

**chapter**

# 6

## COMBINATIONAL-CIRCUIT BUILDING BLOCKS

### CHAPTER OBJECTIVES

In this chapter you will learn about:

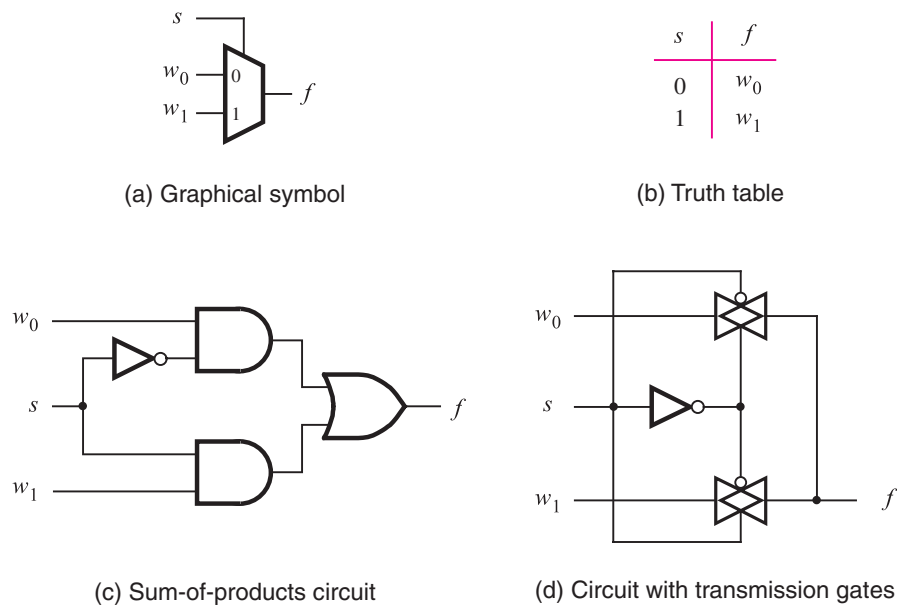
- Commonly used combinational subcircuits
- Multiplexers, which can be used for selection of signals and for implementation of general logic functions
- Circuits used for encoding, decoding, and code-conversion purposes
- Key Verilog constructs used to define combinational circuits

Previous chapters have introduced the basic techniques for design of logic circuits. In practice, a few types of logic circuits are often used as building blocks in larger designs. This chapter discusses a number of these blocks and gives examples of their use. The chapter also includes a major section on Verilog, which describes several key features of the language.

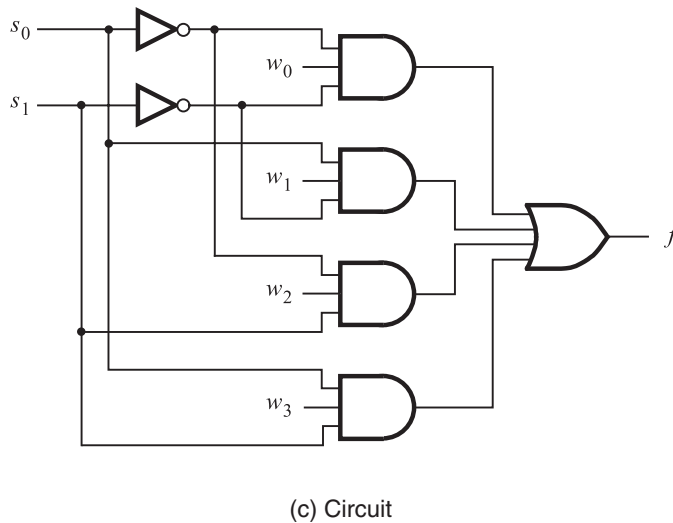
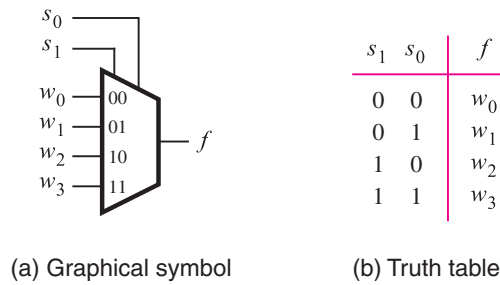
## 6.1 MULTIPLEXERS

Multiplexers were introduced briefly in Chapters 2 and 3. A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. It passes the signal value on one of the data inputs to the output. The data input is selected by the values of the select inputs. Figure 6.1 shows a 2-to-1 multiplexer. Part (a) gives the symbol commonly used. The *select* input,  $s$ , chooses as the output of the multiplexer either input  $w_0$  or  $w_1$ . The multiplexer's functionality can be described in the form of a truth table as shown in part (b) of the figure. Part (c) gives a sum-of-products implementation of the 2-to-1 multiplexer, and part (d) illustrates how it can be constructed with transmission gates.

Figure 6.2a depicts a larger multiplexer with four data inputs,  $w_0, \dots, w_3$ , and two select inputs,  $s_1$  and  $s_0$ . As shown in the truth table in part (b) of the figure, the two-bit number represented by  $s_1s_0$  selects one of the data inputs as the output of the multiplexer. A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 6.2c. It



**Figure 6.1** A 2-to-1 multiplexer.

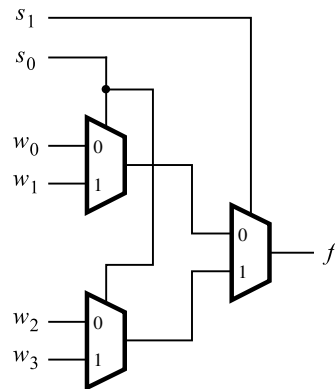


**Figure 6.2** A 4-to-1 multiplexer.

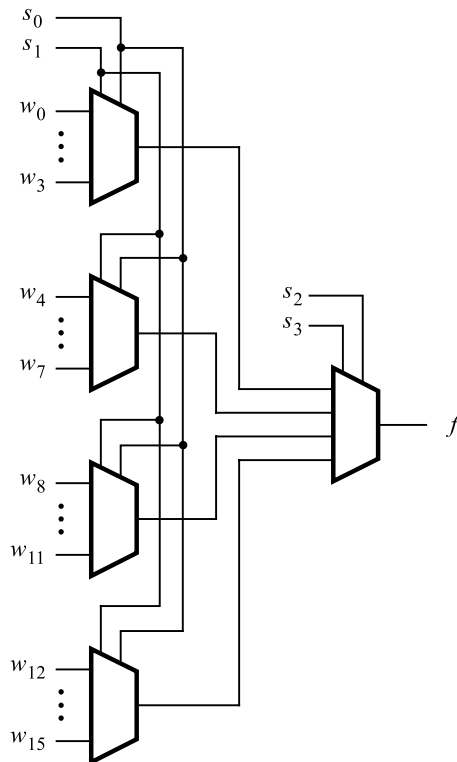
realizes the multiplexer function

$$f = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1 + s_1\bar{s}_0w_2 + s_1s_0w_3$$

It is possible to build larger multiplexers using the same approach. Usually, the number of data inputs,  $n$ , is an integer power of two. A multiplexer that has  $n$  data inputs,  $w_0, \dots, w_{n-1}$ , requires  $\lceil \log_2 n \rceil$  select inputs. Larger multiplexers can also be constructed from smaller multiplexers. For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers as illustrated in Figure 6.3. If the 4-to-1 multiplexer is implemented using transmission gates, then the structure in this figure is always used. Figure 6.4 shows how a 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.



**Figure 6.3** Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.



**Figure 6.4** A 16-to-1 multiplexer.

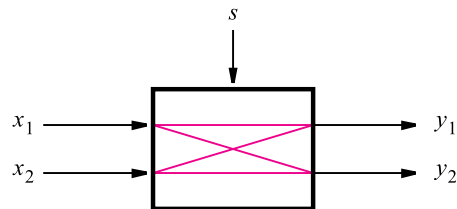
Figure 6.5 shows a circuit that has two inputs,  $x_1$  and  $x_2$ , and two outputs,  $y_1$  and  $y_2$ . As indicated by the blue lines, the function of the circuit is to allow either of its inputs to be connected to either of its outputs, under the control of another input,  $s$ . A circuit that has  $n$  inputs and  $k$  outputs, whose sole function is to provide a capability to connect any input to any output, is usually referred to as an  $n \times k$  crossbar switch. Crossbars of various sizes can be created, with different numbers of inputs and outputs. When there are two inputs and two outputs, it is called a  $2 \times 2$  crossbar.

### Example 6.1

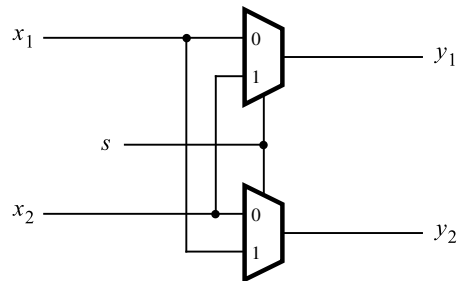
Figure 6.5b shows how the  $2 \times 2$  crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal  $s$ . If  $s = 0$ , the crossbar connects  $x_1$  to  $y_1$  and  $x_2$  to  $y_2$ , while if  $s = 1$ , the crossbar connects  $x_1$  to  $y_2$  and  $x_2$  to  $y_1$ . Crossbar switches are useful in many practical applications in which it is necessary to be able to connect one set of wires to another set of wires, where the connection pattern changes from time to time.

We introduced field-programmable gate array (FPGA) chips in section 3.6.5. Figure 3.39 depicts a small FPGA that is programmed to implement a particular circuit. The logic blocks in the FPGA have two inputs, and there are four tracks in each routing channel. Each of the programmable switches that connects a logic block input or output to an interconnection wire is shown as an X. A small part of Figure 3.39 is reproduced in Figure 6.6a. For clarity,

### Example 6.2

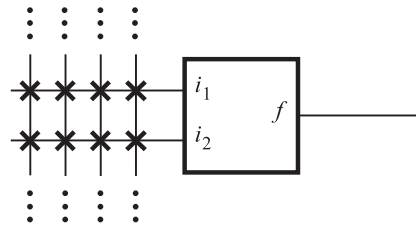


(a) A  $2 \times 2$  crossbar switch

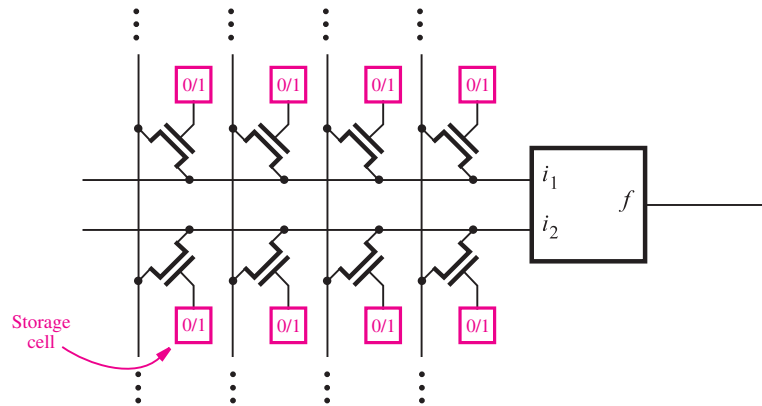


(b) Implementation using multiplexers

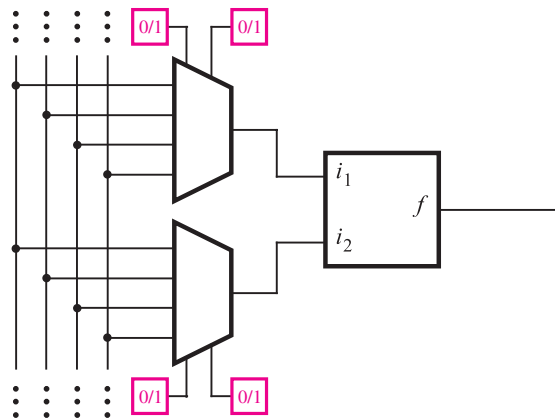
**Figure 6.5** A practical application of multiplexers.



(a) Part of the FPGA in Figure 3.39



(b) Implementation using pass transistors



(c) Implementation using multiplexers

**Figure 6.6** Implementing programmable switches in an FPGA.

the figure shows only a single logic block and the interconnection wires and switches associated with its input terminals.

One way in which the programmable switches can be implemented is illustrated in Figure 6.6*b*. Each  $X$  in part (a) of the figure is realized using an NMOS transistor controlled by a storage cell. This type of programmable switch was also shown in Figure 3.68. We described storage cells briefly in section 3.6.5 and will discuss them in more detail in section 10.1. Each cell stores a single logic value, either 0 or 1, and provides this value as the output of the cell. Each storage cell is built by using several transistors. Thus the eight cells shown in the figure use a significant amount of chip area.

The number of storage cells needed can be reduced by using multiplexers, as shown in Figure 6.6*c*. Each logic block input is fed by a 4-to-1 multiplexer, with the select inputs controlled by storage cells. This approach requires only four storage cells, instead of eight. In commercial FPGAs the multiplexer-based approach is usually adopted.

### 6.1.1 SYNTHESIS OF LOGIC FUNCTIONS USING MULTIPLEXERS

Multiplexers are useful in many practical applications, such as those described above. They can also be used in a more general way to synthesize logic functions. Consider the example in Figure 6.7*a*. The truth table defines the function  $f = w_1 \oplus w_2$ . This function can be implemented by a 4-to-1 multiplexer in which the values of  $f$  in each row of the truth table are connected as constants to the multiplexer data inputs. The multiplexer select inputs are driven by  $w_1$  and  $w_2$ . Thus for each valuation of  $w_1 w_2$ , the output  $f$  is equal to the function value in the corresponding row of the truth table.

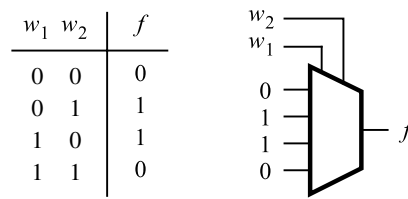
The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 6.7*b*, which allows  $f$  to be implemented by a single 2-to-1 multiplexer. One of the input signals,  $w_1$  in this example, is chosen as the select input of the 2-to-1 multiplexer. The truth table is redrawn to indicate the value of  $f$  for each value of  $w_1$ . When  $w_1 = 0$ ,  $f$  has the same value as input  $w_2$ , and when  $w_1 = 1$ ,  $f$  has the value of  $\bar{w}_2$ . The circuit that implements this truth table is given in Figure 6.7*c*. This procedure can be applied to synthesize a circuit that implements any logic function.

Figure 6.8*a* gives the truth table for the three-input majority function, and it shows how the truth table can be modified to implement the function using a 4-to-1 multiplexer. Any two of the three inputs may be chosen as the multiplexer select inputs. We have chosen  $w_1$  and  $w_2$  for this purpose, resulting in the circuit in Figure 6.8*b*.

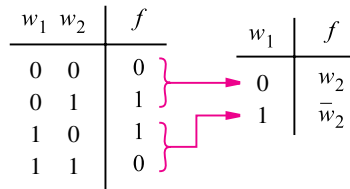
**Example 6.3**

Figure 6.9*a* indicates how the function  $f = w_1 \oplus w_2 \oplus w_3$  can be implemented using 2-to-1 multiplexers. When  $w_1 = 0$ ,  $f$  is equal to the XOR of  $w_2$  and  $w_3$ , and when  $w_1 = 1$ ,  $f$  is the

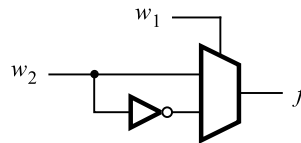
**Example 6.4**



(a) Implementation using a 4-to-1 multiplexer



(b) Modified truth table



(c) Circuit

**Figure 6.7** Synthesis of a logic function using multiplexers.

XNOR of  $w_2$  and  $w_3$ . The left multiplexer in the circuit produces  $w_2 \oplus w_3$ , using the result from Figure 6.7, and the right multiplexer uses the value of  $w_1$  to select either  $w_2 \oplus w_3$  or its complement. Note that we could have derived this circuit directly by writing the function as  $f = (w_2 \oplus w_3) \oplus w_1$ .

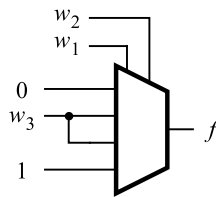
Figure 6.10 gives an implementation of the three-input XOR function using a 4-to-1 multiplexer. Choosing  $w_1$  and  $w_2$  for the select inputs results in the circuit shown.

## 6.1.2 MULTIPLEXER SYNTHESIS USING SHANNON'S EXPANSION

Figures 6.8 through 6.10 illustrate how truth tables can be interpreted to implement logic functions using multiplexers. In each case the inputs to the multiplexers are the constants 0 and 1, or some variable or its complement. Besides using such simple inputs, it is

$w_1$	$w_2$	$w_3$	$f$	$w_1$	$w_2$	$f$
0	0	0	0	0	0	0
0	0	1	0	0	1	$w_3$
0	1	0	0	1	0	$w_3$
0	1	1	1	1	1	1
1	0	0	0			
1	0	1	1			
1	1	0	1			
1	1	1	1			

(a) Modified truth table

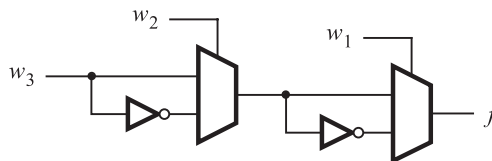


(b) Circuit

**Figure 6.8** Implementation of the three-input majority function using a 4-to-1 multiplexer.

$w_1$	$w_2$	$w_3$	$f$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a) Truth table



(b) Circuit

**Figure 6.9** Three-input XOR implemented with 2-to-1 multiplexers.

$w_1$	$w_2$	$w_3$	$f$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$\left. \begin{matrix} 0 \\ 1 \end{matrix} \right\} w_3$

$\left. \begin{matrix} 1 \\ 0 \end{matrix} \right\} \bar{w}_3$

$\left. \begin{matrix} 1 \\ 0 \end{matrix} \right\} \bar{w}_3$

$\left. \begin{matrix} 0 \\ 1 \end{matrix} \right\} w_3$

(a) Truth table
(b) Circuit

**Figure 6.10** Three-input XOR implemented with a 4-to-1 multiplexer.

possible to connect more complex circuits as inputs to a multiplexer, allowing functions to be synthesized using a combination of multiplexers and other logic gates. Suppose that we want to implement the three-input majority function in Figure 6.8 using a 2-to-1 multiplexer in this way. Figure 6.11 shows an intuitive way of realizing this function. The truth table can be modified as shown on the right. If  $w_1 = 0$ , then  $f = w_2w_3$ , and if  $w_1 = 1$ , then  $f = w_2 + w_3$ . Using  $w_1$  as the select input for a 2-to-1 multiplexer leads to the circuit in Figure 6.11b.

This implementation can be derived using algebraic manipulation as follows. The function in Figure 6.11a is expressed in sum-of-products form as

$$f = \bar{w}_1w_2w_3 + w_1\bar{w}_2w_3 + w_1w_2\bar{w}_3 + w_1w_2w_3$$

It can be manipulated into

$$\begin{aligned} f &= \bar{w}_1(w_2w_3) + w_1(\bar{w}_2w_3 + w_2\bar{w}_3 + w_2w_3) \\ &= \bar{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

which corresponds to the circuit in Figure 6.11b.

Multiplexer implementations of logic functions require that a given function be decomposed in terms of the variables that are used as the select inputs. This can be accomplished by means of a theorem proposed by Claude Shannon [1].

**Shannon's Expansion Theorem** Any Boolean function  $f(w_1, \dots, w_n)$  can be written in the form

$$f(w_1, w_2, \dots, w_n) = \bar{w}_1 \cdot f(0, w_2, \dots, w_n) + w_1 \cdot f(1, w_2, \dots, w_n)$$

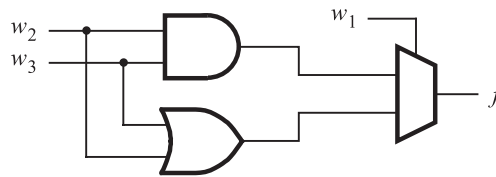
This expansion can be done in terms of any of the  $n$  variables. We will leave the proof of the theorem as an exercise for the reader (see problem 6.9).

$w_1$	$w_2$	$w_3$	$f$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$w_1$	$f$
0	$w_2 w_3$
1	$w_2 + w_3$

(a) Truth table



(b) Circuit

**Figure 6.11** The three-input majority function implemented using a 2-to-1 multiplexer.

To illustrate its use, we can apply the theorem to the three-input majority function, which can be written as

$$f(w_1, w_2, w_3) = w_1 w_2 + w_1 w_3 + w_2 w_3$$

Expanding this function in terms of  $w_1$  gives

$$f = \bar{w}_1(w_2 w_3) + w_1(w_2 + w_3)$$

which is the expression that we derived above.

For the three-input XOR function, we have

$$\begin{aligned} f &= w_1 \oplus w_2 \oplus w_3 \\ &= \bar{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot (\overline{w_2 \oplus w_3}) \end{aligned}$$

which gives the circuit in Figure 6.9b.

In Shannon's expansion the term  $f(0, w_2, \dots, w_n)$  is called the *cofactor* of  $f$  with respect to  $\bar{w}_1$ ; it is denoted in shorthand notation as  $f_{\bar{w}_1}$ . Similarly, the term  $f(1, w_2, \dots, w_n)$  is

called the cofactor of  $f$  with respect to  $w_1$ , written  $f_{w_1}$ . Hence we can write

$$f = \bar{w}_1 f_{\bar{w}_1} + w_1 f_{w_1}$$

In general, if the expansion is done with respect to variable  $w_i$ , then  $f_{w_i}$  denotes  $f(w_1, \dots, w_{i-1}, 1, w_{i+1}, \dots, w_n)$  and

$$f(w_1, \dots, w_n) = \bar{w}_i f_{\bar{w}_i} + w_i f_{w_i}$$

The complexity of the logic expression may vary, depending on which variable,  $w_i$ , is used, as illustrated in Example 6.5.

**Example 6.5** For the function  $f = \bar{w}_1 w_3 + w_2 \bar{w}_3$ , decomposition using  $w_1$  gives

$$\begin{aligned} f &= \bar{w}_1 f_{\bar{w}_1} + w_1 f_{w_1} \\ &= \bar{w}_1 (w_3 + w_2) + w_1 (w_2 \bar{w}_3) \end{aligned}$$

Using  $w_2$  instead of  $w_1$  produces

$$\begin{aligned} f &= \bar{w}_2 f_{\bar{w}_2} + w_2 f_{w_2} \\ &= \bar{w}_2 (\bar{w}_1 w_3) + w_2 (\bar{w}_1 + \bar{w}_3) \end{aligned}$$

Finally, using  $w_3$  gives

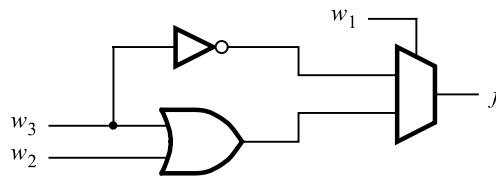
$$\begin{aligned} f &= \bar{w}_3 f_{\bar{w}_3} + w_3 f_{w_3} \\ &= \bar{w}_3 (w_2) + w_3 (\bar{w}_1) \end{aligned}$$

The results generated using  $w_1$  and  $w_2$  have the same cost, but the expression produced using  $w_3$  has a lower cost. In practice, the CAD tools that perform decompositions of this type try a number of alternatives and choose the one that produces the best result.

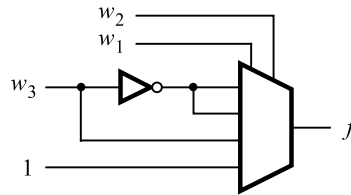
Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of  $w_1$  and  $w_2$  gives

$$\begin{aligned} f(w_1, \dots, w_n) &= \bar{w}_1 \bar{w}_2 \cdot f(0, 0, w_3, \dots, w_n) + \bar{w}_1 w_2 \cdot f(0, 1, w_3, \dots, w_n) \\ &\quad + w_1 \bar{w}_2 \cdot f(1, 0, w_3, \dots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \dots, w_n) \end{aligned}$$

This expansion gives a form that can be implemented using a 4-to-1 multiplexer. If Shannon's expansion is done in terms of all  $n$  variables, then the result is the canonical sum-of-products form, which was defined in section 2.6.1.



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

**Figure 6.12** The circuits synthesized in Example 6.6.

Assume that we wish to implement the function

**Example 6.6**

$$f = \bar{w}_1\bar{w}_3 + w_1w_2 + w_1w_3$$

using a 2-to-1 multiplexer and any other necessary gates. Shannon's expansion using  $w_1$  gives

$$\begin{aligned} f &= \bar{w}_1f_{\bar{w}_1} + w_1f_{w_1} \\ &= \bar{w}_1(\bar{w}_3) + w_1(w_2 + w_3) \end{aligned}$$

The corresponding circuit is shown in Figure 6.12a. Assume now that we wish to use a 4-to-1 multiplexer instead. Further decomposition using  $w_2$  gives

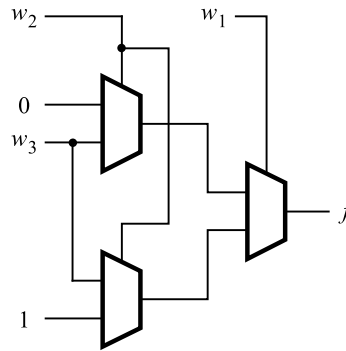
$$\begin{aligned} f &= \bar{w}_1\bar{w}_2f_{\bar{w}_1\bar{w}_2} + \bar{w}_1w_2f_{\bar{w}_1w_2} + w_1\bar{w}_2f_{w_1\bar{w}_2} + w_1w_2f_{w_1w_2} \\ &= \bar{w}_1\bar{w}_2(\bar{w}_3) + \bar{w}_1w_2(\bar{w}_3) + w_1\bar{w}_2(w_3) + w_1w_2(1) \end{aligned}$$

The circuit is shown in Figure 6.12b.

Consider the three-input majority function

**Example 6.7**

$$f = w_1w_2 + w_1w_3 + w_2w_3$$



**Figure 6.13** The circuit synthesized in Example 6.7.

We wish to implement this function using only 2-to-1 multiplexers. Shannon's expansion using  $w_1$  yields

$$\begin{aligned} f &= \bar{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) \\ &= \bar{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

Let  $g = w_2w_3$  and  $h = w_2 + w_3$ . Expansion of both  $g$  and  $h$  using  $w_2$  gives

$$\begin{aligned} g &= \bar{w}_2(0) + w_2(w_3) \\ h &= \bar{w}_2(w_3) + w_2(1) \end{aligned}$$

The corresponding circuit is shown in Figure 6.13. It is equivalent to the 4-to-1 multiplexer circuit derived using a truth table in Figure 6.8.

**Example 6.8** In section 3.6.5 we said that most FPGAs use lookup tables for their logic blocks. Assume that an FPGA exists in which each logic block is a three-input lookup table (3-LUT). Because it stores a truth table, a 3-LUT can realize any logic function of three variables. Using Shannon's expansion, any four-variable function can be realized with at most three 3-LUTs. Consider the function

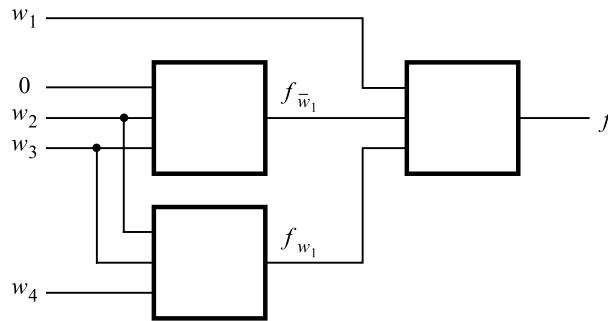
$$f = \bar{w}_2w_3 + \bar{w}_1w_2\bar{w}_3 + w_2\bar{w}_3w_4 + w_1\bar{w}_2\bar{w}_4$$

Expansion in terms of  $w_1$  produces

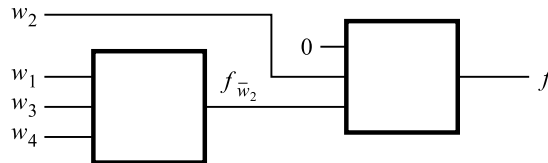
$$\begin{aligned} f &= \bar{w}_1f_{\bar{w}_1} + w_1f_{w_1} \\ &= \bar{w}_1(\bar{w}_2w_3 + w_2\bar{w}_3 + w_2\bar{w}_3w_4) + w_1(\bar{w}_2w_3 + w_2\bar{w}_3w_4 + \bar{w}_2\bar{w}_4) \\ &= \bar{w}_1(\bar{w}_2w_3 + w_2\bar{w}_3) + w_1(\bar{w}_2w_3 + w_2\bar{w}_3w_4 + \bar{w}_2\bar{w}_4) \end{aligned}$$

A circuit with three 3-LUTs that implements this expression is shown in Figure 6.14a. Decomposition of the function using  $w_2$ , instead of  $w_1$ , gives

$$\begin{aligned} f &= \bar{w}_2f_{\bar{w}_2} + w_2f_{w_2} \\ &= \bar{w}_2(w_3 + w_1\bar{w}_4) + w_2(\bar{w}_1\bar{w}_3 + \bar{w}_3w_4) \end{aligned}$$



(a) Using three 3-LUTs



(b) Using two 3-LUTs

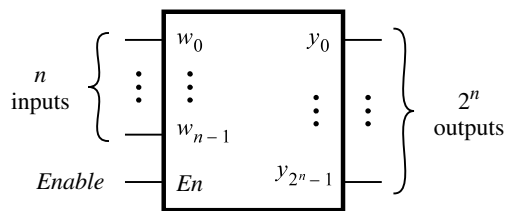
**Figure 6.14** Circuits synthesized in Example 6.8.

Observe that  $\bar{f}_{\bar{w}_2} = f_{w_2}$ ; hence only two 3-LUTs are needed, as illustrated in Figure 6.14b. The LUT on the right implements the two-variable function  $\bar{w}_2 f_{\bar{w}_2} + w_2 f_{w_2}$ .

Since it is possible to implement any logic function using multiplexers, general-purpose chips exist that contain multiplexers as their basic logic resources. Both Actel Corporation [2] and QuickLogic Corporation [3] offer FPGAs in which the logic block comprises an arrangement of multiplexers. Texas Instruments offers gate array chips that have multiplexer-based logic blocks [4].

## 6.2 DECODERS

Decoder circuits are used to decode encoded information. A binary decoder, depicted in Figure 6.15, is a logic circuit with  $n$  inputs and  $2^n$  outputs. Only one output is asserted at a time, and each output corresponds to one valuation of the inputs. The decoder also has an enable input,  $En$ , that is used to disable the outputs; if  $En = 0$ , then none of the decoder outputs is asserted. If  $En = 1$ , the valuation of  $w_{n-1} \cdots w_1 w_0$  determines which of the outputs is asserted. An  $n$ -bit binary code in which exactly one of the bits is set to 1 at a

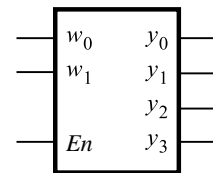


**Figure 6.15** An  $n$ -to- $2^n$  binary decoder.

time is referred to as *one-hot encoded*, meaning that the single bit that is set to 1 is deemed to be “hot.” The outputs of a binary decoder are one-hot encoded.

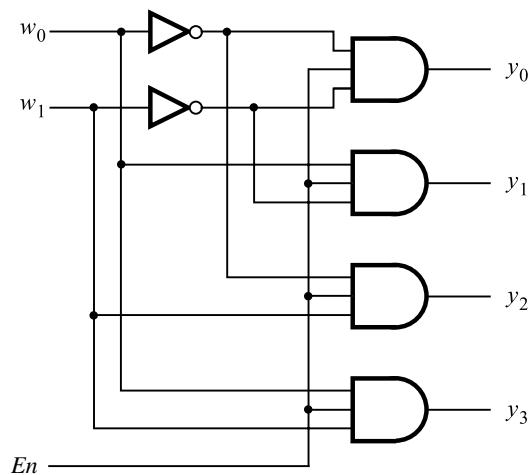
A 2-to-4 decoder is given in Figure 6.16. The two data inputs are  $w_1$  and  $w_0$ . They represent a two-bit number that causes the decoder to assert one of the outputs  $y_0, \dots, y_3$ . Although a decoder can be designed to have either active-high or active-low outputs, in

$En$	$w_1$	$w_0$	$y_0$	$y_1$	$y_2$	$y_3$
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0



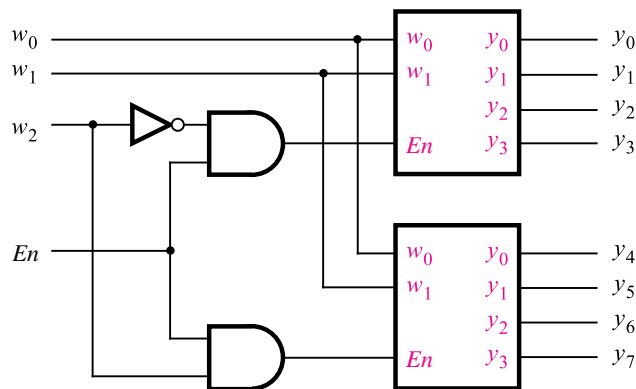
(a) Truth table

(b) Graphical symbol



(c) Logic circuit

**Figure 6.16** A 2-to-4 decoder.



**Figure 6.17** A 3-to-8 decoder using two 2-to-4 decoders.

Figure 6.16 active-high outputs are assumed. Setting the inputs  $w_1w_0$  to 00, 01, 10, or 11 causes the output  $y_0$ ,  $y_1$ ,  $y_2$ , or  $y_3$  to be set to 1, respectively. A graphical symbol for the decoder is given in part (b) of the figure, and a logic circuit is shown in part (c).

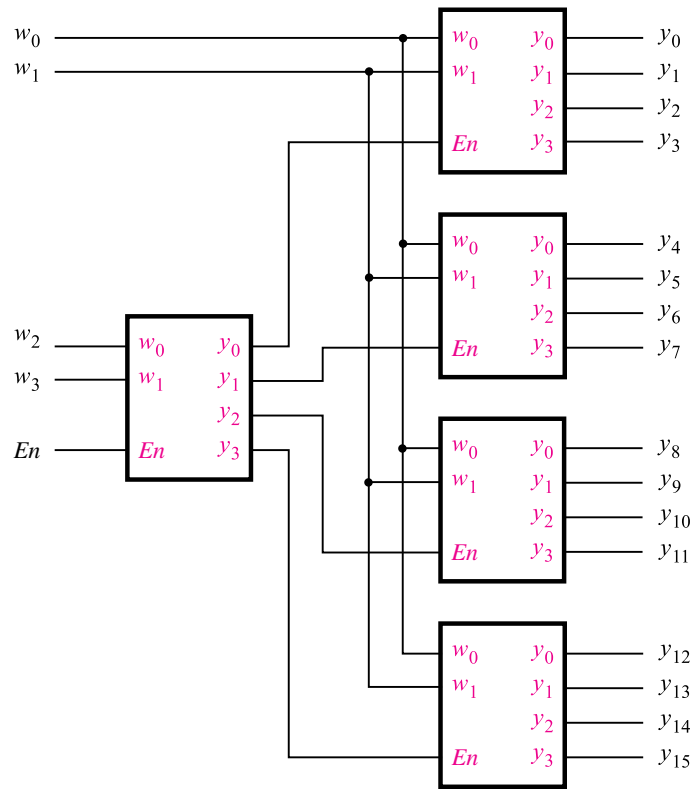
Larger decoders can be built using the sum-of-products structure in Figure 6.16c, or else they can be constructed from smaller decoders. Figure 6.17 shows how a 3-to-8 decoder is built with two 2-to-4 decoders. The  $w_2$  input drives the enable inputs of the two decoders. The top decoder is enabled if  $w_2 = 0$ , and the bottom decoder is enabled if  $w_2 = 1$ . This concept can be applied for decoders of any size. Figure 6.18 shows how five 2-to-4 decoders can be used to construct a 4-to-16 decoder. Because of its treelike structure, this type of circuit is often referred to as a *decoder tree*.

**D**ecoders are useful for many practical purposes. In Figure 6.2c we showed the sum-of-products implementation of the 4-to-1 multiplexer, which requires AND gates to distinguish the four different valuations of the select inputs  $s_1$  and  $s_0$ . Since a decoder evaluates the values on its inputs, it can be used to build a multiplexer as illustrated in Figure 6.19. The enable input of the decoder is not needed in this case, and it is set to 1. The four outputs of the decoder represent the four valuations of the select inputs.

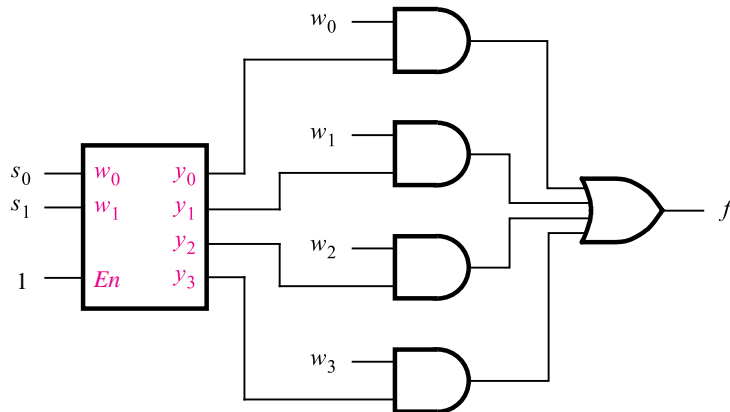
### Example 6.9

**I**n Figure 3.59 we showed how a 2-to-1 multiplexer can be constructed using two tri-state buffers. This concept can be applied to any size of multiplexer, with the addition of a decoder. An example is shown in Figure 6.20. The decoder enables one of the tri-state buffers for each valuation of the select lines, and that tri-state buffer drives the output,  $f$ , with the selected data input. We have now seen that multiplexers can be implemented in various ways. The choice of whether to employ the sum-of-products form, transmission gates, or tri-state buffers depends on the resources available in the chip being used. For instance, most FPGAs that use lookup tables for their logic blocks do not contain tri-state

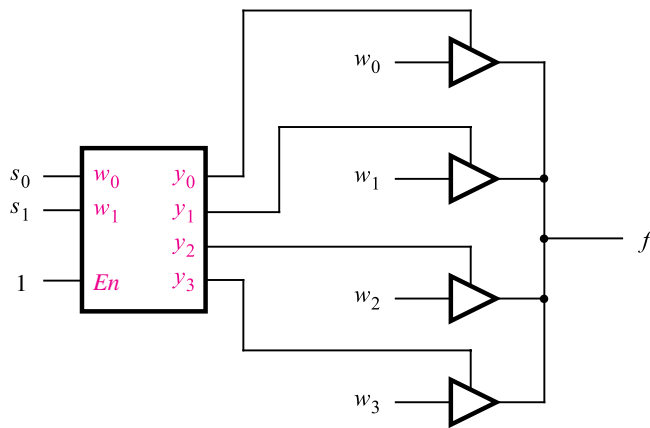
### Example 6.10



**Figure 6.18** A 4-to-16 decoder built using a decoder tree.



**Figure 6.19** A 4-to-1 multiplexer built using a decoder.



**Figure 6.20** A 4-to-1 multiplexer built using a decoder and tri-state buffers.

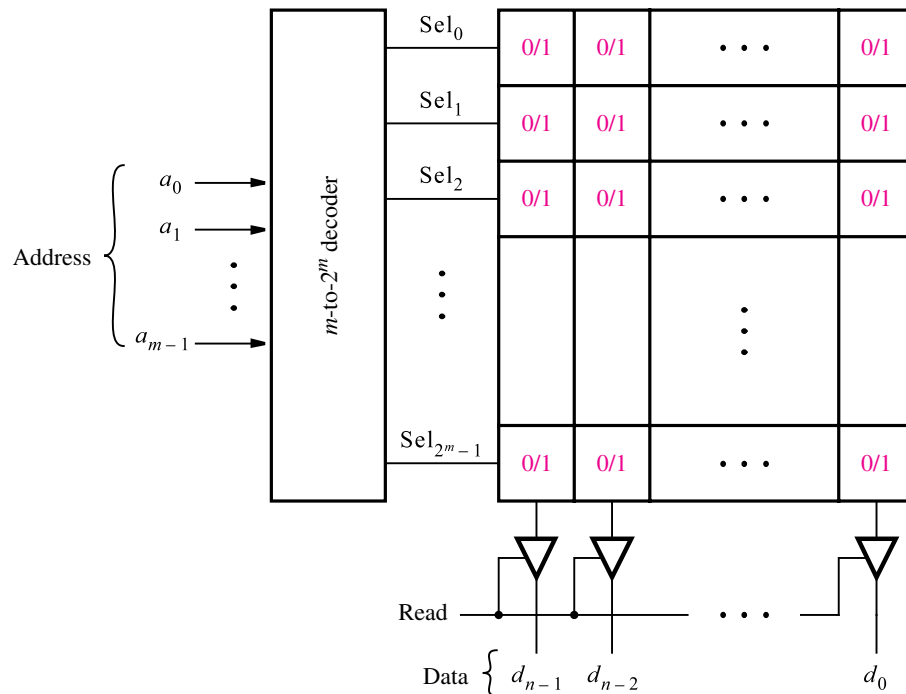
buffers. Hence multiplexers must be implemented in the sum-of-products form using the lookup tables (see Example 6.33).

### 6.2.1 DEMULTIPLEXERS

We showed in section 6.1 that a multiplexer has one output,  $n$  data inputs, and  $\lceil \log_2 n \rceil$  select inputs. The purpose of the multiplexer circuit is to *multiplex* the  $n$  data inputs onto the single data output under control of the select inputs. A circuit that performs the opposite function, namely, placing the value of a single data input onto multiple data outputs, is called a *demultiplexer*. The demultiplexer can be implemented using a decoder circuit. For example, the 2-to-4 decoder in Figure 6.16 can be used as a 1-to-4 demultiplexer. In this case the  $En$  input serves as the data input for the demultiplexer, and the  $y_0$  to  $y_3$  outputs are the data outputs. The valuation of  $w_1w_0$  determines which of the outputs is set to the value of  $En$ . To see how the circuit works, consider the truth table in Figure 6.16a. When  $En = 0$ , all the outputs are set to 0, including the one selected by the valuation of  $w_1w_0$ . When  $En = 1$ , the valuation of  $w_1w_0$  sets the appropriate output to 1.

In general, an  $n$ -to- $2^n$  decoder circuit can be used as a 1-to- $n$  demultiplexer. However, in practice decoder circuits are used much more often as decoders rather than as demultiplexers. In many applications the decoder's  $En$  input is not actually needed; hence it can be omitted. In this case the decoder always asserts one of its data outputs,  $y_0, \dots, y_{2^n-1}$ , according to the valuation of the data inputs,  $w_{n-1} \cdots w_0$ . Example 6.11 uses a decoder that does not have the  $En$  input.

**Example 6.11** One of the most important applications of decoders is in memory blocks, which are used to store information. Such memory blocks are included in digital systems, such as computers, where there is a need to store large amounts of information electronically. One type of memory block is called a *read-only memory* (ROM). A ROM consists of a collection of storage cells, where each cell permanently stores a single logic value, either 0 or 1. Figure 6.21 shows an example of a ROM block. The storage cells are arranged in  $2^m$  rows with  $n$  cells per row. Thus each row stores  $n$  bits of information. The location of each row in the ROM is identified by its *address*. In the figure the row at the top of the ROM has address 0, and the row at the bottom has address  $2^m - 1$ . The information stored in the rows can be accessed by asserting the select lines,  $\text{Sel}_0$  to  $\text{Sel}_{2^m-1}$ . As shown in the figure, a decoder with  $m$  inputs and  $2^m$  outputs is used to generate the signals on the select lines. Since the inputs to the decoder choose the particular address (row) selected, they are called the *address lines*. The information stored in the row appears on the data outputs of the ROM,  $d_{n-1}, \dots, d_0$ , which are called the *data lines*. Figure 6.21 shows that each data line has an associated tri-state buffer that is enabled by the ROM input named *Read*. To access, or *read*, data from the ROM, the address of the desired row is placed on the address lines and *Read* is set to 1.



**Figure 6.21** A  $2^m \times n$  read-only memory (ROM) block.

Many different types of memory blocks exist. In a ROM the stored information can be read out of the storage cells, but it cannot be changed (see problem 6.31). Another type of ROM allows information to be both read out of the storage cells and stored, or *written*, into them. Reading its contents is the normal operation, whereas writing requires a special procedure. Such a memory block is called a programmable ROM (PROM). The storage cells in a PROM are usually implemented using EEPROM transistors. We discussed EEPROM transistors in section 3.10 to show how they are used in PLDs. Other types of memory blocks are discussed in section 10.1.

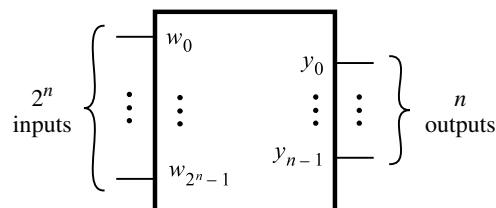
## 6.3 ENCODERS

An encoder performs the opposite function of a decoder. It encodes given information into a more compact form.

### 6.3.1 BINARY ENCODERS

A *binary encoder* encodes information from  $2^n$  inputs into an  $n$ -bit code, as indicated in Figure 6.22. Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1. The truth table for a 4-to-2 encoder is provided in Figure 6.23a. Observe that the output  $y_0$  is 1 when either input  $w_1$  or  $w_3$  is 1, and output  $y_1$  is 1 when input  $w_2$  or  $w_3$  is 1. Hence these outputs can be generated by the circuit in Figure 6.23b. Note that we assume that the inputs are one-hot encoded. All input patterns that have multiple inputs set to 1 are not shown in the truth table, and they are treated as don't-care conditions.

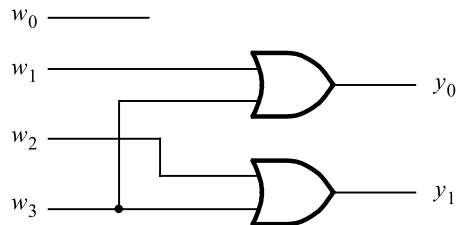
Encoders are used to reduce the number of bits needed to represent given information. A practical use of encoders is for transmitting information in a digital system. Encoding the information allows the transmission link to be built using fewer wires. Encoding is also useful if information is to be stored for later use because fewer bits need to be stored.



**Figure 6.22** A  $2^n$ -to- $n$  binary encoder.

$w_3$	$w_2$	$w_1$	$w_0$	$y_1$	$y_0$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

**Figure 6.23** A 4-to-2 binary encoder.

### 6.3.2 PRIORITY ENCODERS

Another useful class of encoders is based on the priority of input signals. In a *priority encoder* each input has a priority level associated with it. The encoder outputs indicate the active input that has the highest priority. When an input with a high priority is asserted, the other inputs with lower priority are ignored. The truth table for a 4-to-2 priority encoder is shown in Figure 6.24. It assumes that  $w_0$  has the lowest priority and  $w_3$  the highest. The outputs  $y_1$  and  $y_0$  represent the binary number that identifies the highest priority input set to 1. Since it is possible that none of the inputs is equal to 1, an output,  $z$ , is provided to indicate this condition. It is set to 1 when at least one of the inputs is equal to 1. It is set to

$w_3$	$w_2$	$w_1$	$w_0$	$y_1$	$y_0$	$z$
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

**Figure 6.24** Truth table for a 4-to-2 priority encoder.

0 when all inputs are equal to 0. The outputs  $y_1$  and  $y_0$  are not meaningful in this case, and hence the first row of the truth table can be treated as a don't-care condition for  $y_1$  and  $y_0$ .

The behavior of the priority encoder is most easily understood by first considering the last row in the truth table. It specifies that if input  $w_3$  is 1, then the outputs are set to  $y_1y_0 = 11$ . Because  $w_3$  has the highest priority level, the values of inputs  $w_2$ ,  $w_1$ , and  $w_0$  do not matter. To reflect the fact that their values are irrelevant,  $w_2$ ,  $w_1$ , and  $w_0$  are denoted by the symbol  $x$  in the truth table. The second-last row in the truth table stipulates that if  $w_2 = 1$ , then the outputs are set to  $y_1y_0 = 10$ , but only if  $w_3 = 0$ . Similarly, input  $w_1$  causes the outputs to be set to  $y_1y_0 = 01$  only if both  $w_3$  and  $w_2$  are 0. Input  $w_0$  produces the outputs  $y_1y_0 = 00$  only if  $w_0$  is the only input that is asserted.

A logic circuit that implements the truth table can be synthesized by using the techniques developed in Chapter 4. However, a more convenient way to derive the circuit is to define a set of intermediate signals,  $i_0, \dots, i_3$ , based on the observations above. Each signal,  $i_k$ , is equal to 1 only if the input with the same index,  $w_k$ , represents the highest-priority input that is set to 1. The logic expressions for  $i_0, \dots, i_3$  are

$$i_0 = \bar{w}_3\bar{w}_2\bar{w}_1w_0$$

$$i_1 = \bar{w}_3\bar{w}_2w_1$$

$$i_2 = \bar{w}_3w_2$$

$$i_3 = w_3$$

Using the intermediate signals, the rest of the circuit for the priority encoder has the same structure as the binary encoder in Figure 6.23, namely

$$y_0 = i_1 + i_3$$

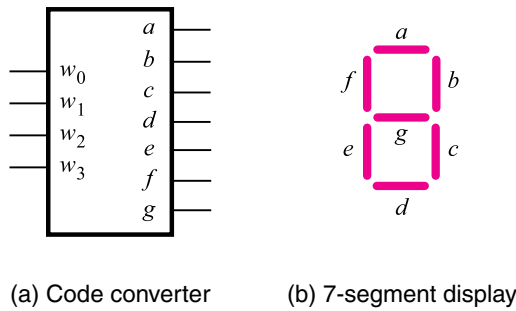
$$y_1 = i_2 + i_3$$

The output  $z$  is given by

$$z = i_0 + i_1 + i_2 + i_3$$

## 6.4 CODE CONVERTERS

The purpose of the decoder and encoder circuits is to convert from one type of input encoding to a different output encoding. For example, a 3-to-8 binary decoder converts from a binary number on the input to a one-hot encoding at the output. An 8-to-3 binary encoder performs the opposite conversion. There are many other possible types of code converters. One common example is a BCD-to-7-segment decoder, which converts one binary-coded decimal (BCD) digit into information suitable for driving a digit-oriented display. As illustrated in Figure 6.25a, the circuit converts the BCD digit into seven signals that are used to drive the segments in the display. Each segment is a small light-emitting diode (LED), which glows when driven by an electrical signal. The segments are labeled from  $a$  to  $g$  in the figure. The truth table for the BCD-to-7-segment decoder is given in Figure 6.25c. For each valuation of the inputs  $w_3, \dots, w_0$ , the seven outputs are set to



(a) Code converter (b) 7-segment display

$w_3$	$w_2$	$w_1$	$w_0$	$a$	$b$	$c$	$d$	$e$	$f$	$g$
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

(c) Truth table

**Figure 6.25** A BCD-to-7-segment display code converter.

display the appropriate BCD digit. Note that the last 6 rows of a complete 16-row truth table are not shown. They represent don't-care conditions because they are not legal BCD codes and will never occur in a circuit that deals with BCD data. A circuit that implements the truth table can be derived using the synthesis techniques discussed in Chapter 4. Finally, we should note that although the word *decoder* is traditionally used for this circuit, a more appropriate term is *code converter*. The term *decoder* is more appropriate for circuits that produce one-hot encoded outputs.

## 6.5 ARITHMETIC COMPARISON CIRCUITS

Chapter 5 presented arithmetic circuits that perform addition, subtraction, and multiplication of binary numbers. Another useful type of arithmetic circuit compares the relative sizes of two binary numbers. Such a circuit is called a *comparator*. This section considers the

design of a comparator that has two  $n$ -bit inputs,  $A$  and  $B$ , which represent unsigned binary numbers. The comparator produces three outputs, called  $AeqB$ ,  $AgtB$ , and  $AltB$ . The  $AeqB$  output is set to 1 if  $A$  and  $B$  are equal. The  $AgtB$  output is 1 if  $A$  is greater than  $B$ , and the  $AltB$  output is 1 if  $A$  is less than  $B$ .

The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of  $A$  and  $B$ . However, even for moderate values of  $n$ , the truth table is large. A better approach is to derive the comparator circuit by considering the bits of  $A$  and  $B$  in pairs. We can illustrate this by a small example, where  $n = 4$ .

Let  $A = a_3a_2a_1a_0$  and  $B = b_3b_2b_1b_0$ . Define a set of intermediate signals called  $i_3, i_2, i_1$ , and  $i_0$ . Each signal,  $i_k$ , is 1 if the bits of  $A$  and  $B$  with the same index are equal. That is,  $i_k = a_k \oplus b_k$ . The comparator's  $AeqB$  output is then given by

$$AeqB = i_3i_2i_1i_0$$

An expression for the  $AgtB$  output can be derived by considering the bits of  $A$  and  $B$  in the order from the most-significant bit to the least-significant bit. The first bit-position,  $k$ , at which  $a_k$  and  $b_k$  differ determines whether  $A$  is less than or greater than  $B$ . If  $a_k = 0$  and  $b_k = 1$ , then  $A < B$ . But if  $a_k = 1$  and  $b_k = 0$ , then  $A > B$ . The  $AgtB$  output is defined by

$$AgtB = a_3\bar{b}_3 + i_3a_2\bar{b}_2 + i_3i_2a_1\bar{b}_1 + i_3i_2i_1a_0\bar{b}_0$$

The  $i_k$  signals ensure that only the first digits, considered from the left to the right, of  $A$  and  $B$  that differ determine the value of  $AgtB$ .

The  $AltB$  output can be derived by using the other two outputs as

$$AltB = \overline{AeqB + AgtB}$$

A logic circuit that implements the four-bit comparator circuit is shown in Figure 6.26. This approach can be used to design a comparator for any value of  $n$ .

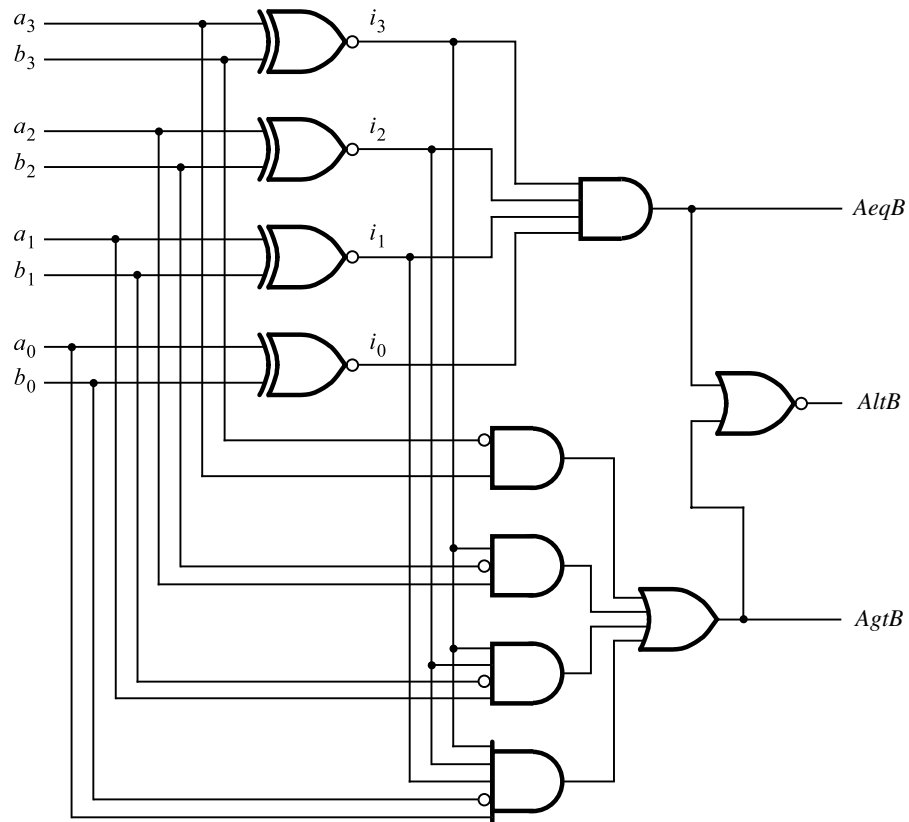
Comparator circuits, like most logic circuits, can be designed in different ways. Another approach for designing a comparator circuit is presented in Example 5.10 in Chapter 5.

## 6.6 VERILOG FOR COMBINATIONAL CIRCUITS

Having presented a number of useful building block circuits, we will now consider how such circuits can be described in Verilog. Rather than using gates or logic expressions, we will specify the circuits in terms of their behavior. We will also give a more rigorous description of previously used behavioral Verilog constructs and introduce some new ones.

### 6.6.1 THE CONDITIONAL OPERATOR

In a logic circuit it is often necessary to choose between several possible signals or values based on the state of some condition. A typical example is a multiplexer circuit in which the output is equal to the data input signal chosen by the valuation of the select inputs. For simple implementation of such choices Verilog provides a *conditional* operator ( $?:$ ) which



**Figure 6.26** A four-bit comparator circuit.

assigns one of two values depending on a conditional expression. It involves three operands used in the syntax

$$\text{conditional\_expression} ? \text{true\_expression} : \text{false\_expression}$$

If the conditional expression evaluates to 1 (true), then the value of `true_expression` is chosen; otherwise, the value of `false_expression` is chosen. For example, the statement

$$A = (B < C) ? (D + 5) : (D + 2);$$

means that if  $B$  is less than  $C$ , the value of  $A$  will be  $D + 5$  or else  $A$  will have the value  $D + 2$ . We used parentheses in the expression to improve readability; they are not necessary. The conditional operator can be used both in continuous assignment statements and in procedural statements inside an **always** block.

**A** 2-to-1 multiplexer can be defined using the conditional operator in an **assign** statement as shown in Figure 6.27. The module, named *mux2to1*, has the inputs  $w_0$ ,  $w_1$ , and  $s$ , and the output  $f$ . The signal  $s$  is used for the selection criterion. The output  $f$  is equal to  $w_1$  if the select input  $s$  has the value 1; otherwise,  $f$  is equal to  $w_0$ . Figure 6.28 shows how the same multiplexer can be defined by using the conditional operator inside an **always** block. **Example 6.12**

The same approach can be used to define a 4-to-1 multiplexer by nesting the conditional operators as indicated in Figure 6.29. The module is named *mux4to1*. Its two select inputs, which are called  $s_1$  and  $s_0$  in Figure 6.2, are represented by the two-bit vector  $S$ . The first conditional expression tests the value of bit  $s_1$ . If  $s_1 = 1$ , then  $s_0$  is tested and  $f$  is set to  $w_3$

```

module mux2to1 (w0, w1, s, f);
  input w0, w1, s;
  output f;

  assign f = s ? w1 : w0;

endmodule

```

**Figure 6.27** A 2-to-1 multiplexer specified using the conditional operator.

```

module mux2to1 (w0, w1, s, f);
  input w0, w1, s;
  output reg f;

  always @(w0, w1, s)
    f = s ? w1 : w0;

endmodule

```

**Figure 6.28** An alternative specification of a 2-to-1 multiplexer using the conditional operator.

```

module mux4to1 (w0, w1, w2, w3, S, f);
  input w0, w1, w2, w3;
  input [1:0] S;
  output f;

  assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule

```

**Figure 6.29** A 4-to-1 multiplexer specified using the conditional operator.

if  $s_0 = 1$  and  $f$  is set to  $w_2$  if  $s_0 = 0$ . This corresponds to the third and fourth rows of the truth table in Figure 6.2*b*. Similarly, if  $s_1 = 0$  the conditional operator on the right chooses  $f = w_1$  if  $s_0 = 1$  and  $f = w_0$  if  $s_0 = 0$ , thus realizing the first two rows of the truth table.

---

## 6.6.2 THE IF-ELSE STATEMENT

We have already used the **if-else** statement in previous chapters. It has the syntax

```
if (conditional_expression) statement;
else statement;
```

The conditional expression may use the operators given in Table A.1. If the expression is evaluated to true then the first statement (or a block of statements delineated by **begin** and **end** keywords) is executed or else the second statement (or a block of statements) is executed.

---

**Example 6.13** Figure 6.30 shows how the **if-else** statement can be used to describe a 2-to-1 multiplexer. The **if** clause states that  $f$  is assigned the value of  $w_0$  when  $s = 0$ . Else,  $f$  is assigned the value of  $w_1$ .

The **if-else** statement can be used to implement larger multiplexers. A 4-to-1 multiplexer is shown in Figure 6.31. The **if-else** clauses set  $f$  to the value of one of the inputs  $w_0, \dots, w_3$ , depending on the valuation of  $S$ . Compiling the code results in the circuit shown in Figure 6.2*c*.

Another way of defining the same circuit is presented in Figure 6.32. In this case, a four-bit vector  $W$  is defined instead of single-bit signals  $w_0, w_1, w_2$ , and  $w_3$ . Also, the four different values of  $S$  are specified as decimal rather than binary numbers.

---

**Example 6.14** Figure 6.4 shows how a 16-to-1 multiplexer is built using five 4-to-1 multiplexers. Figure 6.33 presents Verilog code for this circuit using five instantiations of the *mux4to1* module.

```
module mux2to1 (w0, w1, s, f);
  input w0, w1, s;
  output reg f;

  always @(w0, w1, s)
    if (s == 0)
      f = w0;
    else
      f = w1;

endmodule
```

**Figure 6.30** Code for a 2-to-1 multiplexer using the **if-else** statement.

```

module mux4to1 (w0, w1, w2, w3, S, f);
  input w0, w1, w2, w3;
  input [1:0] S;
  output reg f;

  always @(*)
    if (S == 2'b00)
      f = w0;
    else if (S == 2'b01)
      f = w1;
    else if (S == 2'b10)
      f = w2;
    else
      f = w3;

endmodule

```

**Figure 6.31** Code for a 4-to-1 multiplexer using the **if-else** statement.

```

module mux4to1 (W, S, f);
  input [0:3] W;
  input [1:0] S;
  output reg f;

  always @(W, S)
    if (S == 0)
      f = W[0];
    else if (S == 1)
      f = W[1];
    else if (S == 2)
      f = W[2];
    else
      f = W[3];

endmodule

```

**Figure 6.32** Alternative specification of a 4-to-1 multiplexer.

The data inputs to the *mux16to1* module are the 16-bit vector  $W$ , and the select inputs are the four-bit vector  $S16$ . In the Verilog code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left of Figure 6.4. A four-bit signal named  $M$  is used for this purpose. The first multiplexer instantiated, *Mux1*, corresponds to the multiplexer at the top left of Figure 6.4. Its first four ports, which correspond to  $w_0, \dots, w_3$  in Figure

```

module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output f;
  wire [0:3] M;

  mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
  mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
  mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
  mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
  mux4to1 Mux5 (M[0:3], S16[3:2], f);

endmodule

```

**Figure 6.33** Hierarchical code for a 16-to-1 multiplexer.

6.31, are driven by the signals  $W[0], \dots, W[3]$ . The syntax  $S16[1:0]$  is used to attach the signals  $S16[1]$  and  $S16[0]$  to the two-bit  $S$  port of the *mux4to1* module. The  $M[0]$  signal is connected to the multiplexer's output port. Similarly, *Mux2*, *Mux3*, and *Mux4* are instantiations of the next three multiplexers on the left. The multiplexer on the right of Figure 6.4 is instantiated as *Mux5*. The signals  $M[0], \dots, M[3]$  are connected to its data inputs, and bits  $S16[3]$  and  $S16[2]$ , which are specified by the syntax  $S16[3:2]$ , are attached to the select inputs. The output port generates the *mux16to1* output  $f$ . Compiling the code results in the multiplexer function

$$f = \bar{s}_3\bar{s}_2\bar{s}_1\bar{s}_0w_0 + \bar{s}_3\bar{s}_2\bar{s}_1s_0w_1 + \bar{s}_3\bar{s}_2s_1\bar{s}_0w_2 + \dots + s_3s_2s_1\bar{s}_0w_{14} + s_3s_2s_1s_0w_{15}$$

Since the *mux4to1* module is being instantiated in the code of Figure 6.33, it is necessary to either include the code of Figure 6.32 in the same file as the *mux16to1* module or place the *mux4to1* module in a separate file in the same directory, or a directory with a specified path so that the Verilog compiler can find it. Observe that if the code in Figure 6.31 were used as the required *mux4to1* module, then we would have to list the ports separately, as in  $W[0], W[1], W[2], W[3]$ , rather than as the vector  $W[0:3]$ .

### 6.6.3 THE CASE STATEMENT

The **if-else** statement provides the means for choosing an alternative based on the value of an expression. When there are many possible alternatives, the code based on this statement may become awkward to read. Instead, it is often possible to use the Verilog **case** statement which is defined as

```

case (expression)
  alternative1: statement;
  alternative2: statement;
  .
  .
  .
  alternativej: statement;
  [default: statement;]
endcase

```

The controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included. Otherwise, the Verilog compiler will synthesize memory elements to deal with the unspecified possibilities; we will discuss this issue in Chapter 7.

---

The **case** statement can be used to define a 4-to-1 multiplexer as shown in Figure 6.34. The four values that the select vector  $S$  can have are given as decimal numbers, but they could also be given as binary numbers. **Example 6.15**

```

module mux4to1 (W, S, f);
  input [0:3] W;
  input [1:0] S;
  output reg f;

  always @(W, S)
    case (S)
      0: f = W[0];
      1: f = W[1];
      2: f = W[2];
      3: f = W[3];
    endcase
endmodule

```

**Figure 6.34** A 4-to-1 multiplexer defined using the **case** statement.

**Example 6.16** Figure 6.35 shows how a **case** statement can be used to describe the truth table for a 2-to-4 binary decoder. The module is called *dec2to4*. The data inputs are the two-bit vector  $W$ , and the enable input is  $En$ . The four outputs are represented by the four-bit vector  $Y$ .

In the truth table for the decoder in Figure 6.16a, the inputs are listed in the order  $En w_1 w_0$ . To represent these three signals in the controlling expression, the Verilog code uses the concatenate operator to combine the  $En$  and  $W$  signals into a three-bit vector. The four alternatives in the **case** statement correspond to the truth table in Figure 6.16a where  $En = 1$ , and the decoder outputs have the same patterns as in the first four rows of the truth table. The last clause uses the **default** keyword and sets the decoder outputs to 0000, because it represents all other cases, namely those where  $En = 0$ .

**Example 6.17** The 2-to-4 decoder can be specified using a combination of **if-else** and **case** statements as given in Figure 6.36. The **case** alternatives are evaluated if  $En = 1$ ; otherwise, all four bits of the output  $Y$  are set to the value 0.

**Example 6.18** The tree structure of the 4-to-16 decoder in Figure 6.18 can be defined as shown in Figure 6.37. The inputs are a four-bit vector  $W$  and an enable signal  $En$ . The outputs are represented by the 16-bit vector  $Y$ . The circuit uses five instances of the 2-to-4 decoder defined in either Figure 6.35 or 6.36. The outputs of the leftmost decoder in Figure 6.18 are denoted as the four-bit vector  $M$  in Figure 6.37.

```

module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output reg [0:3] Y;

  always @(W, En)
    case ({En, W})
      3'b100: Y = 4'b1000;
      3'b101: Y = 4'b0100;
      3'b110: Y = 4'b0010;
      3'b111: Y = 4'b0001;
      default: Y = 4'b0000;
    endcase

endmodule

```

**Figure 6.35** Verilog code for a 2-to-4 binary decoder.

```

module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output reg [0:3] Y;

  always @(W, En)
  begin
    if (En == 0)
      Y = 4'b0000;
    else
      case (W)
        0: Y = 4'b1000;
        1: Y = 4'b0100;
        2: Y = 4'b0010;
        3: Y = 4'b0001;
      endcase
    end
  endmodule

```

**Figure 6.36** Alternative code for a 2-to-4 binary decoder.

```

module dec4to16 (W, Y, En);
  input [3:0] W;
  input En;
  output [0:15] Y;
  wire [0:3] M;

  dec2to4 Dec1 (W[3:2], M[0:3], En);
  dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
  dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
  dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
  dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule

```

**Figure 6.37** Verilog code for a 4-to-16 decoder.

---

Another example of a **case** statement is given in Figure 6.38. The module, *seg7*, represents the BCD-to-7-segment decoder in Figure 6.25. The BCD input is the four-bit vector named *bcd*, and the seven outputs are the seven-bit vector named *leds*. The **case** alternatives are listed so that they resemble the truth table in Figure 6.25c. Note that there is a comment to the right of the **case** statement, which labels the seven outputs with the letters from *a* **Example 6.19**

```

module seg7 (bcd, leds);
  input [3:0] bcd;
  output reg [1:7] leds;

  always @(bcd)
    case (bcd) //abcdefg
      0: leds = 7'b1111110;
      1: leds = 7'b0110000;
      2: leds = 7'b1101101;
      3: leds = 7'b1111001;
      4: leds = 7'b0110011;
      5: leds = 7'b1011011;
      6: leds = 7'b1011111;
      7: leds = 7'b1110000;
      8: leds = 7'b1111111;
      9: leds = 7'b1111011;
      default: leds = 7'bx;
    endcase
endmodule

```

**Figure 6.38** Code for a BCD-to-7-segment decoder.

to *g*. These labels indicate to the reader the correlation between the bits of the *leds* vector in the Verilog code and the seven segments in Figure 6.25*b*. The final **case** alternative sets all seven bits of *leds* to *x*. Recall that *x* is used in Verilog to denote a don't-care condition. This alternative represents the don't-care conditions discussed for Figure 6.25, which are the cases where the *bcd* input does not represent a valid BCD digit.

---

**Example 6.20** An arithmetic logic unit (ALU) is a logic circuit that performs various Boolean and arithmetic operations on *n*-bit operands. In section 3.5 we discussed a family of standard chips called the 7400-series chips. We said that some of these chips contain basic logic gates, and others provide commonly used logic circuits. One example of an ALU is the chip called the 74381. Table 6.1 specifies the functionality of this chip. It has 2 four-bit data inputs, *A* and *B*, a three-bit select input, *S*, and a four-bit output, *F*. As the table shows, *F* is defined by various arithmetic or Boolean operations on the inputs *A* and *B*. In this table + means arithmetic addition, and − means arithmetic subtraction. To avoid confusion, the table uses the words XOR, OR, and AND for the Boolean operations. Each Boolean operation is done in a bitwise fashion. For example,  $F = A \text{ AND } B$  produces the four-bit result  $f_0 = a_0b_0$ ,  $f_1 = a_1b_1$ ,  $f_2 = a_2b_2$ , and  $f_3 = a_3b_3$ .

Figure 6.39 shows how the functionality of the 74381 ALU can be described in Verilog code. The case statement shown corresponds directly to Table 6.1. To check the functionality of the code, we synthesized a circuit for implementation in a PLD, and show a timing simulation in Figure 6.40. For each valuation of *s*, the circuit generates the appropriate Boolean or arithmetic operation.

---

**Table 6.1** The functionality of the 74381 ALU.

Operation	Inputs	Outputs
	$s_2 s_1 s_0$	F
Clear	0 0 0	0 0 0 0
B-A	0 0 1	$B - A$
A-B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	$A \text{ XOR } B$
OR	1 0 1	$A \text{ OR } B$
AND	1 1 0	$A \text{ AND } B$
Preset	1 1 1	1 1 1 1

```
// 74381 ALU
module alu (s, A, B, F);
  input [2:0] s;
  input [3:0] A, B;
  output reg [3:0] F;

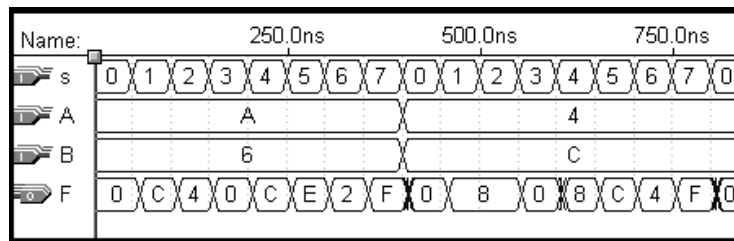
  always @(s, A, B)
    case (s)
      0: F = 4'b0000;
      1: F = B - A;
      2: F = A - B;
      3: F = A + B;
      4: F = A ^ B;
      5: F = A | B;
      6: F = A & B;
      7: F = 4'b1111;
    endcase
endmodule
```

**Figure 6.39** Code that represents the functionality of the 74381 ALU chip.

### The Casex and Casez Statements

In the **case** statement it is possible to use the logic values 0, 1, z, and x in the **case** alternatives. A bit-by-bit comparison is used to determine the match between the expression and one of the alternatives.

Verilog provides two variants of the **case** statement that treat the z and x values in a different way. The **casez** statement treats all z values in the case alternatives and the



**Figure 6.40** Timing simulation for the code in Figure 6.39.

controlling expression as don't cares. The **casex** statement treats all *z* and *x* values as don't cares.

**Example 6.21** Figure 6.41 gives Verilog code for the priority encoder defined in Figure 6.24. The desired priority scheme is realized by using a **casex** statement. The first alternative specifies that, if the input  $w_3$  is 1, then the output is set to  $y_1y_0 = 3$ . This assignment does not depend on the values of inputs  $w_2$ ,  $w_1$ , or  $w_0$ ; hence their values do not matter. The other alternatives in the **casex** statement are evaluated only if  $w_3 = 0$ . The second alternative states that if  $w_2$  is 1, then  $y_1y_0 = 2$ . If  $w_2 = 0$ , then the next alternative results in  $y_1y_0 = 1$  if  $w_1 = 1$ . If  $w_3 = w_2 = w_1 = 0$  and  $w_0 = 1$ , then the fourth alternative results in  $y_1y_0 = 0$ .

```

module priority (W, Y, z);
  input [3:0] W;
  output reg [1:0] Y;
  output reg z;

  always @(W)
  begin
    z = 1;
    casex (W)
      4'b1xxx: Y = 3;
      4'b01xx: Y = 2;
      4'b001x: Y = 1;
      4'b0001: Y = 0;
      default: begin
        z = 0;
        Y = 2'bx;
      end
    endcase
  end

endmodule

```

**Figure 6.41** Verilog code for a priority encoder.

The priority encoder's output  $z$  must be set to 1 whenever at least one of the data inputs is 1. This output is set to 1 at the start of the **always** block. If none of the four alternatives matches the value of  $W$ , then the **default** clause is executed. It consists of a two-statement block that resets  $z$  to 0 and indicates that the  $Y$  output can be set to any pattern because it will be ignored.

### 6.6.4 THE FOR LOOP

If the structure of a desired circuit exhibits a certain regularity, it may be possible to define the circuit using a **for** loop. We introduced the **for** loop in section 5.5.4, where it was useful in a generic specification of a ripple-carry adder. The **for** loop has the syntax

```
for (initial_index; terminal_index; increment) statement;
```

A loop control variable, which has to be of type **integer**, is set to the value given as the initial index. It is used in the statement or a block of statements delineated by **begin** and **end** keywords. After each iteration, the control variable is changed as defined in the increment. The iterations end after the control variable has reached the terminal index.

Unlike **for** loops in high-level programming languages, the Verilog **for** loop does not specify changes that take place in time through successive loop iterations. Instead, during each iteration it specifies a different subcircuit. In Figure 5.28 the **for** loop was used to define a cascade of full-adder subcircuits to form an  $n$ -bit ripple-carry adder. The **for** loop can be used to define many other structures as illustrated by the next two examples.

Figure 6.42 shows how the **for** loop can be used to specify a 2-to-4 decoder circuit. The effect of the loop is to repeat the **if-else** statement four times, for  $k = 0, \dots, 3$ . The first **Example 6.22**

```
module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output reg [0:3] Y;
  integer k;

  always @(W, En)
    for (k = 0; k <= 3; k = k+1)
      if ((W == k) && (En == 1))
        Y[k] = 1;
      else
        Y[k] = 0;

endmodule
```

**Figure 6.42** A 2-to-4 binary decoder specified using the **for** loop.

loop iteration sets  $y_0 = 1$  if  $W = 0$  and  $En = 1$ . Similarly, the other three iterations set the values of  $y_1, y_2,$  and  $y_3$  according to the values of  $W$  and  $En$ .

This arrangement can be used to specify a large  $n$ -to- $2^n$  decoder simply by increasing the sizes of vectors  $W$  and  $Y$  accordingly, and making  $n - 1$  be the terminal index value of  $k$ .

---

**Example 6.23** The priority encoder of Figure 6.24 can be defined by the Verilog code in Figure 6.43. In the **always** block, the output bits  $y_1$  and  $y_0$  are first set to the don't-care state and  $z$  is cleared to 0. Then, if one or more of the four inputs  $w_3, \dots, w_0$  is equal to 1, the **for** loop will set the valuation of  $y_1y_0$  to match the index of the highest priority input that has the value 1. Note that each successive iteration through the loop corresponds to a higher priority. Verilog semantics specify that a signal that receives multiple assignments in an **always** block retains the last assignment. Thus the iteration that corresponds to the highest priority input that is equal to 1 will override any setting of  $Y$  established during the previous iterations.

---

### 6.6.5 VERILOG OPERATORS

In this section we discuss the Verilog operators that are useful for synthesizing logic circuits. Table 6.2 lists these operators in groups that reflect the type of operation performed. A more complete listing of the operators is given in Table A.1.

```

module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;
    integer k;

    always @(W)
    begin
        Y = 2'bx;
        z = 0;
        for (k = 0; k < 4; k = k+1)
            if (W[k])
                begin
                    Y = k;
                    z = 1;
                end
            end
    end

endmodule

```

**Figure 6.43** A priority encoder specified using the **for** loop.

**Table 6.2** Verilog operators.

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
		Bitwise OR	2
	^	Bitwise XOR	2
	~ ^ or ^ ~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
		OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
		Reduction OR	1
	~	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^ ~	Reduction XNOR	1
Arithmetic	+	Addition	2
	-	Subtraction	2
	~	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal to	2
	<=	Less than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{,}	Concatenation	Any number
Replication	{ {} }	Replication	Any number
Conditional	?:	Conditional	3

To illustrate the results produced by the various operators, we will use three-bit vectors  $A[2:0]$ ,  $B[2:0]$  and  $C[2:0]$ , as well as scalars  $f$  and  $w$ .

### Bitwise Operators

Bitwise operators operate on individual bits of operands. The  $\sim$  operator forms the 1's complement of the operand such that the statement

$$C = \sim A;$$

produces the result  $c_2 = \bar{a}_2$ ,  $c_1 = \bar{a}_1$ , and  $c_0 = \bar{a}_0$ , where  $a_i$  and  $c_i$  are the bits of the vectors  $A$  and  $C$ .

Other bitwise operators operate on pairs of bits. The statement

$$C = A \& B;$$

generates  $c_2 = a_2 \cdot b_2$ ,  $c_1 = a_1 \cdot b_1$ , and  $c_0 = a_0 \cdot b_0$ . Similarly, the  $|$  and  $\wedge$  operators perform bitwise OR and XOR operations. The  $\wedge\sim$  operator, which can also be written as  $\sim\wedge$ , produces the XNOR such that

$$C = A \sim\wedge B;$$

gives  $c_2 = \overline{a_2 \oplus b_2}$ ,  $c_1 = \overline{a_1 \oplus b_1}$ , and  $c_0 = \overline{a_0 \oplus b_0}$ . If the operands are of unequal size, then the shorter operand is extended by padding 0s to the left.

A scalar function may be assigned a value as a result of a bitwise operation on two vector operands. In this case, it is only the least-significant bits of the operands that are involved in the operation. Hence the statement

$$f = A \wedge B;$$

yields  $f = a_0 \oplus b_0$ .

The bitwise operations may involve operands that include the unknown logic value  $x$ . Then the operations are performed according to the truth tables in Figure 6.44. For example, if  $P = 4'b101x$  and  $Q = 4'b1001$ , then  $P \& Q = 4'b100x$  while  $P | Q = 4'b1011$ .

### Logical Operators

The  $!$  operator has the same effect on a scalar operand as the  $\sim$  operator. Thus,  $f = !w = \sim w$ . But the effect on a vector operand is different, namely if

$$f = !A;$$

then  $f = \overline{a_2 + a_1 + a_0}$ .

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

**Figure 6.44** Truth tables for bitwise operators.

The `&&` operator implements the AND operation such that

$$f = A \&\& B;$$

produces  $f = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0)$ . Similarly, using the `||` operator in

$$f = A || B;$$

gives  $f = (a_2 + a_1 + a_0) + (b_2 + b_1 + b_0)$ .

### Reduction Operators

The reduction operators perform an operation on the bits of a single vector operand and produce a one-bit result. Using the `&` operator in

$$f = \&A;$$

produces  $f = a_2 \cdot a_1 \cdot a_0$ . Similarly,

$$f = \^A;$$

gives  $f = a_2 \oplus a_1 \oplus a_0$ , and so on. As an example of reduction operator use, consider the parity function discussed in section 5.8. The XOR circuit that computes the parity bit,  $p$ , of an  $n$ -bit vector  $X$  can be defined with the statement

$$p = \^X;$$

### Arithmetic Operators

We have already encountered the arithmetic operators in Chapter 5. They perform standard arithmetic operations. Thus

$$C = A + B;$$

puts the three-bit sum of  $A$  plus  $B$  into  $C$ , while

$$C = A - B;$$

puts the difference of  $A$  and  $B$  into  $C$ . The operation

$$C = -A;$$

places the 2's complement of  $A$  into  $C$ .

The addition, subtraction, and multiplication operations are supported by most CAD synthesis tools. However, the division operation is often not supported. When the Verilog compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

### Relational Operators

The relational operators are typically used as conditions in **if-else** and **for** statements. These operators function in the same way as the corresponding operators in the C programming language. An expression that uses the relational operators returns the value 1 if it is evaluated as true, and the value 0 if evaluated as false. If there are any  $x$  (unknown) or  $z$  bits in the operands, then the expression takes the value  $x$ .

```

module compare (A, B, AeqB, AgtB, AltB);
  input [3:0] A, B;
  output reg AeqB, AgtB, AltB;

  always @(A, B)
  begin
    AeqB = 0;
    AgtB = 0;
    AltB = 0;
    if (A == B)
      AeqB = 1;
    else if (A > B)
      AgtB = 1;
    else
      AltB = 1;
  end

endmodule

```

**Figure 6.45** Verilog code for a four-bit comparator.

---

**Example 6.24** The use of relational operators in the **if-else** statement is illustrated in Figure 6.45. The defined circuit is the four-bit comparator described in section 6.5.

---

### Equality Operators

The expression  $(A == B)$  is evaluated as true if  $A$  is equal to  $B$  and false otherwise. The  $!=$  operator has the opposite effect. The result is ambiguous ( $x$ ) if either operand contains  $x$  or  $z$  values.

### Shift Operators

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted, the vacant bit positions are filled with 0s. For example,

$$B = A \ll 1;$$

results in  $b_2 = a_1$ ,  $b_1 = a_0$ , and  $b_0 = 0$ . Similarly,

$$B = A \gg 2;$$

yields  $b_2 = b_1 = 0$  and  $b_0 = a_2$ .

### Concatenate Operator

This operator concatenates two or more vectors to create a larger vector. For example,

$$D = \{A, B\};$$

defines the six-bit vector  $D = a_2a_1a_0b_2b_1b_0$ . Similarly, the concatenation

$$E = \{3'b111, A, 2'b00\};$$

produces the eight-bit vector  $E = 111a_2a_1a_000$ .

### Replication Operator

This operator allows repetitive concatenation of the same vector, which is replicated the number of times indicated in the replication constant. For example,  $\{3\{A\}\}$  is equivalent to writing  $\{A, A, A\}$ . The specification  $\{4\{2'b10\}\}$  produces the eight-bit vector 10101010.

The replication operator may be used in conjunction with the concatenate operator. For instance,  $\{2\{A\}, 3\{B\}\}$  is equivalent to  $\{A, A, B, B, B\}$ . We introduced the concatenate and replication operators in section 5.5.6 and illustrated their use in specifying the adder circuits.

### Conditional Operator

The conditional operator is discussed fully in section 6.6.1.

### Operator Precedence

The Verilog operators are assumed to have the precedence indicated in Table 6.3. The order of precedence is from top to bottom; operators in the top row have the highest precedence and those in the bottom row have the lowest precedence. The operators listed in the same row have the same precedence.

**Table 6.3** Precedence of Verilog operators.

Operator type	Operator symbols	Precedence
Complement	! ~ -	Highest precedence
Arithmetic	* / + -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~& & ~^   ~	
Logical	&& 	
Conditional	?:	Lowest precedence

The designer can use parentheses to change the precedence of operators in Verilog code or remove any possible misinterpretation. It is a good practice to use parentheses to make the code unambiguous and easy to read.

### 6.6.6 THE GENERATE CONSTRUCT

In section 5.5.4 we introduced the **generate** loop capability which can be used to create multiple instances of subcircuits. A subcircuit may be defined in a block of statements delineated by the **generate** and **endgenerate** keywords. The subcircuit is instantiated multiple times using a generate-index variable. This variable is defined using the **genvar** keyword and it can have only positive integer values. It is not possible to use an index declared as a normal **integer** variable.

---

**Example 6.25** Figure 6.46 shows how the **generate** construct can be used to specify an  $n$ -bit ripple-carry adder. The subcircuit is a full-adder defined structurally in terms of primitive gates as introduced in Figure 5.22. The **for** loop causes the full-adder block to be instantiated  $n$  times.

In this example, the **for** statement is used in the **generate** block to control the selection of the generated objects. The **generate** block can also contain **if-else** and **case** statements to determine which objects are generated.

---

```

module addern (carryin, X, Y, S, carryout);
  parameter n = 32;
  input carryin;
  input [n-1:0] X, Y;
  output [n-1:0] S;
  output carryout;
  wire [n:0] C;

  genvar k;
  assign C[0] = carryin;
  assign carryout = C[n];
  generate
    for (k = 0; k < n; k = k+1)
      begin: fulladd_stage
        wire z1, z2, z3; //wires within full-adder
        xor (S[k], X[k], Y[k], C[k]);
        and (z1, X[k], Y[k]);
        and (z2, X[k], C[k]);
        and (z3, Y[k], C[k]);
        or (C[k+1], z1, z2, z3);
      end
    endgenerate

endmodule

```

**Figure 6.46** Using the **generate** loop to define an  $n$ -bit ripple-carry adder.

### 6.6.7 TASKS AND FUNCTIONS

In high-level programming languages it is possible to use subroutines and functions to avoid replicating specific routines that may be needed in several places of a given program. Verilog provides similar capabilities, known as tasks and functions. They can be used to modularize large designs and make the Verilog code easier to understand.

#### Verilog Task

A task is declared by the keyword **task** and it comprises a block of statements that ends with the keyword **endtask**. The task must be included in the module that calls it. It may have input and output ports. These are not the ports of the module that contains the task, which are used to make external connections to the module. The task ports are used only to pass values between the module and the task.

---

In Figure 6.33 we showed the Verilog code for a 16-to-1 multiplexer that instantiates five copies of a 4-to-1 multiplexer circuit given in a separate module named *mux4to1*. The same circuit can be specified using the task approach as shown in Figure 6.47. Observe the key differences. The task *mux4to1* is included in the module *mux16to1*. It is called from an **always** block by means of an appropriate **case** statement. The output of a task must be a variable, hence *g* is of **reg** type. **Example 6.26**

---

#### Verilog Function

A function is declared by the keyword **function** and it comprises a block of statements that ends with the keyword **endfunction**. The function must have at least one input and it returns a single value that is placed where the function is invoked.

---

Figure 6.48 shows how the code in Figure 6.47 can be written to use a function. The Verilog compiler essentially inserts the body of the function at each place where it is called. Hence the clause **Example 6.27**

```
0: f = mux4to1 (W[0:3], S16[1:0]);
```

becomes

```
0: case (S16[1:0])
    0: f = W[0];
    1: f = W[1];
    2: f = W[2];
    3: f = W[3];
endcase
```

```

module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output reg f;

  always @(W, S16)
    case (S16[3:2])
      0: mux4to1 (W[0:3], S16[1:0], f);
      1: mux4to1 (W[4:7], S16[1:0], f);
      2: mux4to1 (W[8:11], S16[1:0], f);
      3: mux4to1 (W[12:15], S16[1:0], f);
    endcase

  // Task that specifies a 4-to-1 multiplexer
  task mux4to1;
    input [0:3] X;
    input [1:0] S4;
    output reg g;

    case (S4)
      0: g = X[0];
      1: g = X[1];
      2: g = X[2];
      3: g = X[3];
    endcase
  endtask

endmodule

```

**Figure 6.47** Use of a task in Verilog code.

The function serves as a convenience that makes the mux16to1 module compact and easier to read.

---

A Verilog function can invoke another function but it cannot call a Verilog task. A task may call another task and it may invoke a function. In Figure 6.47 we defined the task after the **always** block that calls it. In contrast, in Figure 6.48 we defined the function before the **always** block that invokes it. Both possibilities are allowed in the Verilog standard for both tasks and functions. However, some tools require functions to be defined before the statements that invoke them.

```

module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output reg f;

  // Function that specifies a 4-to-1 multiplexer
  function mux4to1;
    input [0:3] X;
    input [1:0] S4;

    case (S4)
      0: mux4to1 = X[0];
      1: mux4to1 = X[1];
      2: mux4to1 = X[2];
      3: mux4to1 = X[3];
    endcase
  endfunction

  always @(W, S16)
    case (S16[3:2])
      0: f = mux4to1 (W[0:3], S16[1:0]);
      1: f = mux4to1 (W[4:7], S16[1:0]);
      2: f = mux4to1 (W[8:11], S16[1:0]);
      3: f = mux4to1 (W[12:15], S16[1:0]);
    endcase

endmodule

```

**Figure 6.48** The code from Figure 6.47 using a function.

---

## 6.7 CONCLUDING REMARKS

This chapter has introduced a number of circuit building blocks. Examples using these blocks to construct larger circuits will be presented in Chapters 7 and 10. To describe the building block circuits efficiently, several Verilog constructs have been introduced. In many cases a given circuit can be described in various ways, using different constructs. A circuit that can be described using an **if-else** statement can also be described using a **case** statement or perhaps a **for** loop. In general, there are no strict rules that dictate when one style should be preferred over another. With experience the user develops a sense for which types of statements work well in a particular design situation. Personal preference also influences how the code is written.

Verilog is not a programming language, and Verilog code should not be written as if it were a computer program. The statements discussed in this chapter can be used to create

large, complex circuits. A good way to design such circuits is to construct them using well-defined modules, in the manner that we illustrated for the multiplexers, decoders, encoders, and so on. Additional examples using the Verilog statements introduced in this chapter are given in Chapters 7 and 8. In Chapter 10 we provide a number of examples of using Verilog code to describe larger digital systems. For more information on Verilog, the reader can consult more specialized books [5–11].

In the next chapter we introduce logic circuits that include the ability to store logic signal values in memory elements.

## 6.8 EXAMPLES OF SOLVED PROBLEMS

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

**Example 6.28 Problem:** Implement the function  $f(w_1, w_2, w_3) = \sum m(0, 1, 3, 4, 6, 7)$  by using a 3-to-8 binary decoder and an OR gate.

**Solution:** The decoder generates a separate output for each minterm of the required function. These outputs are then combined in the OR gate, giving the circuit in Figure 6.49.

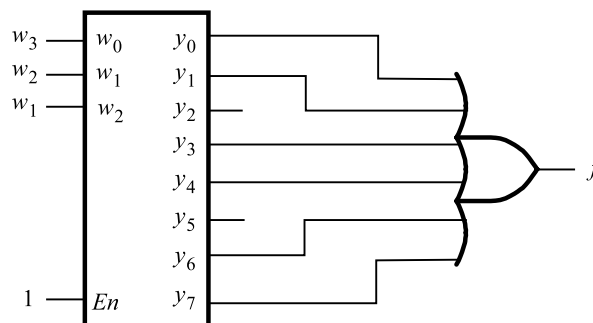
**Example 6.29 Problem:** Derive a circuit that implements an 8-to-3 binary encoder.

**Solution:** The truth table for the encoder is shown in Figure 6.50. Only those rows for which a single input variable is equal to 1 are shown; the other rows can be treated as don't care cases. From the truth table it is seen that the desired circuit is defined by the equations

$$y_2 = w_4 + w_5 + w_6 + w_7$$

$$y_1 = w_2 + w_3 + w_6 + w_7$$

$$y_0 = w_1 + w_3 + w_5 + w_7$$



**Figure 6.49** Circuit for Example 6.28.

$w_7$	$w_6$	$w_5$	$w_4$	$w_3$	$w_2$	$w_1$	$w_0$	$y_2$	$y_1$	$y_0$
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

**Figure 6.50** Truth table for an 8-to-3 binary encoder.

**Problem:** Implement the function

**Example 6.30**

$$f(w_1, w_2, w_3, w_4) = \bar{w}_1\bar{w}_2\bar{w}_4\bar{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

by using a 4-to-1 multiplexer and as few other gates as possible. Assume that only the uncomplemented inputs  $w_1, w_2, w_3,$  and  $w_4$  are available.

**Solution:** Since variables  $w_1$  and  $w_4$  appear in more product terms in the expression for  $f$  than the other three variables, let us perform Shannon's expansion with respect to these two variables. The expansion gives

$$\begin{aligned} f &= \bar{w}_1\bar{w}_4f_{\bar{w}_1\bar{w}_4} + \bar{w}_1w_4f_{\bar{w}_1w_4} + w_1\bar{w}_4f_{w_1\bar{w}_4} + w_1w_4f_{w_1w_4} \\ &= \bar{w}_1\bar{w}_4(\bar{w}_2\bar{w}_5) + \bar{w}_1w_4(w_3w_5) + w_1\bar{w}_4(w_2 + w_3) + w_1w_4(1) \end{aligned}$$

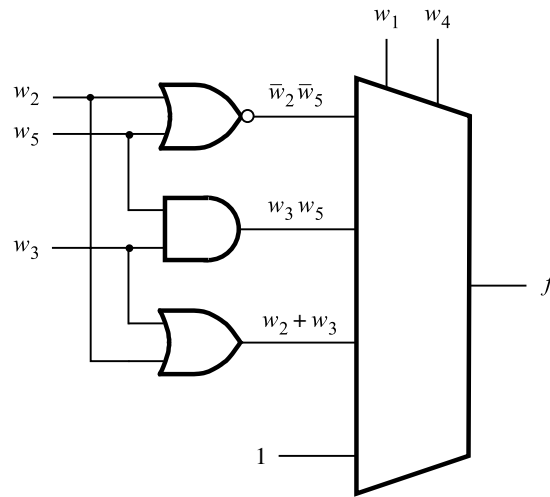
We can use a NOR gate to implement  $\bar{w}_2\bar{w}_5 = \overline{w_2 + w_5}$ . We also need an AND gate and an OR gate. The complete circuit is presented in Figure 6.51.

**Problem:** In Chapter 4 we pointed out that the rows and columns of a Karnaugh map are labeled using Gray code. This is a code in which consecutive valuations differ in one variable only. Figure 6.52 depicts the conversion between three-bit binary and Gray codes. Design a circuit that can convert a binary code into Gray code according to the figure.

**Example 6.31**

**Solution:** From the figure it follows that

$$\begin{aligned} g_2 &= b_2 \\ g_1 &= b_1\bar{b}_2 + \bar{b}_1b_2 \\ &= b_1 \oplus b_2 \\ g_0 &= b_0\bar{b}_1 + \bar{b}_0b_1 \\ &= b_0 \oplus b_1 \end{aligned}$$



**Figure 6.51** Circuit for Example 6.30.

$b_2$	$b_1$	$b_0$	$g_2$	$g_1$	$g_0$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

**Figure 6.52** Binary to Gray code conversion.

**Example 6.32 Problem:** In section 6.1.2 we showed that any logic function can be decomposed using Shannon's expansion theorem. For a four-variable function,  $f(w_1, \dots, w_4)$ , the expansion with respect to  $w_1$  is

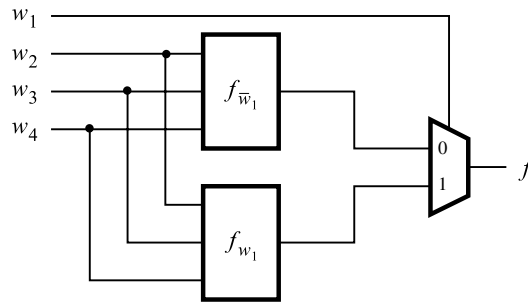
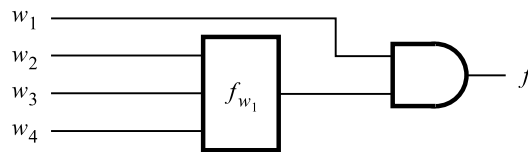
$$f(w_1, \dots, w_4) = \bar{w}_1 f_{\bar{w}_1} + w_1 f_{w_1}$$

A circuit that implements this expression is given in Figure 6.53a.

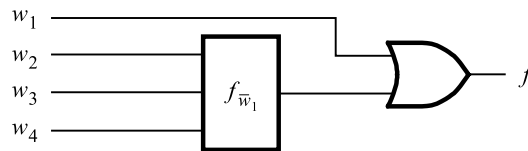
(a) If the decomposition yields  $f_{\bar{w}_1} = 0$ , then the multiplexer in the figure can be replaced by a single logic gate. Show this circuit.

(b) Repeat part (a) for the case where  $f_{w_1} = 1$ .

**Solution:** The desired circuits are shown in parts (b) and (c) of Figure 6.53.

(a) Shannon's expansion of the function  $f$ .

(b) Solution for part a



(c) Solution for part b

**Figure 6.53** Circuits for Example 6.32.

**Problem:** In several commercial FPGAs the logic blocks are 4-LUTs. What is the minimum number of 4-LUTs needed to construct a 4-to-1 multiplexer with select inputs  $s_1$  and  $s_0$  and data inputs  $w_3$ ,  $w_2$ ,  $w_1$ , and  $w_0$ ?

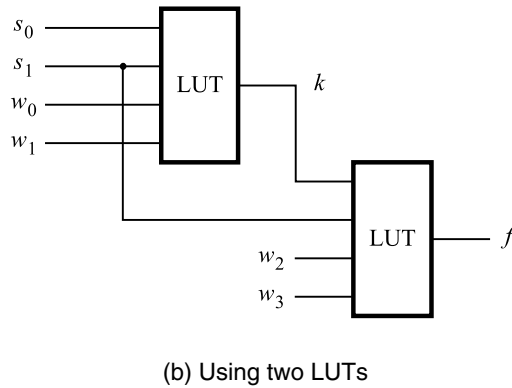
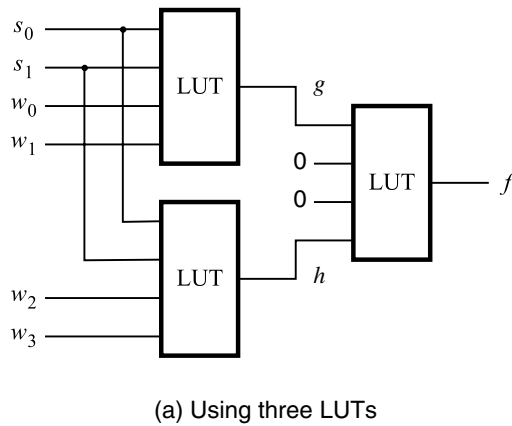
**Example 6.33**

**Solution:** A straightforward attempt is to use directly the expression that defines the 4-to-1 multiplexer

$$f = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1 + s_1\bar{s}_0w_2 + s_1s_0w_3$$

Let  $g = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1$  and  $h = s_1\bar{s}_0w_2 + s_1s_0w_3$ , so that  $f = g + h$ . This decomposition leads to the circuit in Figure 6.54a, which requires three LUTs.

When designing logic circuits, one can sometimes come up with a clever idea which leads to a superior implementation. Figure 6.54b shows how it is possible to implement



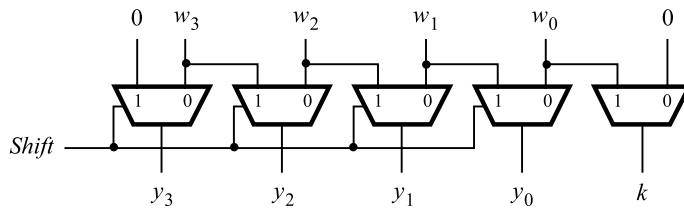
**Figure 6.54** Circuits for Example 6.33.

the multiplexer with just two LUTs, based on the following observation. The truth table in Figure 6.2b indicates that when  $s_1 = 0$  the output must be either  $w_0$  or  $w_1$ , as determined by the value of  $s_0$ . This can be generated by the first LUT. The second LUT must make the choice between  $w_2$  and  $w_3$  when  $s_1 = 1$ . But, the choice can be made only by knowing the value of  $s_0$ . Since it is impossible to have five inputs in the LUT, more information has to be passed from the first to the second LUT. Observe that when  $s_1 = 1$  the output  $f$  will be equal to either  $w_2$  or  $w_3$ , in which case it is not necessary to know the values of  $w_0$  or  $w_1$ . Hence, in this case we can pass on the value of  $s_0$  through the first LUT, rather than  $w_0$  or  $w_1$ . This can be done by making the function of this LUT

$$k = \bar{s}_1 \bar{s}_0 w_0 + \bar{s}_1 s_0 w_1 + s_1 s_0$$

Then, the second LUT performs the function

$$f = \bar{s}_1 k + s_1 \bar{k} w_3 + s_1 k w_4$$



**Figure 6.55** A shifter circuit.

**Problem:** In digital systems it is often necessary to have circuits that can shift the bits of a vector by one or more bit positions to the left or right. Design a circuit that can shift a four-bit vector  $W = w_3w_2w_1w_0$  one bit position to the right when a control signal *Shift* is equal to 1. Let the outputs of the circuit be a four-bit vector  $Y = y_3y_2y_1y_0$  and a signal  $k$ , such that if  $Shift = 1$  then  $y_3 = 0$ ,  $y_2 = w_3$ ,  $y_1 = w_2$ ,  $y_0 = w_1$ , and  $k = w_0$ . If  $Shift = 0$  then  $Y = W$  and  $k = 0$ . **Example 6.34**

**Solution:** The required circuit can be implemented with five 2-to-1 multiplexers as shown in Figure 6.55. The *Shift* signal is used as the select input to each multiplexer.

**Problem:** The shifter circuit in Example 6.34 shifts the bits of an input vector by one bit position to the right. It fills the vacated bit on the left side with 0. A more versatile shifter circuit may be able to shift by more bit positions at a time. If the bits that are shifted out are placed into the vacated positions on the left, then the circuit effectively rotates the bits of the input vector by a specified number of bit positions. Such a circuit is often called a *barrel shifter*. Design a four-bit barrel shifter that rotates the bits by 0, 1, 2, or 3 bit positions as determined by the valuation of two control signals  $s_1$  and  $s_0$ . **Example 6.35**

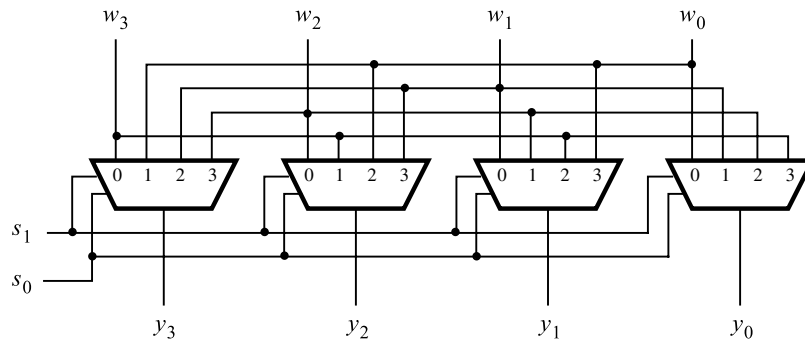
**Solution:** The required action is given in Figure 6.56a. The barrel shifter can be implemented with four 4-to-1 multiplexers as shown in Figure 6.56b. The control signals  $s_1$  and  $s_0$  are used as the select inputs to the multiplexers.

**Problem:** Write Verilog code that represents the circuit in Figure 6.19. Use the *dec2to4* module in Figure 6.35 as a subcircuit in your code. **Example 6.36**

**Solution:** The code is shown in Figure 6.57. Note that the *dec2to4* module can be included in the same file as we have done in the figure, but it can also be in a separate file in the project directory.

$s_1$	$s_0$	$y_3$	$y_2$	$y_1$	$y_0$
0	0	$w_3$	$w_2$	$w_1$	$w_0$
0	1	$w_0$	$w_3$	$w_2$	$w_1$
1	0	$w_1$	$w_0$	$w_3$	$w_2$
1	1	$w_2$	$w_1$	$w_0$	$w_3$

(a) Truth table



(b) Circuit

**Figure 6.56** A barrel shifter circuit.

**Example 6.37 Problem:** Write Verilog code that represents the shifter circuit in Figure 6.55.

**Solution:** One possibility is to specify the structure of this circuit as shown in Figure 6.58. The **if-else** construct is used to define the desired shifting of individual bits. A typical Verilog compiler will implement this code with 2-to-1 multiplexers as depicted in Figure 6.55.

An alternative is to make use of the shift operator defined in section 6.6.5, as indicated in Figure 6.59.

**Example 6.38 Problem:** Write Verilog code that defines the barrel shifter in Figure 6.56.

**Solution:** The code in Figure 6.60 is a possible solution. The rotate function is accomplished by concatenating two copies of the input vector  $W$  and shifting the obtained 8-bit vector to the right by the number of bit positions specified as the input  $S$ . The four least-significant bits of the resulting 8-bit vector are the desired output  $Y$ .

```

module mux4to1 (W, S, f);
  input [0:3] W;
  input [1:0] S;
  output f;
  wire [0:3] Y;

  dec2to4 decoder (S, Y, 1);
  assign f = |(W & Y);

endmodule

module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output reg [0:3] Y;

  always @(W, En)
    case ({En, W})
      3'b100: Y = 4'b1000;
      3'b101: Y = 4'b0100;
      3'b110: Y = 4'b0010;
      3'b111: Y = 4'b0001;
      default: Y = 4'b0000;
    endcase

endmodule

```

**Figure 6.57** Verilog code for Example 6.36.

---

## PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

- 6.1** Show how the function  $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 4, 5, 7)$  can be implemented using a 3-to-8 binary decoder and an OR gate.
- 6.2** Show how the function  $f(w_1, w_2, w_3) = \sum m(1, 2, 3, 5, 6)$  can be implemented using a 3-to-8 binary decoder and an OR gate.
- \*6.3** Consider the function  $f = \bar{w}_1\bar{w}_3 + w_2\bar{w}_3 + \bar{w}_1w_2$ . Use the truth table to derive a circuit for  $f$  that uses a 2-to-1 multiplexer.
- 6.4** Repeat problem 6.3 for the function  $f = \bar{w}_2\bar{w}_3 + w_1w_2$ .
- \*6.5** For the function  $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 6)$ , use Shannon's expansion to derive an implementation using a 2-to-1 multiplexer and any other necessary gates.

```

module shifter (W, Shift, Y, k);
  input [3:0] W;
  input Shift;
  output reg [3:0] Y;
  output reg k;

  always @(W, Shift)
  begin
    if (Shift)
      begin
        Y[3] = 0;
        Y[2:0] = W[3:1];
        k = W[0];
      end
    else
      begin
        Y = W;
        k = 0;
      end
    end
  end

endmodule

```

**Figure 6.58** Verilog code for the circuit in Figure 6.55.

- 6.6** Repeat problem 6.5 for the function  $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$ .
- 6.7** Consider the function  $f = \bar{w}_2 + \bar{w}_1 \bar{w}_3 + w_1 w_3$ . Show how repeated application of Shannon's expansion can be used to derive the minterms of  $f$ .
- 6.8** Repeat problem 6.7 for  $f = w_2 + \bar{w}_1 \bar{w}_3$ .
- 6.9** Prove Shannon's expansion theorem presented in section 6.1.2.
- \*6.10** Section 6.1.2 shows Shannon's expansion in sum-of-products form. Using the principle of duality, derive the equivalent expression in product-of-sums form.
- 6.11** Consider the function  $f = \bar{w}_1 \bar{w}_2 + \bar{w}_2 \bar{w}_3 + w_1 w_2 w_3$ . Give a circuit that implements  $f$  using the minimal number of two-input LUTs. Show the truth table implemented inside each LUT.
- \*6.12** For the function in problem 6.11, the cost of the minimal sum-of-products expression is 14, which includes four gates and 10 inputs to the gates. Use Shannon's expansion to derive a multilevel circuit that has a lower cost and give the cost of your circuit.
- 6.13** Consider the function  $f(w_1, w_2, w_3, w_4) = \sum m(0, 1, 3, 6, 8, 9, 14, 15)$ . Derive an implementation using the minimum possible number of three-input LUTs.
- \*6.14** Give two examples of logic functions with five inputs,  $w_1, \dots, w_5$ , that can be realized using 2 four-input LUTs.

```

module shifter (W, Shift, Y, k);
  input [3:0] W;
  input Shift;
  output reg [3:0] Y;
  output reg k;

  always @(W, Shift)
  begin
    if (Shift)
    begin
      Y = W >> 1;
      k = W[0];
    end
    else
    begin
      Y = W;
      k = 0;
    end
  end

endmodule

```

**Figure 6.59** Alternative Verilog code for the circuit in Figure 6.55.

```

module barrel (W, S, Y);
  input [3:0] W;
  input [1:0] S;
  output [3:0] Y;
  wire [3:0] T;

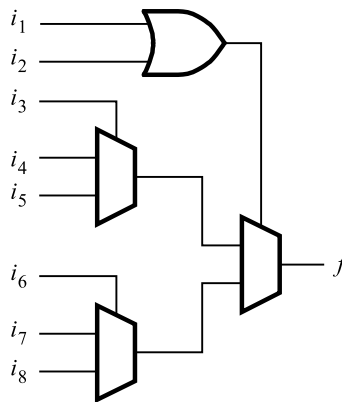
  assign {T, Y} = {W, W} >> S;

endmodule

```

**Figure 6.60** Verilog code for the barrel shifter.

- 6.15** For the function,  $f$ , in Example 6.30 perform Shannon's expansion with respect to variables  $w_1$  and  $w_2$ , rather than  $w_1$  and  $w_4$ . How does the resulting circuit compare with the circuit in Figure 6.51?
- 6.16** Actel Corporation manufactures an FPGA family called Act 1, which has the multiplexer-based logic block illustrated in Figure P6.1. Show how the function  $f = w_2\bar{w}_3 + w_1w_3 + \bar{w}_2w_3$  can be implemented using only one Act 1 logic block.



**Figure P6.1** The Actel Act 1 logic block.

**6.17** Show how the function  $f = w_1\bar{w}_3 + \bar{w}_1w_3 + w_2\bar{w}_3 + w_1\bar{w}_2$  can be realized using Act 1 logic blocks. Note that there are no NOT gates in the chip; hence complements of signals have to be generated using the multiplexers in the logic block.

**\*6.18** Consider the Verilog code in Figure P6.2. What type of circuit does the code represent? Comment on whether or not the style of code used is a good choice for the circuit that it represents.

```

module problem6_18 (W, En, y0, y1, y2, y3);
  input [1:0] W;
  input En;
  output reg y0, y1, y2, y3;

  always @(W, En)
  begin
    y0 = 0;
    y1 = 0;
    y2 = 0;
    y3 = 0;
    if (En)
      if (W == 0) y0 = 1;
      else if (W == 1) y1 = 1;
      else if (W == 2) y2 = 1;
      else y3 = 1;
  end

endmodule

```

**Figure P6.2** Code for problem 6.18.

- 6.19** Write Verilog code that represents the function in problem 6.2, using a **case** statement.
- 6.20** Write Verilog code for a 4-to-2 binary encoder.
- 6.21** Write Verilog code for an 8-to-3 binary encoder.
- 6.22** Figure P6.3 shows a modified version of the code for a 2-to-4 decoder in Figure 6.42. This code is almost correct but contains one error. What is the error?

```

module dec2to4 (W, Y, En);
    input [1:0] W;
    input En;
    output reg [0:3] Y;
    integer k;

    always @(W, En)
        for (k = 0; k <= 3; k = k+1)
            if (W == k)
                Y[k] = En;

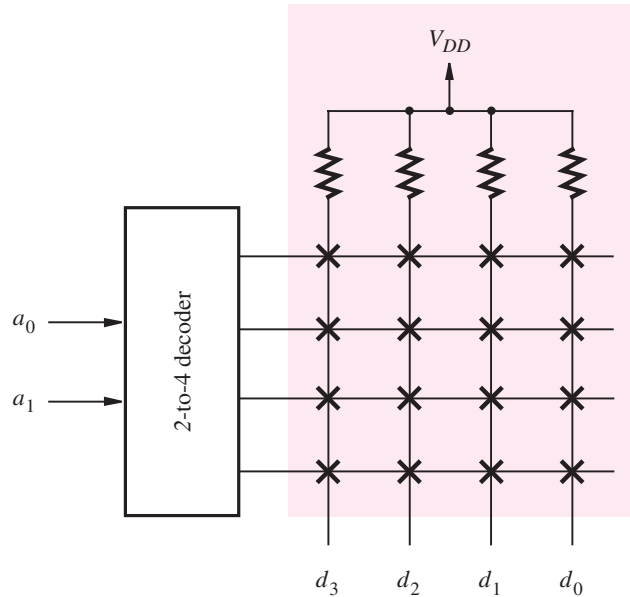
endmodule

```

**Figure P6.3** Code for problem 6.22.

- 6.23** Derive the circuit for an 8-to-3 priority encoder.
- 6.24** Using a **casex** statement, write Verilog code for an 8-to-3 priority encoder.
- 6.25** Repeat problem 6.24, using a **for** loop.
- 6.26** Create a Verilog module named *if2to4* that represents a 2-to-4 binary decoder using an **if-else** statement. Create a second module named *h3to8* that represents the 3-to-8 binary decoder in Figure 6.17 using two instances of the *if2to4* module.
- 6.27** Create a Verilog module named *h6to64* that represents a 6-to-64 binary decoder. Use the treelike structure in Figure 6.18, in which the 6-to-64 decoder is built using nine instances of the *h3to8* decoder created in problem 6.26.
- 6.28** Write Verilog code that represents the circuit in Figure 6.19. Use the *dec2to4* module in Figure 6.35 as a subcircuit in your code.
- \*6.29** Derive minimal sum-of-products expressions for the outputs *a*, *b*, and *c* of the 7-segment display in Figure 6.25.
- 6.30** Derive minimal sum-of-products expressions for the outputs *d*, *e*, *f*, and *g* of the 7-segment display in Figure 6.25.
- 6.31** Figure 6.21 shows a block diagram of a ROM. A circuit that implements a small ROM, with four rows and four columns, is depicted in Figure P6.4. Each X in the figure represents a switch that determines whether the ROM produces a 1 or 0 when that location is read.

- (a) Show how a switch ( $X$ ) can be realized using a single NMOS transistor.
- (b) Draw the complete  $4 \times 4$  ROM circuit, using your switches from part (a). The ROM should be programmed to store the bits 0101 in row 0 (the top row), 1010 in row 1, 1100 in row 2, and 0011 in row 3 (the bottom row).
- (c) Show how each ( $X$ ) can be implemented as a programmable switch (as opposed to providing either a 1 or 0 permanently), using an EEPROM cell as shown in Figure 3.64. Briefly describe how the storage cell is used.



**Figure P6.4** A  $4 \times 4$  ROM circuit.

- 6.32** Show the complete circuit for a ROM using the storage cells designed in Part (a) of problem 6.31 that realizes the logic functions

$$d_3 = a_0 \oplus a_1$$

$$d_2 = \overline{a_0 \oplus a_1}$$

$$d_1 = a_0 a_1$$

$$d_0 = a_0 + a_1$$

## REFERENCES

1. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits," *Transactions AIEE* 57 (1938), pp. 713–723.

2. Actel Corporation, "MX FPGA Data Sheet," <http://www.actel.com>.
3. QuickLogic Corporation, "pASIC 3 FPGA Data Sheet," <http://www.quicklogic.com>.
4. R. Landers, S. Mahant-Shetti, and C. Lemonds, "A Multiplexer-Based Architecture for High-Density, Low Power Gate Arrays," *IEEE Journal of Solid-State Circuits* 30, no. 4 (April 1995).
5. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
6. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
7. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).
8. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
9. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
10. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
11. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).