## Creating a Single Datapath

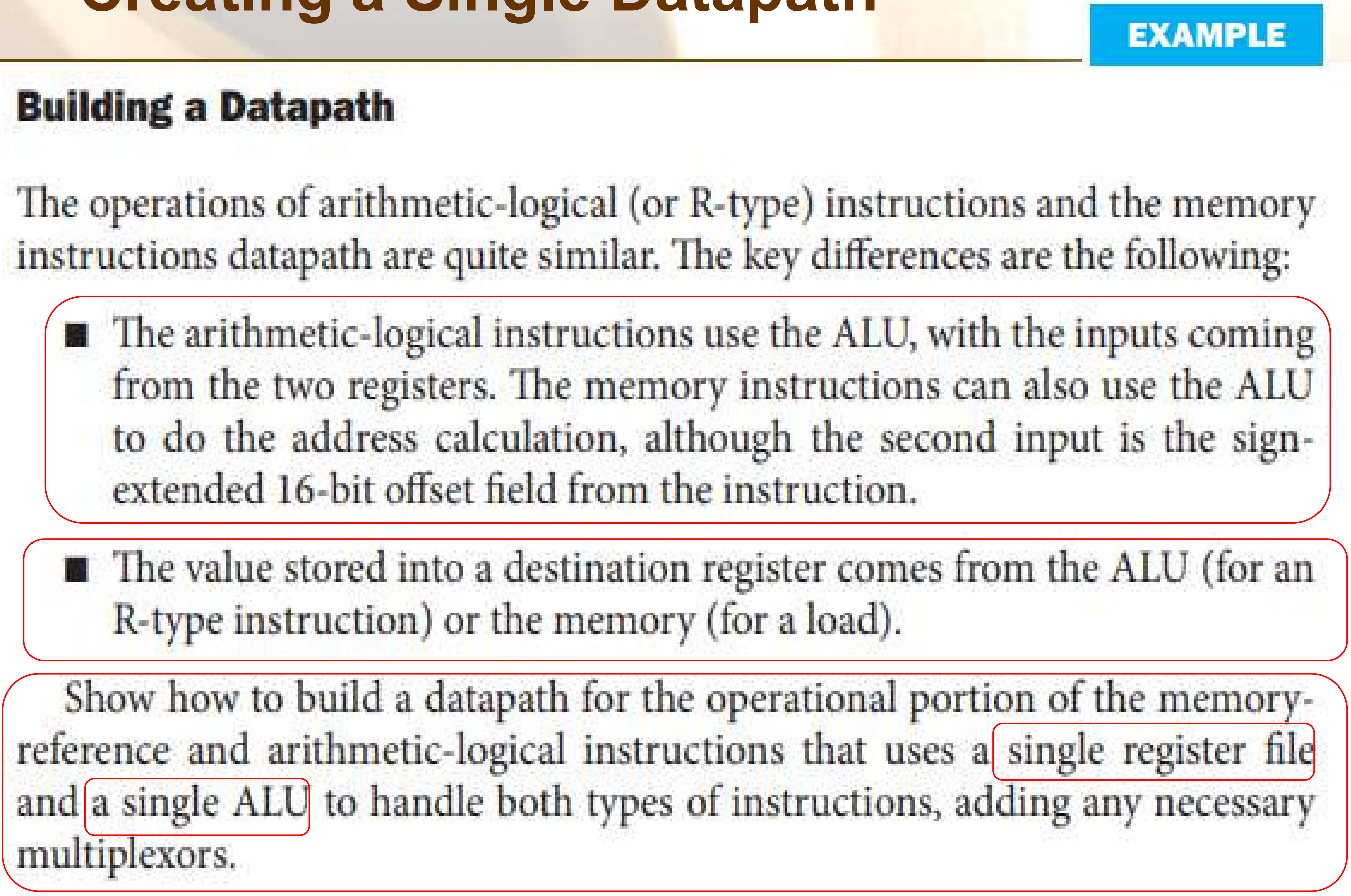
- We have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation.
- This simplest datapath will attempt to execute all instructions in one clock cycle.
- This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated.
- We therefore need a memory for instructions separate from one for data.

## Creating a Single Datapath

- Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

- To share a datapath element between two different instruction classes, 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.

## Creating a Single Datapath



EXAMPLE

**Building a Datapath**

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- 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).

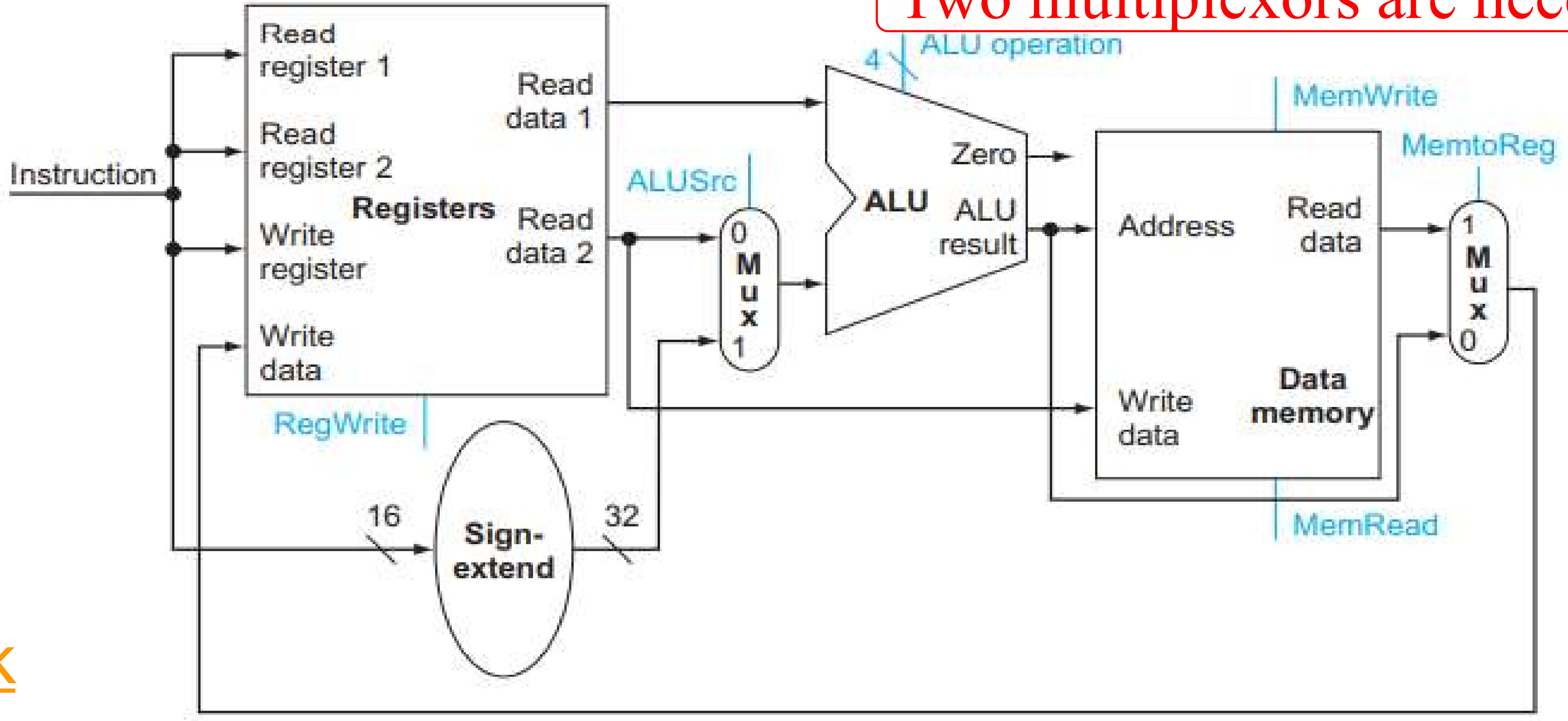
Show how to build a datapath for the operational portion of the memory-reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

## Creating a Single Datapath

**ANSWER**

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. Figure 10 shows the operational portion of the combined datapath.

Two multiplexors are needed



Back

**FIGURE 10 The datapath for the memory instructions and the R-type instructions.**

[J. Hennessy and D. Patterson, “Computer Organization and Design: The Hardware/Software Interface”, 5th Edition.]

## Creating a Single Datapath

- Now we can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch (Figure 6), the datapath from R-type and memory instructions (Figure 10), and the datapath for branches (Figure 9).
- Figure 11 shows the datapath we obtain by composing the separate pieces.
- The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder from Figure 9 for computing the branch target address.
- An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.

# Creating a Single Datapath

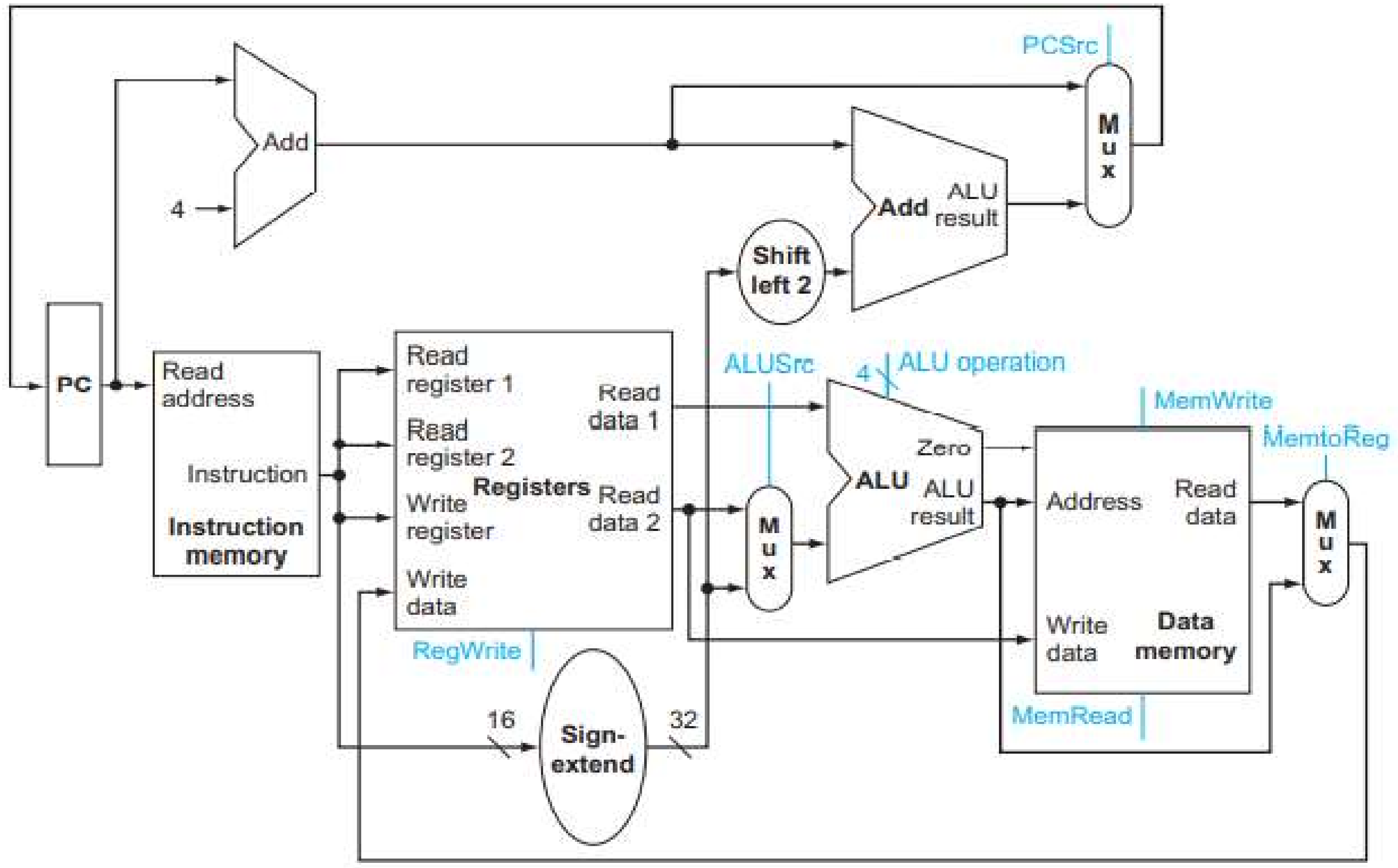


**FIGURE 11 The simple datapath for the core MIPS architecture combines the elements required by different instruction classes.**

[J. Hennessy and D. Patterson, “Computer Organization and Design: The Hardware/Software Interface”, 5th Edition.]

## Creating a Single Datapath with Control Unit

- Now that we have completed this simple datapath, we can add the control unit.
- The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control.
- The ALU control is different in a number of ways, and it will be useful to design it first before we design the rest of the control unit.

## Creating a Single Datapath with Control Unit

I. Which of the following is correct for a load instruction?

a. MemtoReg should be set to cause the data from memory to be sent to the register file

b. MemtoReg should be set to cause the correct register destination to be sent to the register file.

c. We do not care about the setting of MemtoReg for loads.

II. The single-cycle datapath conceptually described must have separate instruction and data memories, because

a. the formats of data and instructions are different in MIPS, and hence different memories are needed.

b. having separate memories is less expensive.

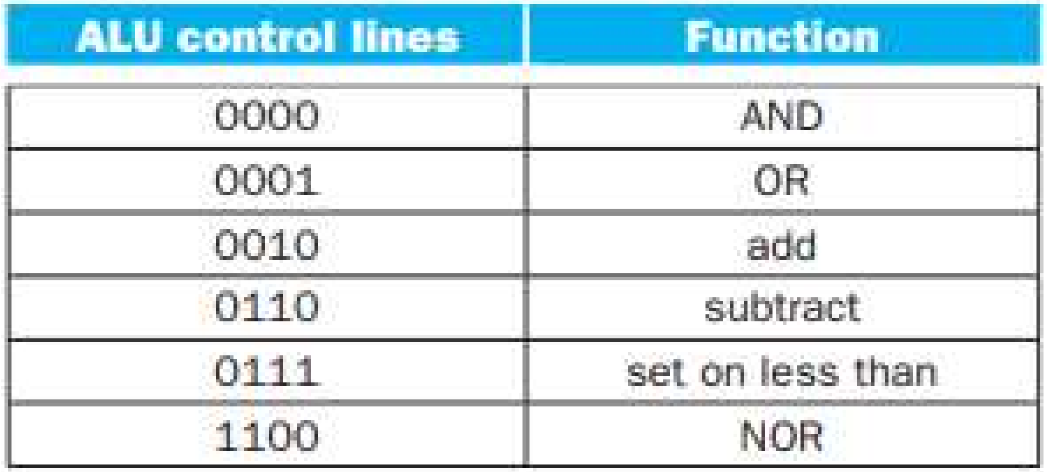
c. the processor operates in one cycle and cannot use a single-ported memory for two different accesses within that cycle.

## A Simple Implementation Scheme

- We build the simple implementation using the datapath and adding a simple control function.
- This simple implementation covers *load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than*.

The ALU Control

- The MIPS ALU defines the 6 functions by following combinations of four control inputs:



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

Back

## A Simple Implementation Scheme

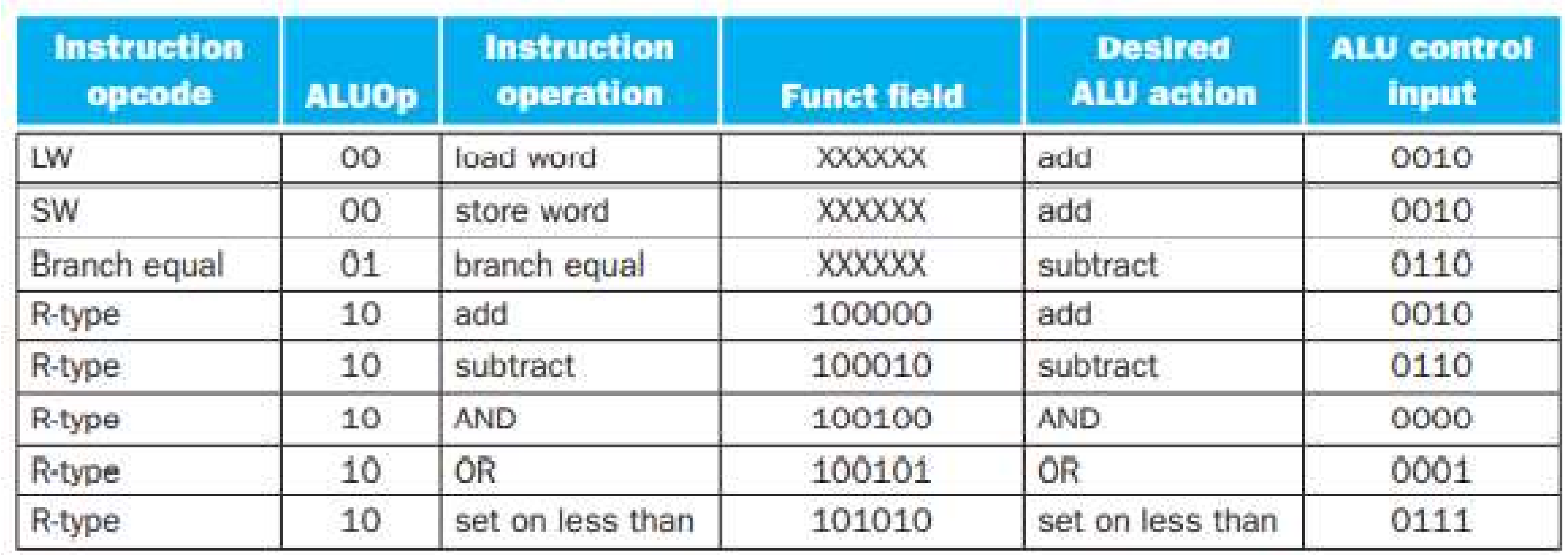
- Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR is needed for other parts of the MIPS instruction set not found in the subset we are implementing.)
- For load word and store word instructions, we use the ALU to compute the memory address by addition.
- For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction.
- For branch equal, the ALU must perform a subtraction.

## A Simple Implementation Scheme

- 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. (Slide 63)

## A Simple Implementation Scheme

- In Figure 12 shows how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code.



| 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 12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction.**

BACK2

BACK

## A Simple Implementation Scheme

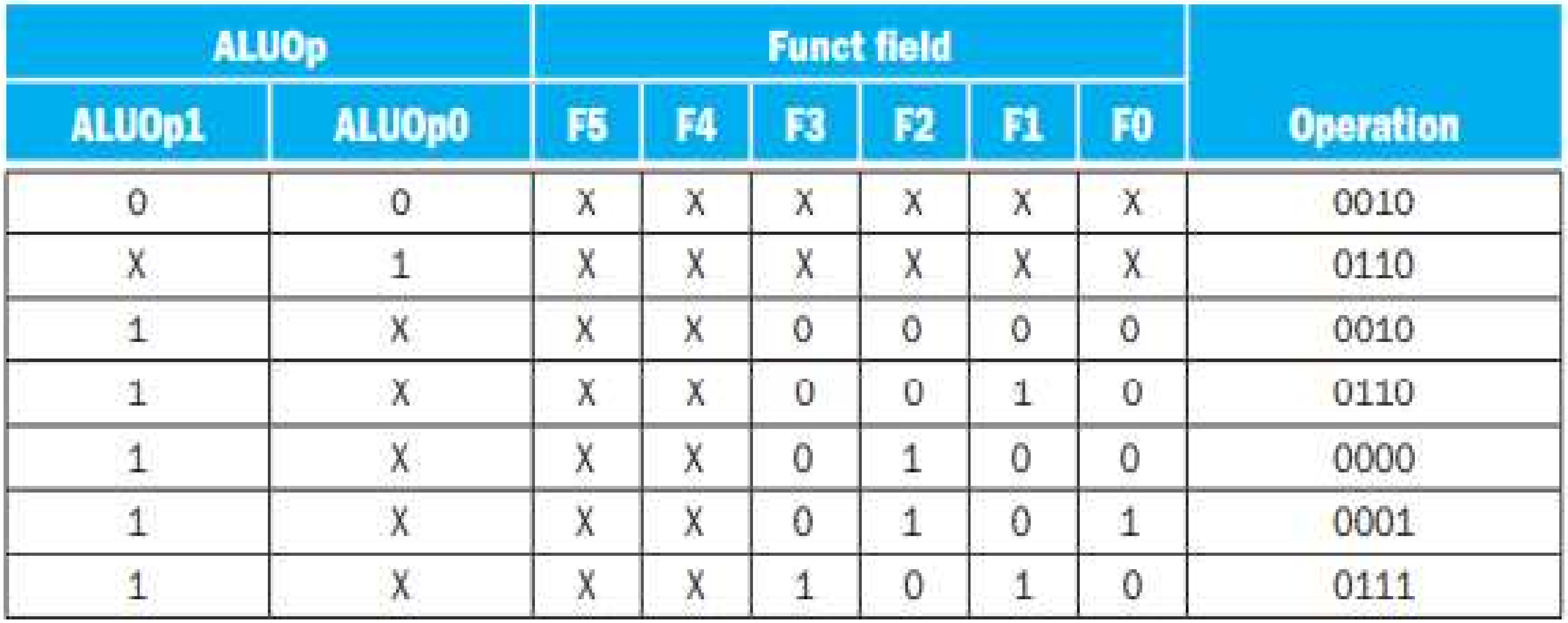
- This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique.
- Using multiple levels of control can reduce the size of the main control unit.
- Using several smaller control units may also potentially increase the speed of the control unit.
- Such optimizations are important, since the speed of the control unit is often critical to clock cycle time.

## A Simple Implementation Scheme

- There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the four ALU operation control bits.
- Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10(R-type), we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.

## A Simple Implementation Scheme

- As a step in designing this logic, it is useful to create a truth table for the interesting combinations of the function code field and the ALUOp bits (Figure 13).



| ALUOp | | Funct field | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | Operation |
| 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 13 The truth table for the 4 ALU control bits (called Operation).**

BACK

[J. Hennessy and D. Patterson, “Computer Organization and Design: The Hardware/Software Interface”, 5th Edition.]

## A Simple Implementation Scheme

- Truth table shows how the 4-bit ALU control is set depending on these two input fields.
- Since the full truth table is very large ($2^8 = 256$ entries) and we don’t care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value.

## A Simple Implementation Scheme

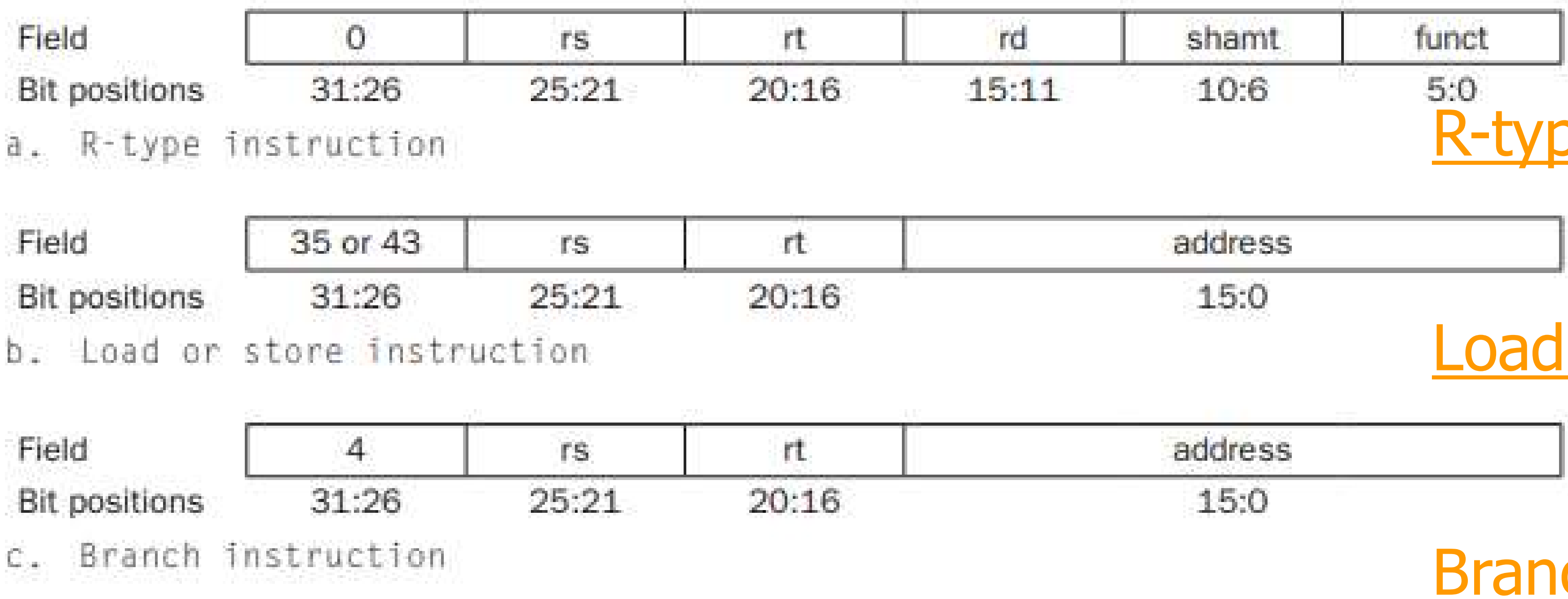
- Because in many instances we do not care about the values of some of the inputs, and because we wish to keep the tables compact, we also include don’t-care terms.
- A don’t-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column.
- For example, when the ALUOp bits are 00, as in the first row of Figure 13, we always set the ALU control to 0010(add), independent of the function code.
- The function code inputs will be don’t cares in this line of the truth table.
- Once the truth table has been constructed, it can be optimized and then turned into gates. This process is completely mechanical.

# Designing the Main Control Unit

- To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load-store instructions.

Datapath

opcode



|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| 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

R-type

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

b. Load or store instruction

Load-store

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

c. Branch instruction

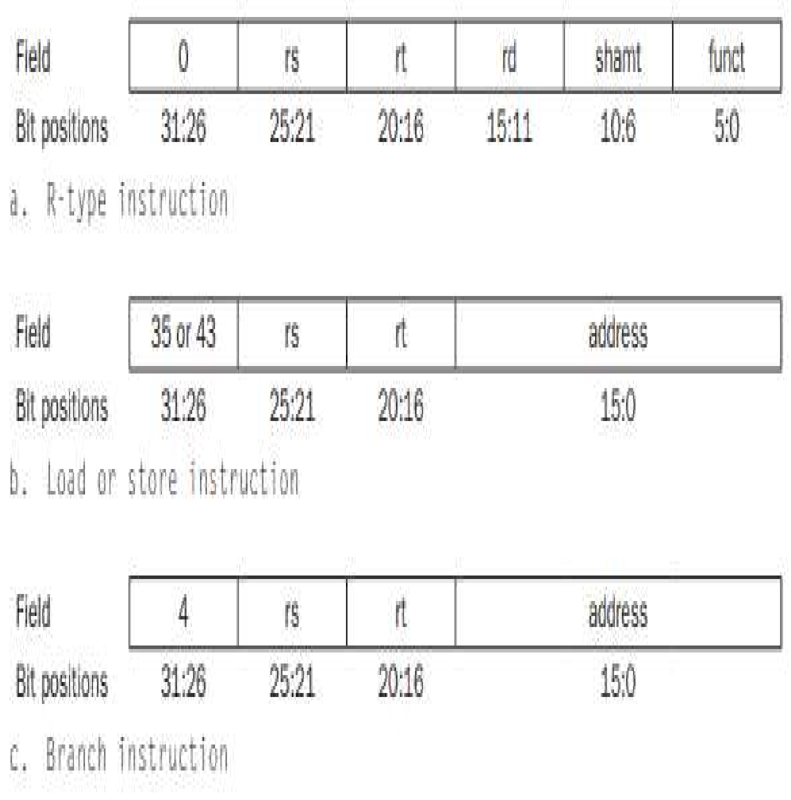
Branch

**FIGURE 14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.** [The shamt field is used only for shifts]

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

# Designing the Main Control Unit

- The op field is called the opcode, is always contained in bits 31:26. We will refer to this field as Op.
- The two registers to be read are always specified by the rs and rt fields at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The base register for load and store instructions is always in bit positions 25:21 (rs).



| 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

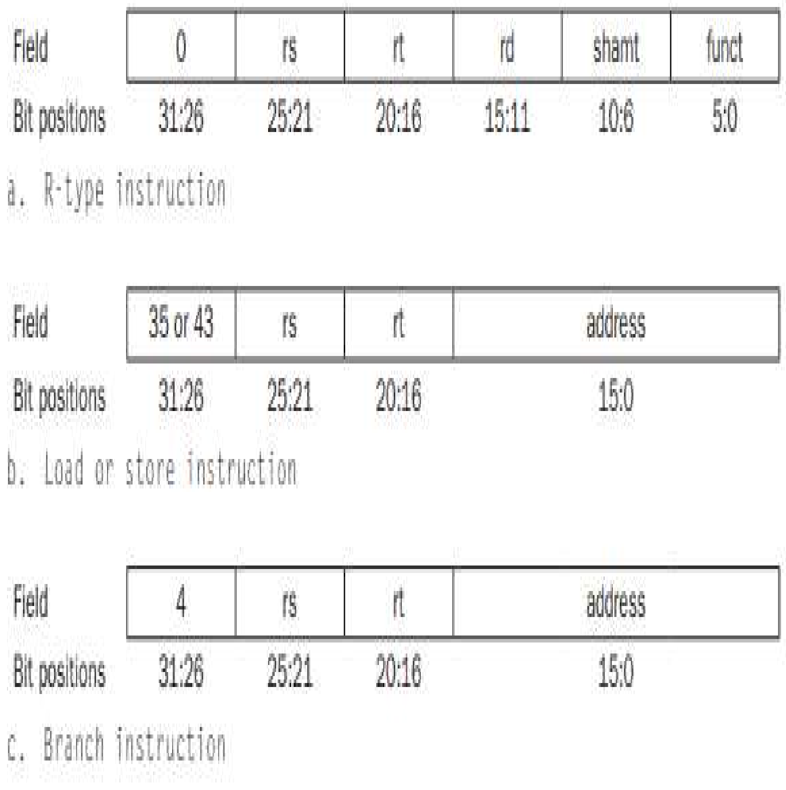
| Field | 4 | rs | rt | address |
|---|---|---|---|---|
| Bit positions | 31:26 | 25:21 | 20:16 | 15:0 |

c. Branch instruction

**FIGURE 14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.**

## Designing the Main Control Unit

- The 16-bit off set for branch equal, load and store is always in positions 15:0.
- The destination register is in one of two places.
  - For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd).
- Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written (destination register).



| 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 |

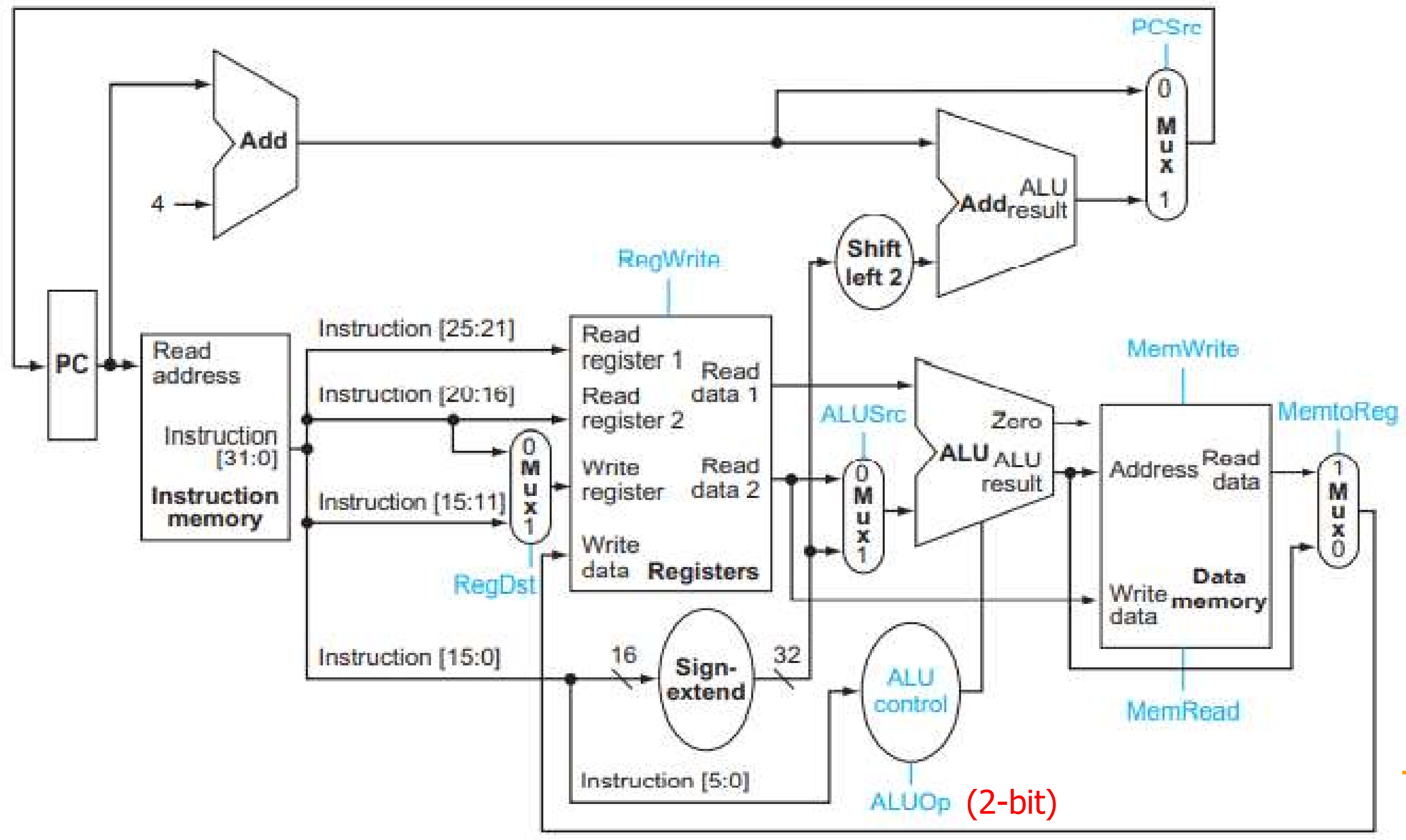
c. Branch instruction

FIGURE 14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.

## Designing the Main Control Unit

- Using this information, we can add the instruction labels and extra multiplexor (for the Write register number input of the register file) to the simple datapath.
- Figure 15 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors.
- Since all the multiplexors have two inputs, they each require a single control line.

## Designing the Main Control Unit



Back

**FIGURE 15 The datapath of Figure .11 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.

[J. Hennessy and D. Patterson, “Computer Organization and Design: The Hardware/Software Interface”, 5th Edition.]

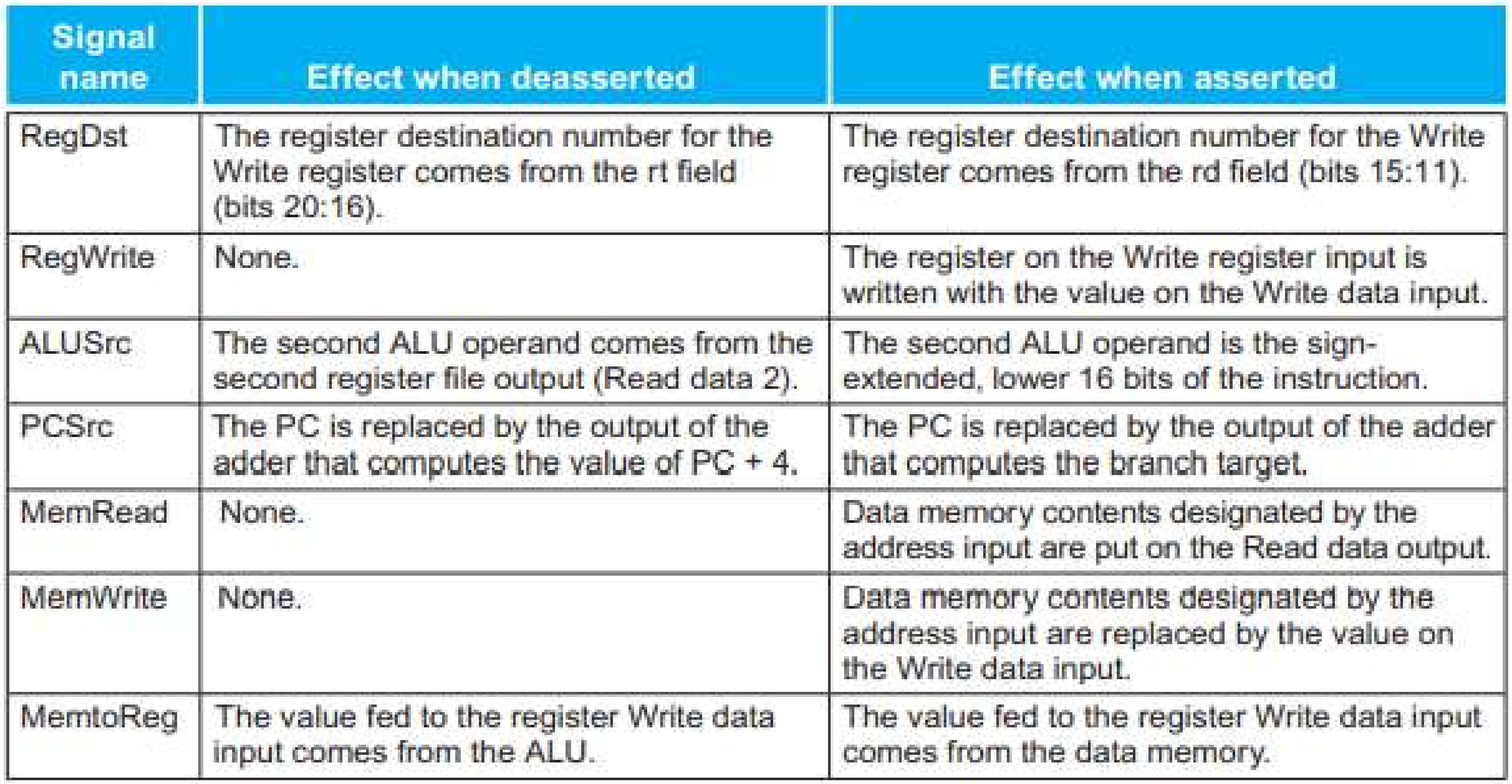
## Designing the Main Control Unit

- Figure 15 shows seven single-bit control lines plus the 2-bit ALUOp control signal.
- We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution.

ALUOp

## Designing the Main Control Unit

- Figure 16 describes the function 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. |

**FIGURE 16 The effect of each of the seven control signals.**

Refer

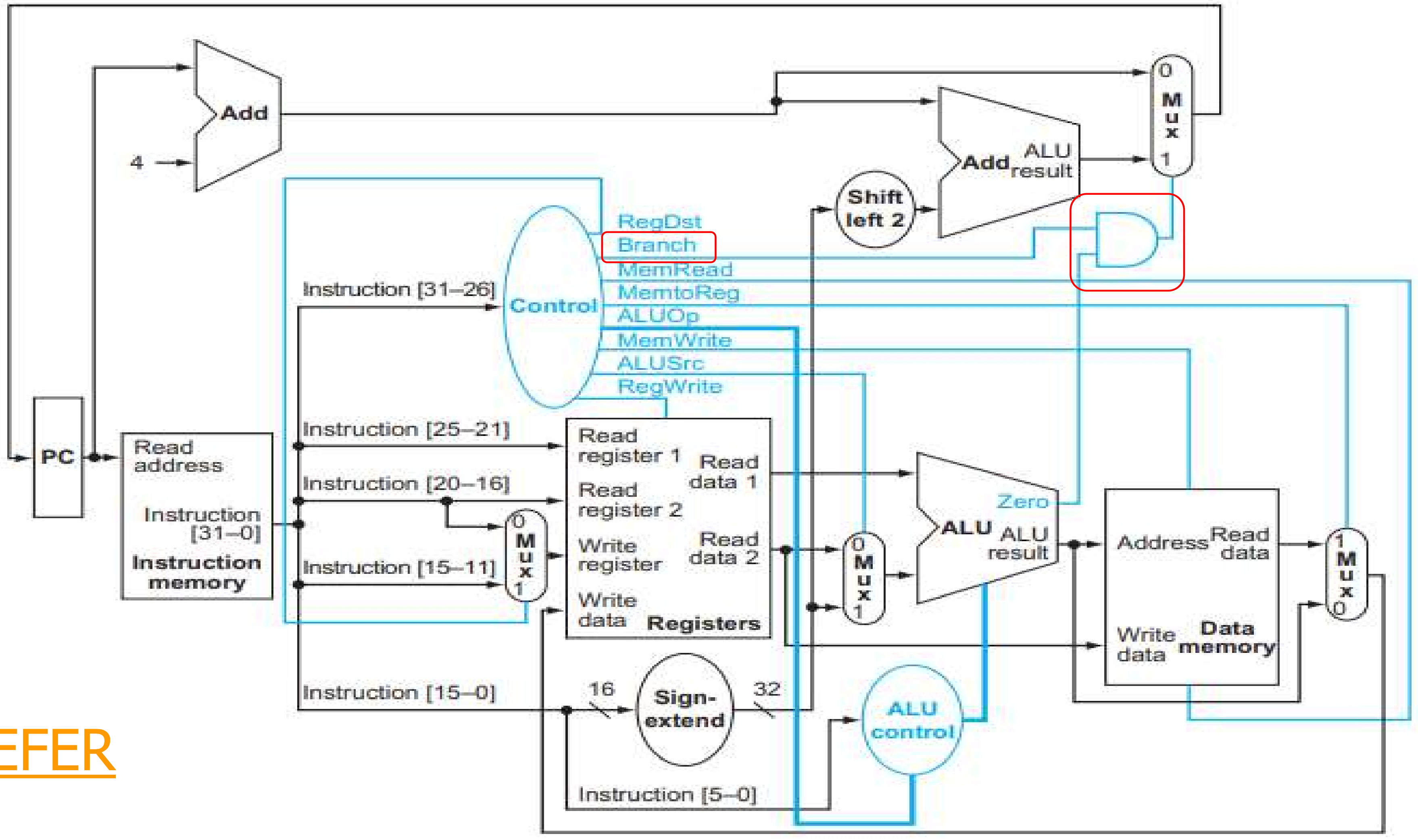
[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

## Designing the Main Control Unit

- The control unit can set all but one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception.
- PCSrc control line should be asserted if the instruction is branch on equal (a decision that the control unit can make) AND the Zero output of the ALU, which is used for equality comparison, is asserted.
- To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call Branch, with the Zero signal out of the ALU.
- These nine control signals (seven from Figure 16 and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits 31 to 26.

## Designing the Main Control Unit

- Figure 17 shows the datapath with the control unit and the control signals. [J. Hennessy and D. Patterson, “Computer Organization and Design: The Hardware/Software Interface”, 5th Edition.]

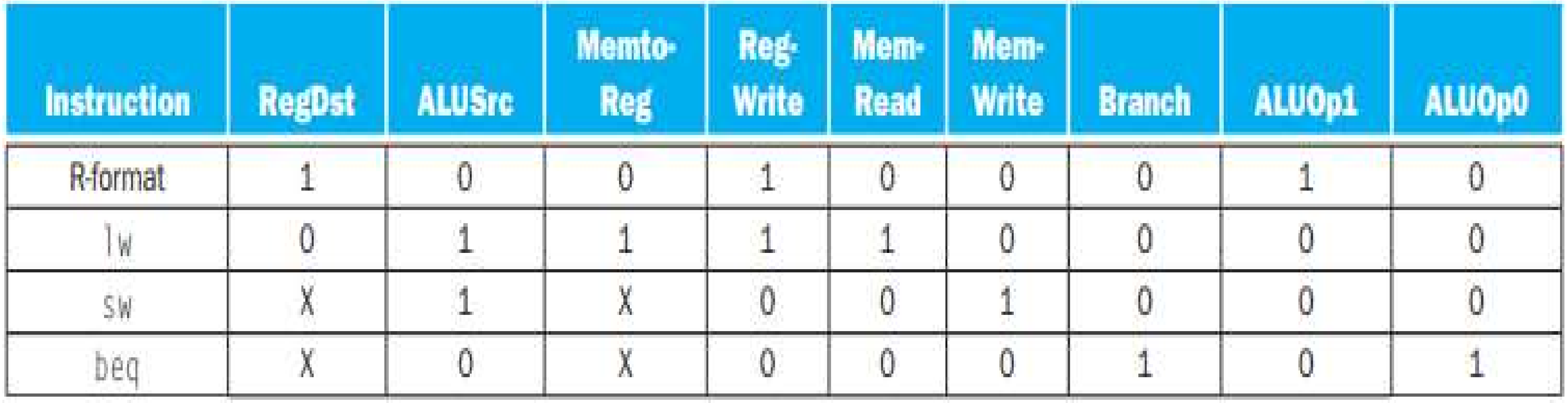


REFER

**FIGURE 17 The simple datapath with the control unit.**

# Designing the Main Control Unit

- Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X) for each of the opcode values.
- Figure18 defines how the control signals should be set for each opcode.



| 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 |

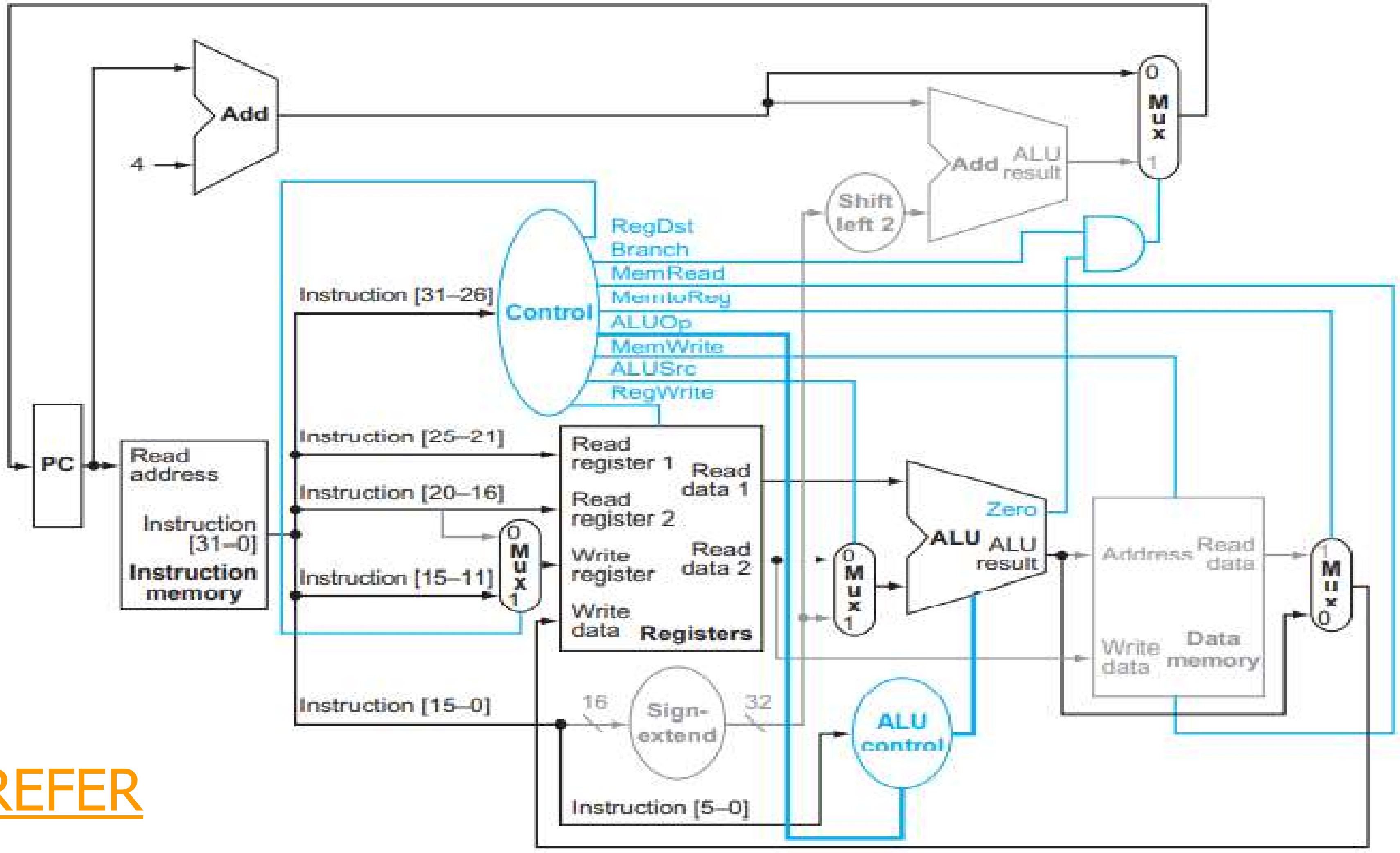
**FIGURE 18** The setting of the control lines is completely determined by the opcode fields of the instruction.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

# Operation of the Datapath R-type instruction

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Figure 19 shows the operation of the datapath for an R-type instruction, such as add $t1,$t2,$t3.



REFER

**FIGURE 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.

## Operation of the Datapath R-type instruction

add $t1,$t2,$t3

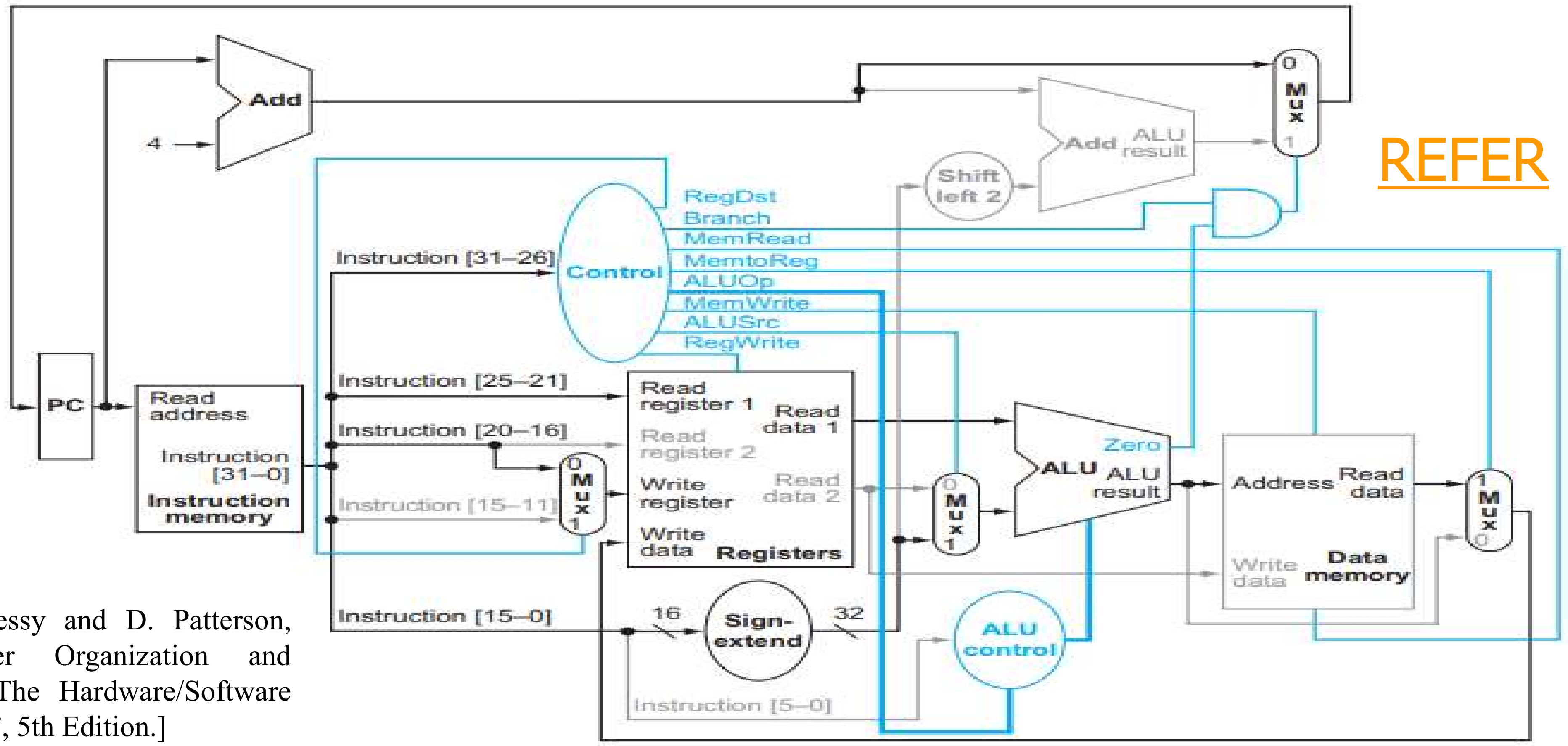
- Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

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

Refer

## Operation of the Datapath Memory reference instruction

- Similarly, we can illustrate the execution of a load word, such as *lw $t1, offset($t2)*
- Figure 20 shows the active functional units and asserted control lines for a load.



REFER

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

FIGURE 20 The datapath in operation for a load instruction.

## Operation of the Datapath

*lw $t1, offset($t2)*

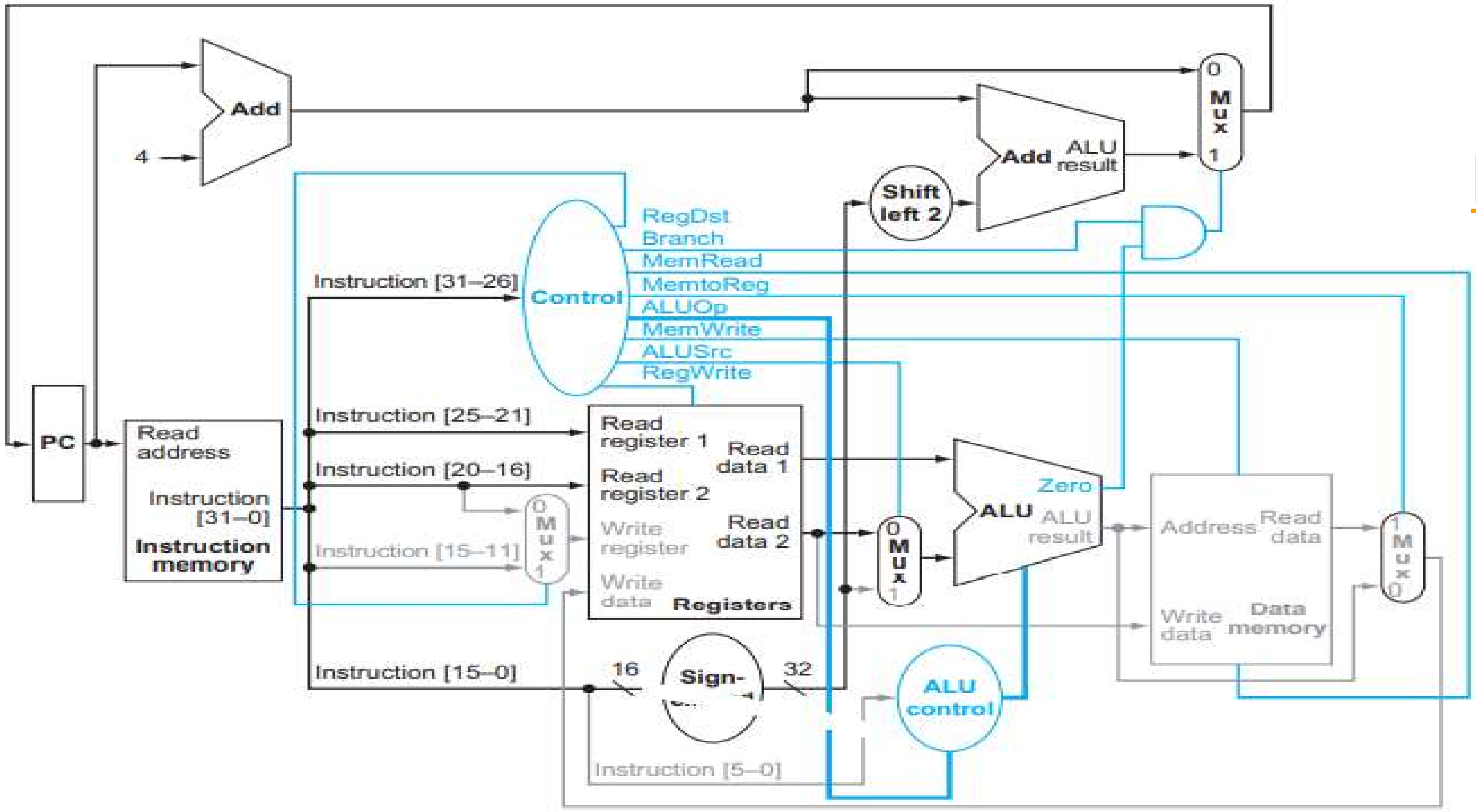
- We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):

  1. An instruction is fetched from the instruction memory, and the PC is incremented.
  2. A register ($t2) value is read from the register file.
  3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
  4. The sum from the ALU is used as the address for the data memory.
  5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

# Operation of the Datapath Branch instruction

- Finally, we can show the operation of the branch-on-equal instruction, such as *beq $t1, $t2, offset*, in the same fashion.
- It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]



REFER

**FIGURE 21 The datapath in operation for a branch-on-equal instruction.**

## Operation of the Datapath

*beq $t1, $t2, offset*

- Figure 21 shows the four steps in execution:

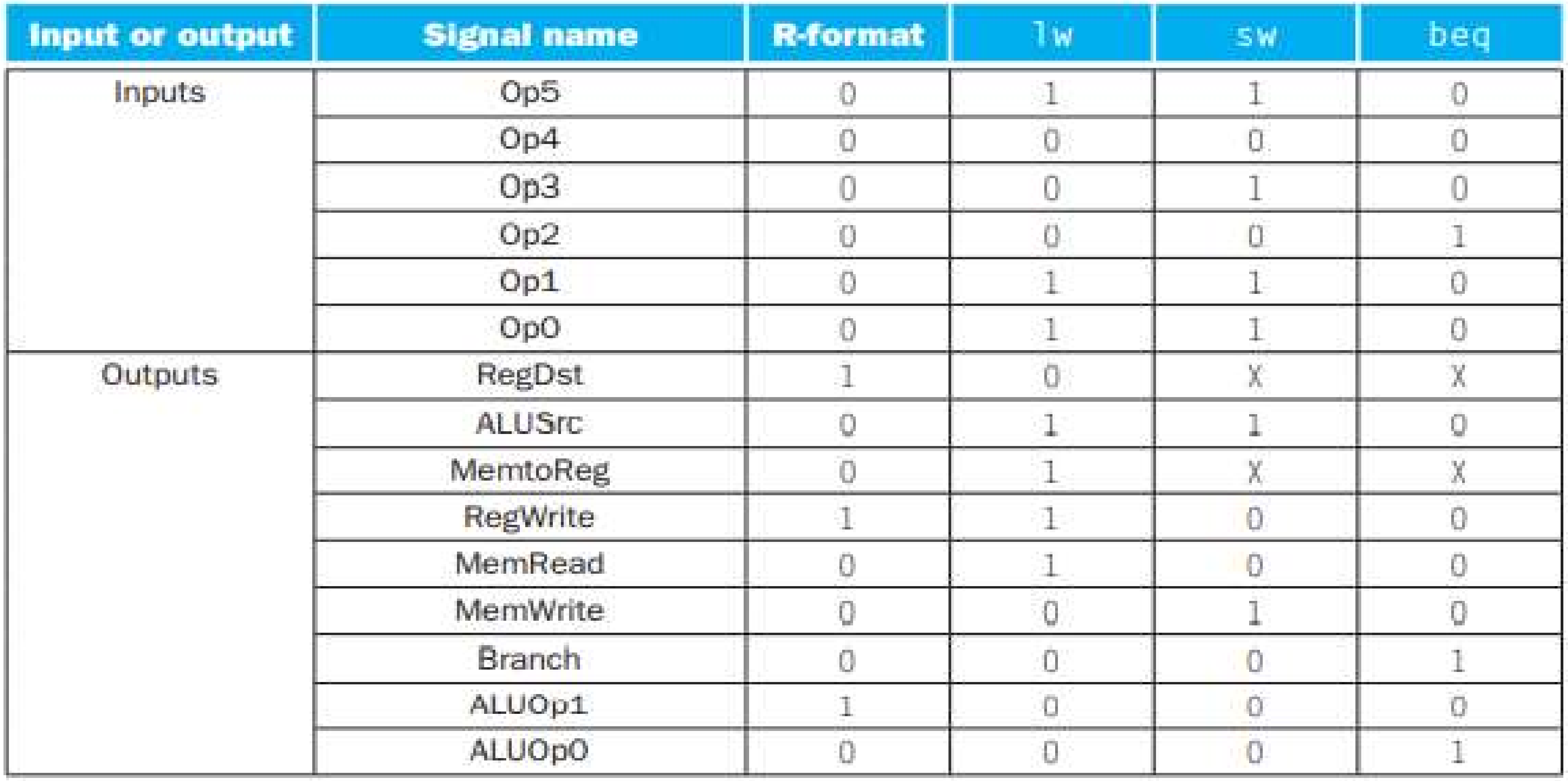
  1. An instruction is fetched from the instruction memory, and the PC is incremented.
  2. Two registers, $t1 and $t2, are read from the register file.
  3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the 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

## Finalizing Control

- The outputs are the control lines, and the input is the 6-bit opcode [31:26] field, Op [5:0].
- Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.
- Figure 22 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs.
- It completely specifies the control function, and we can implement it directly in gates in an automated fashion.

## Finalizing Control

OP VALUE - 35(LOAD) OR 43(STORE) OR 4(BRANCH EQ) OR 0 (R TYPE)



| 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 .22 The control function for the simple single-cycle implementation is completely specified by this truth table.**

[J. Hennessy and D. Patterson, “Computer Organization and Design: The Hardware/Software Interface”, 5th Edition.]

## Why a Single-Cycle Implementation Is Not Used Today

- Although the single-cycle design will work correctly, it would not be used in modern designs because it is inefficient.
- To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design.
- Of course, the longest possible path in the processor determines the clock cycle.
- This path is almost certainly a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file.
- Although the CPI is 1, the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

## Why a Single-Cycle Implementation Is Not Used Today

- The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set.
- Historically, early computers with very simple instruction sets did use this implementation technique.
- However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all.
- Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, it's useless to try implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time.

## Why a Single-Cycle Implementation Is Not Used Today

- A single cycle implementation thus violates the great idea of making the common case fast.
- In next section, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath but is much more efficient by having a much higher throughput.
- Pipelining improves efficiency by executing multiple instructions simultaneously.