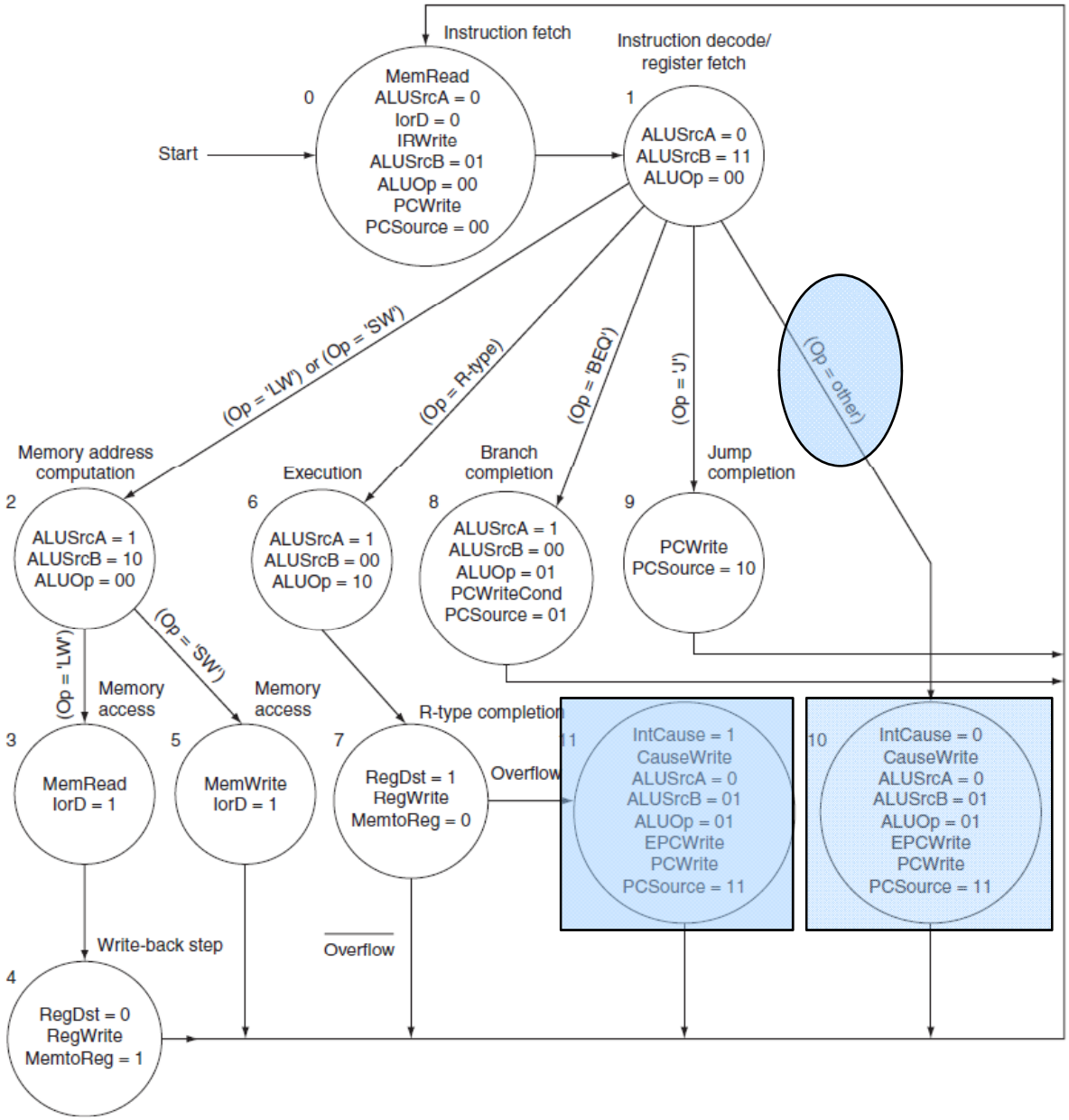


FIGURE 5.40 This shows the finite state machine with the additions to handle exception detection. States 10 and 11 are the new

Assessing performance

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

Performance of Single-Cycle Machines

- **CPU execution time = Instruction count × CPI × Clock cycle time**
- **CPU execution time = Instruction count × Clock cycle time (CPI=1)**

- Assume that the operation times for the major functional units in this implementation are the following:
- ■ Memory units: 200 picoseconds (ps)
- ■ ALU and adders: 100 ps
- ■ Register file (read or write): 50 ps

- assume the following instruction mix: 25% loads, 10% stores, 45% ALU instructions, 15% branches, and 5% jumps

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Thus, the average time per instruction with a variable clock is

$$\text{CPU clock cycle} = 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\%$$

$$= 447.5 \text{ ps}$$

$$\frac{\text{CPU performance}_{\text{variable clock}}}{\text{CPU performance}_{\text{single clock}}} = \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}}$$

$$= \frac{600}{447.5} = 1.34$$

- Implementing a variable-speed clock for each instruction class is extremely difficult, and the overhead for such an approach could be larger than any advantage gained

Multicycle CPI

- Assume 25% loads (1% load byte + 24% load word), 10% stores (1% store byte + 9% store word), 11% branches (6% beq, 5% bne), 2% jumps (1%jal + 1% jr), and 52% ALU
- number of clock cycles for each instruction class is: Loads: 5, Stores: 4 ALU instructions: 4, Branches: 3, Jumps: 3

The CPI is given by the following:

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}} = \frac{\sum \text{Instruction count}_i \times \text{CPI}_i}{\text{Instruction count}}$$

$$= \sum \frac{\text{Instruction count}_i}{\text{Instruction count}} \times \text{CPI}_i$$

The ratio

$$\frac{\text{Instruction count}_i}{\text{Instruction count}}$$

is simply the instruction frequency for the instruction class i. We can therefore substitute to obtain

$$\text{CPI} = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

Single vs. Multicycle

$$CPU\,time = \frac{Instructions}{Program} X \frac{Cycles}{Instruction} X \frac{Seconds}{Cycle}$$

Signal Cycle Datapath:	Multiple Cycle Datapath:
• **CPI = 1** ☺ • **Long cycle time L (critical path based)** ☹	• **Short cycle time** ☺ • **CPI = 3~5 L** ☹

Can we achieve a CPI of 1 (on average) with a clock cycle time similar to the multiple cycle datapath?

Thank you very much!