diagram or a microprogram. In the latter case, each microinstruction represents a state. In an implementation using a finite state controller, the next-state function will be computed with logic. Section C.3 constructs such an implementation both for a ROM and a PLA.

An alternative method of implementation computes the next-state function by using a counter that increments the current state to determine the next state. When the next state doesn't follow sequentially, other logic is used to determine the state. Section C.4 explores this type of implementation and shows how it can be used for the finite state control created in Chapter 5.

In Section C.5, we show how a microprogram representation of sequential control is translated to control logic.

C.2 Implementing Combinational Control Units

In this section, we show how the ALU control unit and main control unit for the single clock design are mapped down to the gate level. With modern CAD systems this process is completely mechanical. The examples illustrate how a CAD system takes advantage of the structure of the control function, including the presence of don't-care terms.

Mapping the ALU Control Function to Gates

Figure C.2.1 shows the truth table for the ALU control function that was developed in Section 5.3. A logic block that implements this ALU control function will have three distinct outputs (called Operation2, Operation1, and Operation0), each corresponding to one of the three bits of the ALU control in the last column of Figure C.2.1. The logic function for each output is constructed by combining all the truth table entries that set that particular output. For example, the low-order bit of the ALU control (Operation0) is set by the last two entries of the truth table in Figure C.2.1. Thus the truth table for Operation0 will have these two entries.

Figure C.2.2 shows the truth tables for each of the three ALU control bits. We have taken advantage of the common structure in each truth table to incorporate additional don't cares. For example, the five lines in the truth table of Figure C.2.1 that set Operation1 are reduced to just two entries in Figure C.2.2. A logic minimization program will use the don't-care terms to reduce the number of gates and the number of inputs to each gate in a logic gate realization of these truth tables.

From the simplified truth table in Figure C.2.2, we can generate the logic shown in Figure C.2.3, which we call the *ALU control block*. This process is

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 C.2.1 The truth table for the three ALU control bits (called Operation) as a function of the ALUOp and function code field. This table is the same as that shown Figure 5.13.

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

a. The truth table for Operation2 = 1 (this table corresponds to the left bit of the Operation field in Figure C.2.1)

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

b. The truth table for Operation1 = 1

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

c. The truth table for Operation0 = 1

FIGURE C.2.2 The truth tables for the three ALU control lines. Only the entries for which the output is 1 are shown. The bits in each field are numbered from right to left starting with 0; thus F5 is the most significant bit of the function field, and F0 is the least significant bit. Similarly, the names of the signals corresponding to the 3-bit operation code supplied to the ALU are Operation2, Operation1, and Operation0 (with the last being the least significant bit). Thus the truth table above shows the input combinations for which the ALU control should be 010, 001, 110, or 111 (the combinations 011, 100, and 101 are not used). The ALUOp bits are named ALUOp1 and ALUOp0. The three output values depend on the 2-bit ALUOp field and, when that field is equal to 10, the 6-bit function code in the instruction. Accordingly, when the ALUOp field is not equal to 10, we don't care about the function code value (it is represented by an X). See Appendix B for more background on don't cares.

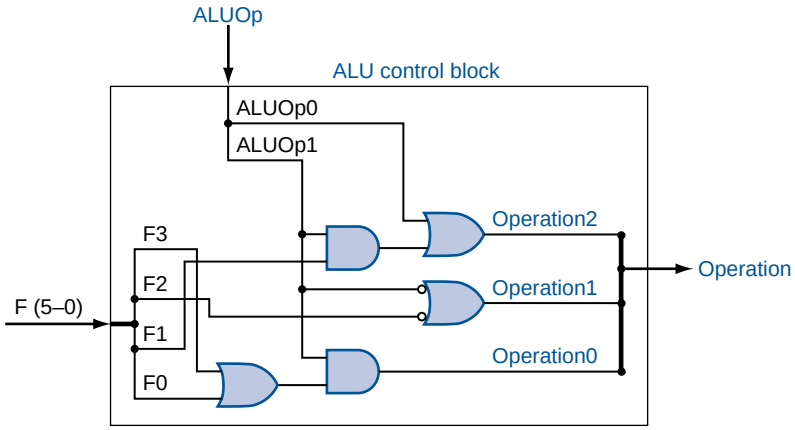

FIGURE C.2.3 The ALU control block generates the three ALU control bits, based on the function code and ALUOp bits. This logic is generated directly from the truth table in Figure C.2.2. Only 4 of the 6 bits in the function code are actually needed as inputs, since the upper 2 bits are always don't cares. Let's examine how this logic relates to the truth table of Figure C.2.2. Consider the Operation2 output, which is generated by two lines in the truth table for Operation2. The second line is the AND of two terms (F1 = 1 and ALUOp1 = 1); the top two-input AND gate corresponds to this term. The other term that causes Operation2 to be asserted is simply ALUOp0. These two terms are combined with an OR gate whose output is Operation2. The outputs Operation0 and Operation1 are derived in similar fashion from the truth table.

straightforward and can be done with a computer-aided design (CAD) program. An example of how the logic gates can be derived from the truth tables is given in the legend to Figure C.2.3.

This ALU control logic is simple because there are only three outputs, and only a few of the possible input combinations need to be recognized. If a large number of possible ALU function codes had to be transformed into ALU control signals, this simple method would not be efficient. Instead, you could use a decoder, a memory, or a structured array of logic gates. These techniques are described in Appendix B, and we will see examples when we examine the implementation of the multicycle controller in Section C.3.

Elaboration: In general, a logic equation and truth table representation of a logic function are equivalent. (We discuss this in further detail in Appendix B.) However, when a truth table only specifies the entries that result in nonzero outputs, it may not completely describe the logic function. A full truth table completely indicates all don't-care entries. For example, the encoding 11 for ALUOp always generates a don't care in the output. Thus a complete truth table would have XXX in the output portion for all entries

with 11 in the ALUOp field. These don't-care entries allow us to replace the ALUOp field 10 and 01 with 1X and X1, respectively. Incorporating the don't-care terms and minimizing the logic is both complex and error-prone and, thus, is better left to a program.

Mapping the Main Control Function to Gates

Implementing the main control function with an unstructured collection of gates, as we did for the ALU control, is reasonable because the control function is neither complex nor large, as we can see from the truth table shown in Figure C.2.4. However, if most of the 64 possible opcodes were used and there were many more control lines, the number of gates would be much larger and each gate could have many more inputs.

Since any function can be computed in two levels of logic, another way to implement a logic function is with a structured two-level logic array. Figure C.2.5 shows such an implementation. It uses an array of AND gates followed by an array of OR gates. This structure is called a *programmable logic array* (PLA). A PLA is one of the most common ways to implement a control function. We will return to the topic of using structured logic elements to implement control when we implement the finite state controller in the next section.

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.2.4 The control function for the simple one-clock implementation is completely specified by this truth table. This table is the same as that shown in Figure 5.22.

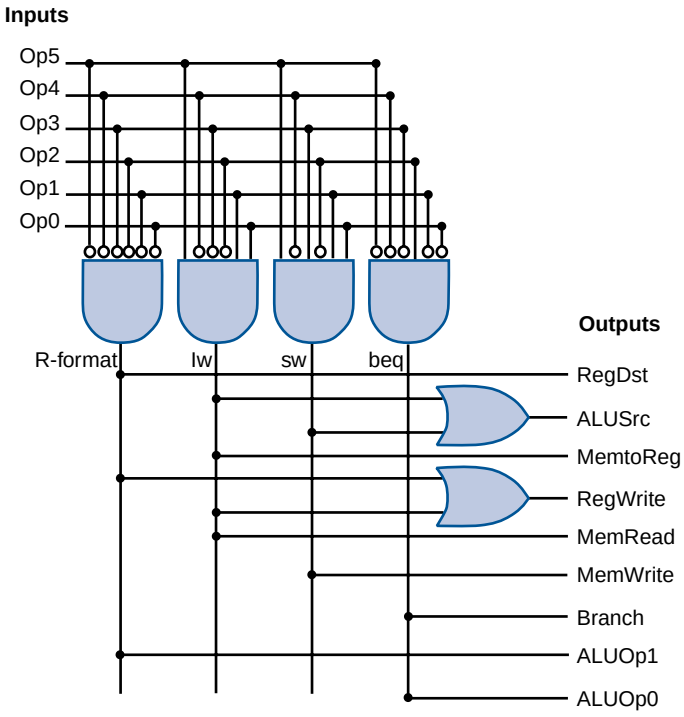

FIGURE C.2.5 The structured implementation of the control function as described by the truth table in Figure C.2.4. The structure, called a programmable logic array (PLA), uses an array of AND gates followed by an array of OR gates. The inputs to the AND gates are the function inputs and their inverses (bubbles indicate inversion of a signal). The inputs to the OR gates are the outputs of the AND gates (or, as a degenerate case, the function inputs and inverses). The output of the OR gates is the function outputs.

C.3 Implementing Finite State Machine Control

To implement the control as a finite state machine, we must first assign a number to each of the 10 states; any state could use any number, but we will use the sequential numbering for simplicity as we did in Chapter 5. (Figure C.3.1 is a copy of the finite state diagram from Figure 5.38 on page 339, reproduced for ease of access.) With 10 states we will need 4 bits to encode the state number, and we call these state bits: S3, S2, S1, S0. The current-state number will be stored in a state register, as shown in Figure C.3.2. If the states are assigned sequentially, state *i* is