chapter

6

COMBINATIONAL-CIRCUIT BUILDING BLOCKS

CHAPTER OBJECTIVES

In this chapter you will learn about:

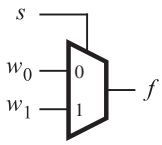
- Commonly used combinational subcircuits
- Multiplexers, which can be used for selection of signals and for implementation of general logic functions
- Circuits used for encoding, decoding, and code-conversion purposes
- Key VHDL constructs used to define combinational circuits

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 VHDL, 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.

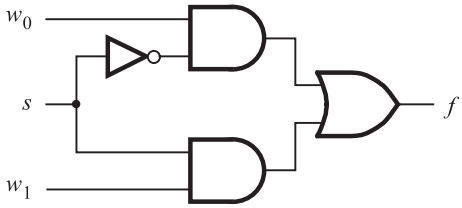
Figure 6.2*a* depicts a larger multiplexer with four data inputs, $w_0, \ldots, w_3$, and two select inputs, s_1 and s_0. As shown in the truth table in part (*b*) of the figure, the two-bit number represented by $s_1 s_0$ selects one of the data inputs as the output of the multiplexer.



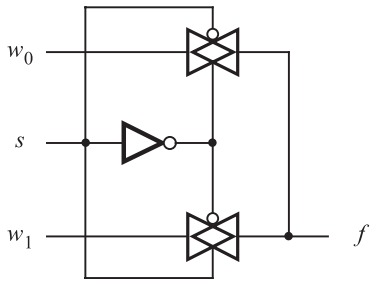
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

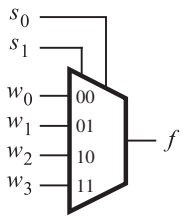


(c) Sum-of-products circuit



(d) Circuit with transmission gates

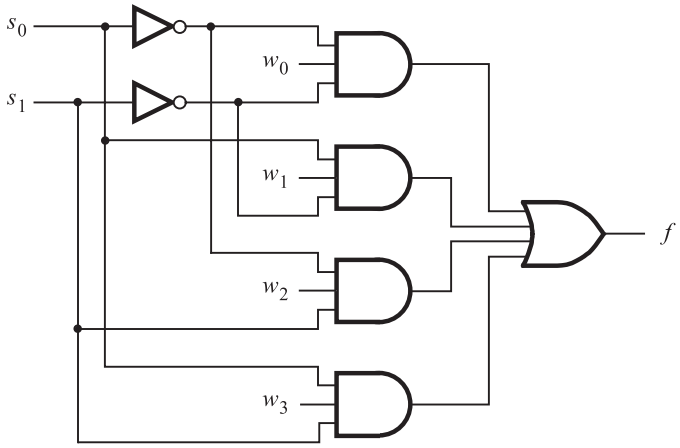
Figure 6.1 A 2-to-1 multiplexer.



(a) Graphical symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table



(c) Circuit

Figure 6.2 A 4-to-1 multiplexer.

A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 6.2*c*. It 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, $w_0, \ldots, w_{n-1}$, 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.

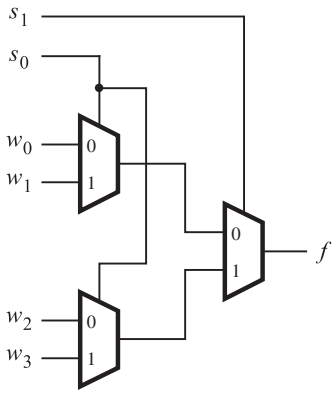


Figure 6.3 Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

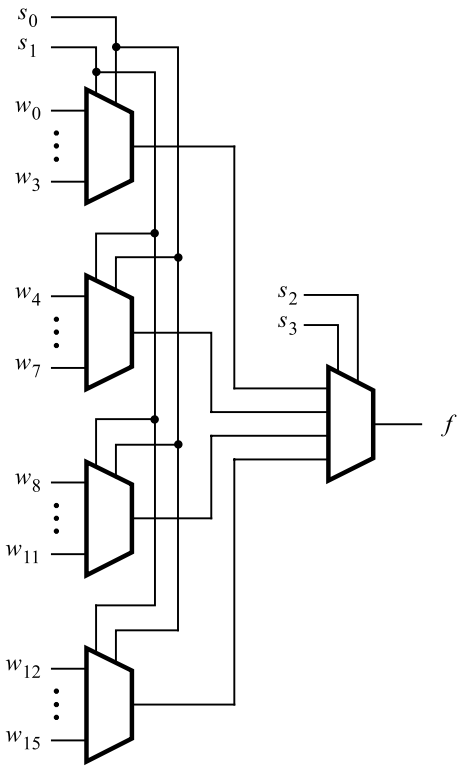


Figure 6.4 A 16-to-1 multiplexer.

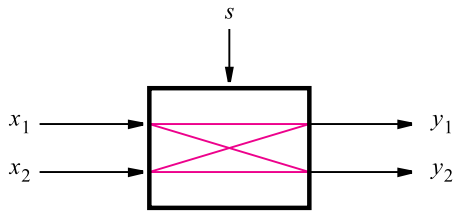
Example 6.1

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 2×2 crossbar.

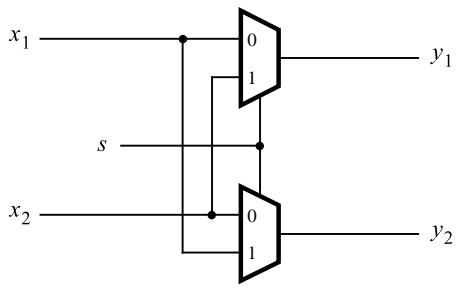
Figure 6.5*b* shows how the 2×2 crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal s. If $s = 0$, the crossbar connects x_1 to y_1 and x_2 to y_2, while if $s = 1$, the crossbar connects x_1 to y_2 and x_2 to y_1. Crossbar switches are useful 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.

Example 6.2

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 6.6*a*. For clarity,

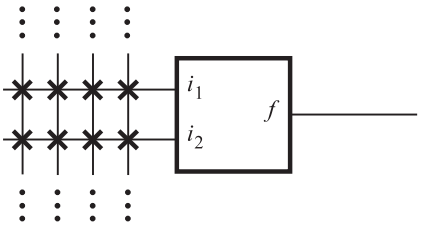


(a) A 2x2 crossbar switch

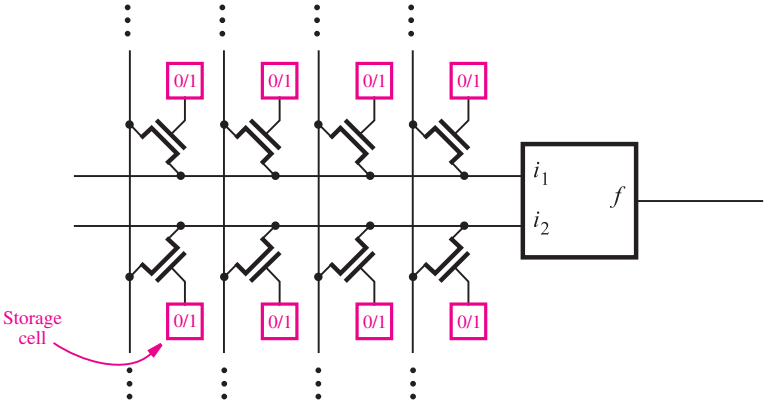


(b) Implementation using multiplexers

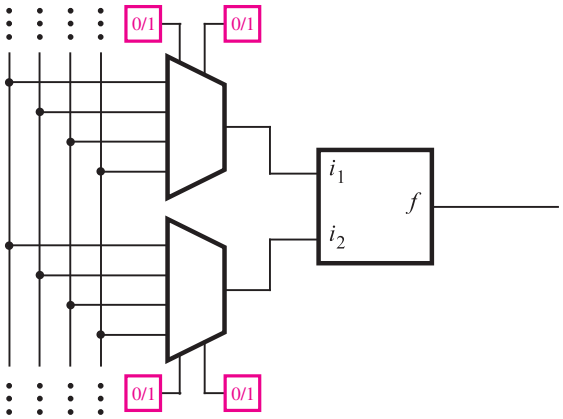
Figure 6.5 A practical application of multiplexers.



(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.

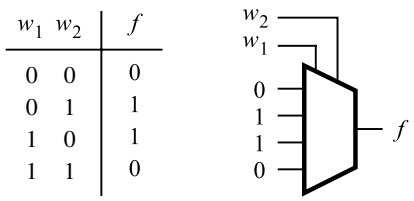
6.1.1 SYNTHESIS OF LOGIC FUNCTIONS USING MULTIPLEXERS

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_1w_2, the output f is equal to the function value in the corresponding row of the truth table.

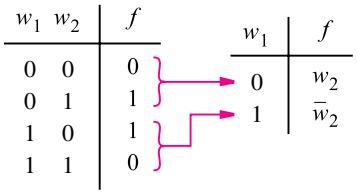
The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 6.7*b*, which allows f to be implemented by a single 2-to-1 multiplexer. One of the input signals, w_1 in this example, is chosen as 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 $\overline{w}_2$. 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.

Example 6.3

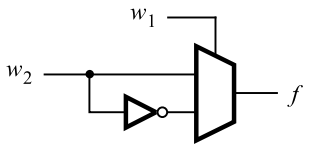
Figure 6.8*a* gives the truth table for the three-input majority function, and it shows how the truth table can be modified to implement the function using a 4-to-1 multiplexer. Any two of the three inputs may be chosen as the multiplexer select inputs. We have chosen w_1 and w_2 for this purpose, resulting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



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



(b) Modified truth table

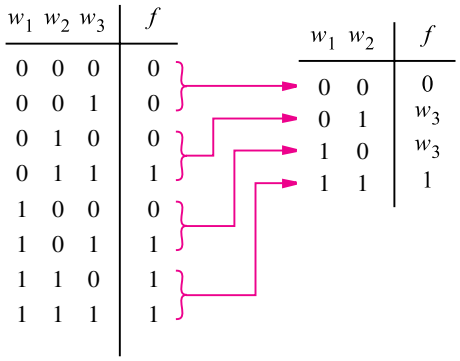


(c) Circuit

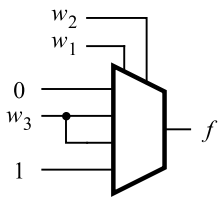
Figure 6.7 Synthesis of a logic function using mutiplexers.

Example 6.4 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 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.

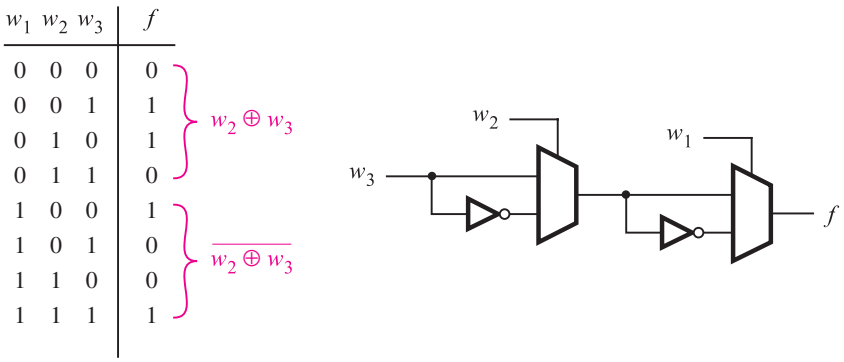


(a) Modified truth table



(b) Circuit

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

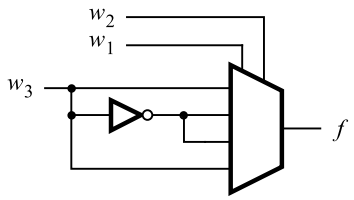


(a) Truth table (b) Circuit

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

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w}_3$
0	1	1	0	
1	0	0	1	$\overline{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



(b) Circuit

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

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 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 6.11*b*.

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

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

It can be manipulated into

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

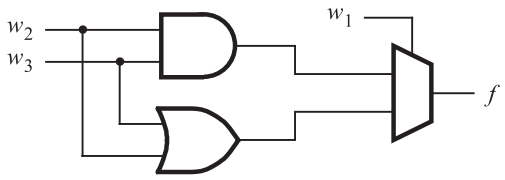
which corresponds to the circuit in Figure 6.11*b*.

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].

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_2w_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.

Shannon's Expansion Theorem

Any Boolean function $f(w_1, \ldots, w_n)$ can be written in the form

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

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).

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

$$f(w_1, w_2, w_3) = w_1w_2 + w_1w_3 + w_2w_3$$

Expanding this function in terms of w_1 gives

$$f = \overline{w}_1(w_2w_3) + w_1(w_2 + w_3)$$

which is the expression that we derived above.

For the three-input XOR function, we have

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

which gives the circuit in Figure 6.9*b*.

In Shannon's expansion the term $f(0, w_2, \ldots, w_n)$ is called the *cofactor* of f with respect to $\overline{w}_1$; it is denoted in shorthand notation as $f_{\overline{w}_1}$. Similarly, the term $f(1, w_2, \ldots, w_n)$ is called the cofactor of f with respect to w_1, written f_{w_1}. Hence we can write

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

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

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

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 $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, decomposition using w_1 gives

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

Using w_2 instead of w_1 produces

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

Finally, using w_3 gives

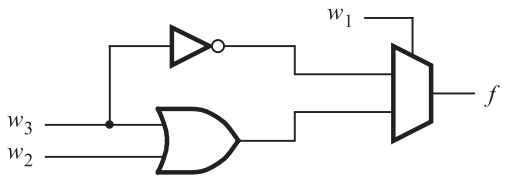
$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3 (w_2) + w_3 (\overline{w}_1) \end{aligned}$$

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.

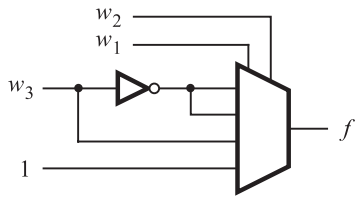
Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of w_1 and w_2 gives

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

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.

Example 6.6

Assume that we wish to implement the function

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

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

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

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

$$\begin{aligned} f &= \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1 w_2 f_{w_1 w_2} \\ &= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1 w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1 w_2(1) \end{aligned}$$

The circuit is shown in Figure 6.12*b*.

Example 6.7

Consider the three-input majority function

$$f = w_1w_2 + w_1w_3 + w_2w_3$$

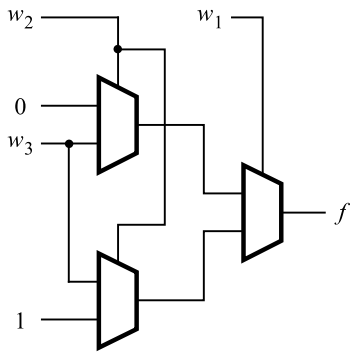


Figure 6.13 The circuit synthesized in Example 6.7.

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

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

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

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

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

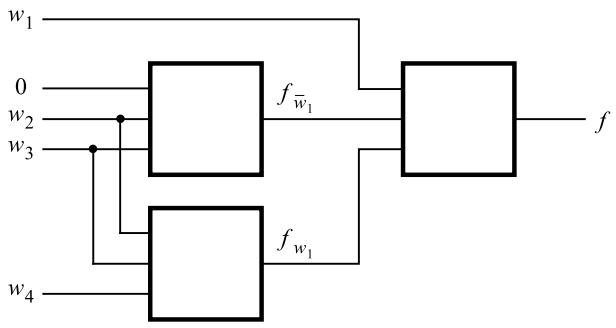
$$f = \overline{w}_2w_3 + \overline{w}_1w_2\overline{w}_3 + w_2\overline{w}_3w_4 + w_1\overline{w}_2\overline{w}_4$$

Expansion in terms of w_1 produces

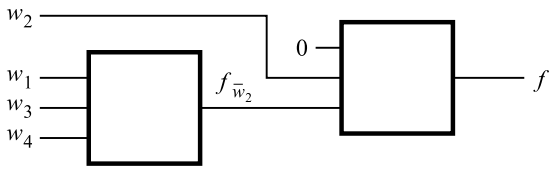
$$\begin{aligned} f &= \overline{w}_1f_{\overline{w}_1} + w_1f_{w_1} \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2\overline{w}_3w_4) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3) + w_1(\overline{w}_2w_3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

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

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



(a) Using three 3-LUTs



(b) Using two 3-LUTs

Figure 6.14 Circuits synthesized in Example 6.8.

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

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

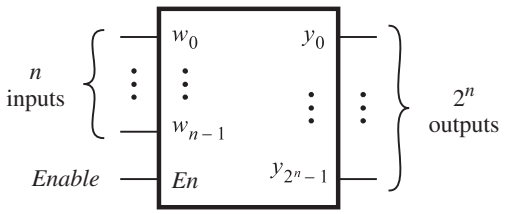


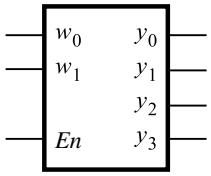
Figure 6.15 An n-to-2^n binary decoder.

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.

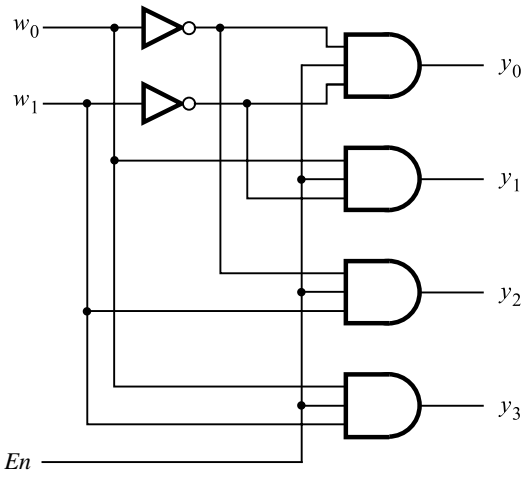
A 2-to-4 decoder is given in Figure 6.16. The two data inputs are w_1 and w_0. They represent a two-bit number that causes the decoder to assert one of the outputs $y_0, \ldots, y_3$. Although a decoder can be designed to have either active-high or active-low outputs, in

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.

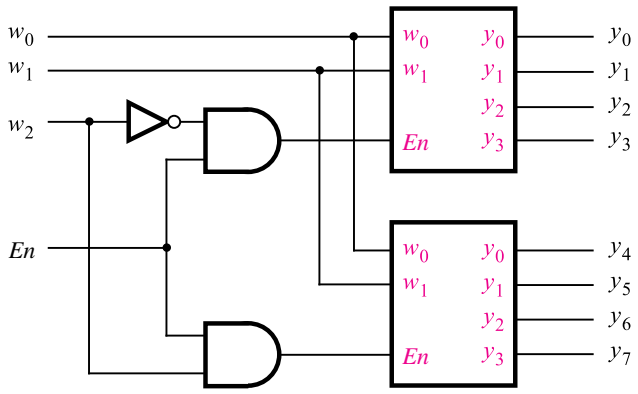


Figure 6.17 A 3-to-8 decoder using two 2-to-4 decoders.

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 6.16*c*, 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*.

Example 6.9

Decoders are useful for many practical purposes. In Figure 6.2*c* 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 6.10

In 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

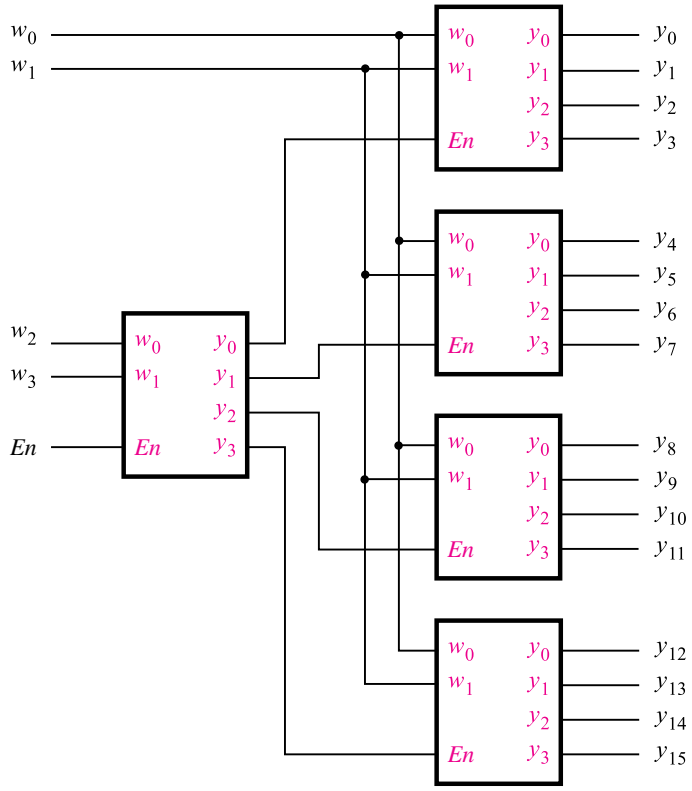


Figure 6.18 A 4-to-16 decoder built using a decoder tree.

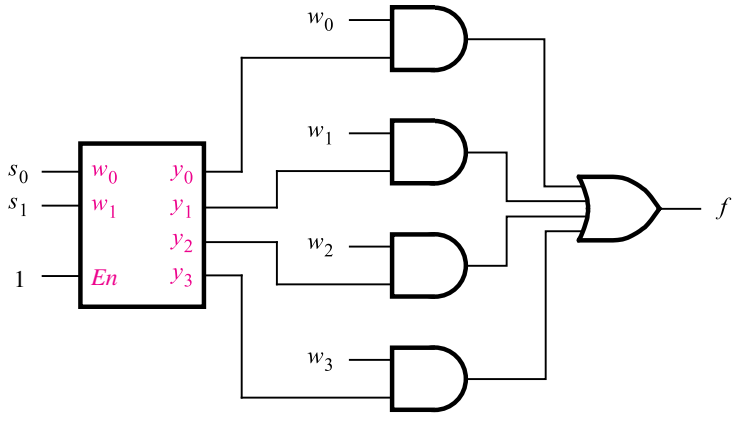


Figure 6.19 A 4-to-1 multiplexer built using a decoder.

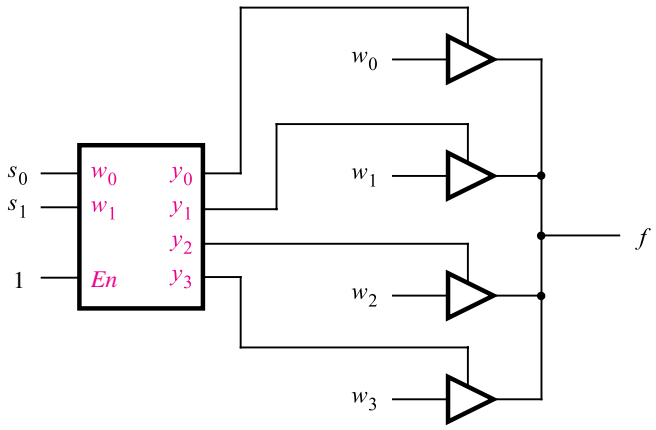


Figure 6.20 A 4-to-1 multiplexer built using a decoder and tri-state buffers.

buffers. Hence multiplexers must be implemented in the sum-of-products form using the lookup tables (see Example 6.30).

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 6.16*a*. 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, $y_0, \ldots, y_{2^n-1}$, 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, Sel_0 to 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, $d_{n-1}, \ldots, d_0$, 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.

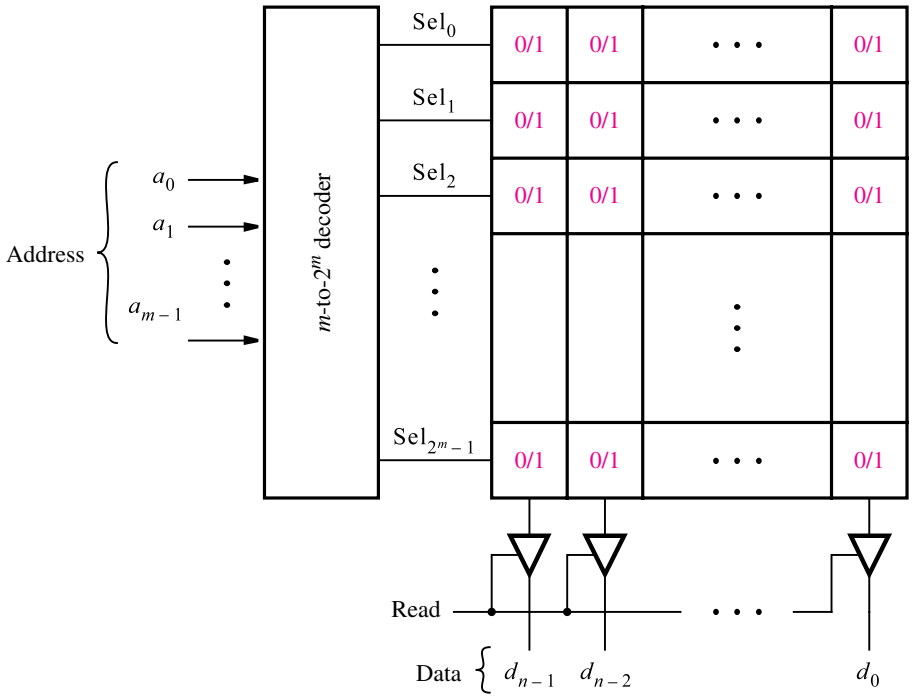


Figure 6.21 A $2^m \times n$ read-only memory (ROM) block.

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 6.32). 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 6.23*a*. 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 6.23*b*. 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.

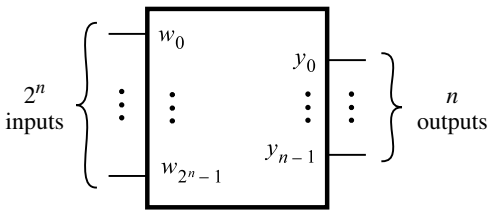
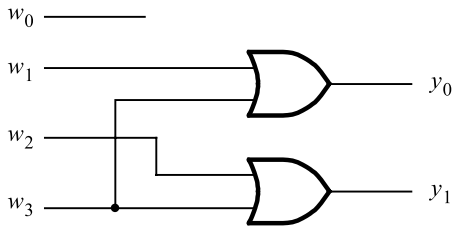


Figure 6.22 A 2^n-to-n binary encoder.

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, $i_0, \ldots, i_3$, 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 $i_0, \ldots, i_3$ are

$$i_0 = \overline{w}_3\overline{w}_2\overline{w}_1w_0$$
$$i_1 = \overline{w}_3\overline{w}_2w_1$$
$$i_2 = \overline{w}_3w_2$$
$$i_3 = w_3$$

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

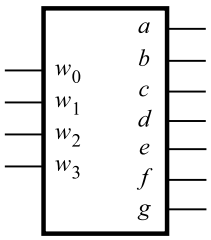
$$y_0 = i_1 + i_3$$
$$y_1 = i_2 + i_3$$

The output z is given by

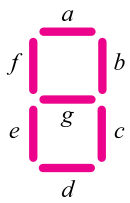
$$z = i_0 + i_1 + i_2 + i_3$$

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 6.25*a*, 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 6.25*c*. For each valuation of the inputs $w_3, \ldots, w_0$, 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, $i_k = \overline{a_k \oplus b_k}$. 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

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

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 VHDL FOR COMBINATIONAL CIRCUITS

Having presented a number of useful circuits that can be used as building blocks in larger circuits, we will now consider how such circuits can be described in VHDL. Rather than relying on the simple VHDL statements used in previous examples, such as logic expressions, we will specify the circuits in terms of their behavior. We will also introduce a number of new VHDL constructs.

6.6.1 ASSIGNMENT STATEMENTS

VHDL provides several types of statements that can be used to assign logic values to signals. In the examples of VHDL code given so far, only simple assignment statements have been used, either for logic or arithmetic expressions. This section introduces other types of

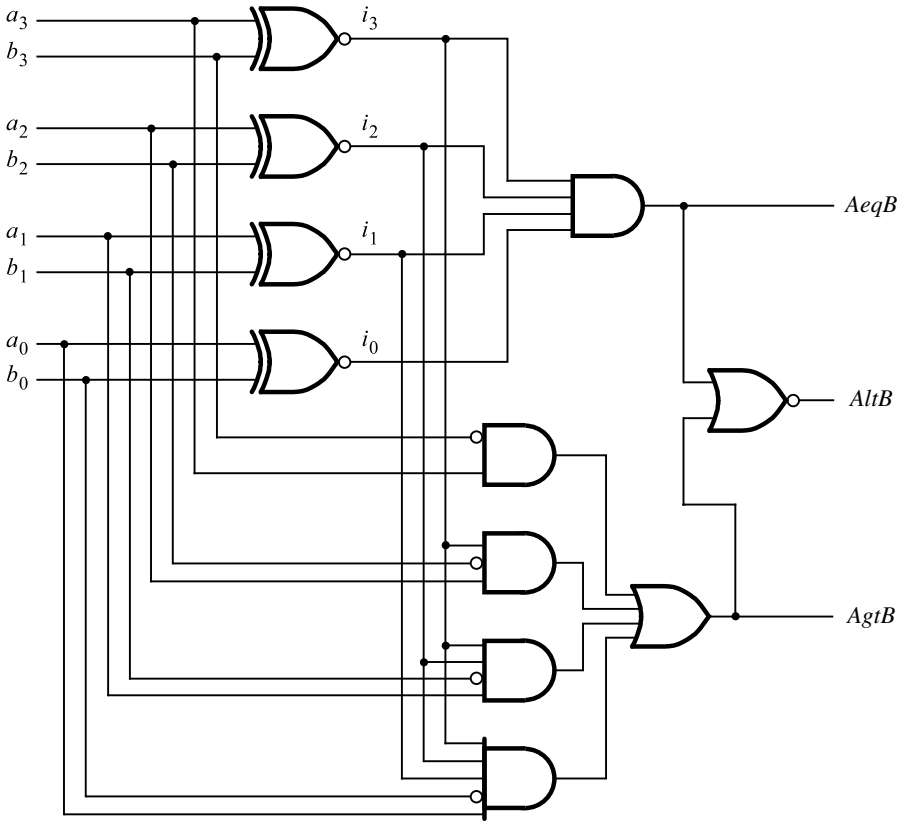


Figure 6.26 A four-bit comparator circuit.

assignment statements, which are called selected signal assignments, conditional signal assignments, generate statements, if-then-else statements, and case statements.

6.6.2 SELECTED SIGNAL ASSIGNMENT

A selected signal assignment allows a signal to be assigned one of several values, based on a selection criterion. Figure 6.27 shows how it can be used to describe a 2-to-1 multiplexer. The entity, named *mux2to1*, has the inputs w_0, w_1, and s, and the output f. The selected signal assignment begins with the keyword WITH, which specifies that s is to be used for the selection criterion. The two WHEN clauses state that f is assigned the value of w_0 when $s = 0$; otherwise, f is assigned the value of w_1. The WHEN clause that selects w_1 uses the word OTHERS, instead of the value 1. This is required because the VHDL syntax specifies that a WHEN clause must be included for every possible value of the selection signal s.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    WITH s SELECT
        f <=  w0 WHEN '0',
              w1 WHEN OTHERS ;
END Behavior ;
```

Figure 6.27 VHDL code for a 2-to-1 multiplexer.

Since it has the STD_LOGIC type, discussed in section 4.12, s can take the values 0, 1, Z, −, and others. The keyword OTHERS provides a convenient way of accounting for all logic values that are not explicitly listed in a WHEN clause.

Example 6.12

A 4-to-1 multiplexer is described by the entity named *mux4to1*, shown in Figure 6.28. The two select inputs, which are called s_1 and s_0 in Figure 6.2, are represented by the two-bit STD_LOGIC_VECTOR signal s. The selected signal assignment sets f to the value of one of the inputs $w_0, \ldots, w_3$, depending on the valuation of s. Compiling the code results in the circuit shown in Figure 6.2*c*. At the end of Figure 6.28, the *mux4to1* entity is defined as a component in the package named *mux4to1_package*. We showed in section 5.5.2 that the component declaration allows the entity to be used as a subcircuit in other VHDL code.

Example 6.13

Figure 6.4 showed how a 16-to-1 multiplexer is built using five 4-to-1 multiplexers. Figure 6.29 presents VHDL code for this circuit, using the *mux4to1* component. The lines of code are numbered so that we can easily refer to them. The *mux4to1_package* is included in the code, because it provides the component declaration for *mux4to1*.

The data inputs to the *mux16to1* entity are the 16-bit signal named w, and the select inputs are the four-bit signal named s. In the VHDL code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left of Figure 6.4. Line 11 defines a four-bit signal named m for this purpose, and lines 13 to 16 instantiate the four multiplexers. For instance, line 13 corresponds to the multiplexer at the top left of Figure 6.4. Its first four ports, which correspond to $w_0, \ldots, w_3$ in Figure 6.28, are driven by the signals $w(0), \ldots, w(3)$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( w0, w1, w2, w3  : IN   STD_LOGIC ;
           s               : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           f               : OUT  STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
    WITH s SELECT
        f <=  w0 WHEN "00",
              w1 WHEN "01",
              w2 WHEN "10",
              w3 WHEN OTHERS ;
END Behavior ;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
PACKAGE mux4to1_package IS
    COMPONENT mux4to1
        PORT ( w0, w1, w2, w3  : IN   STD_LOGIC ;
               s               : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               f               : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END mux4to1_package ;
```

Figure 6.28 VHDL code for a 4-to-1 multiplexer.

The syntax s(1 DOWNTO 0) is used to attach the signals $s(1)$ and $s(0)$ to the two-bit s port of the *mux4to1* component. The $m(0)$ signal is connected to the multiplexer's output port.

Line 17 instantiates the multiplexer on the right of Figure 6.4. The signals $m_0, \ldots, m_3$ are connected to its data inputs, and bits $s(3)$ and $s(2)$, which are specified by the syntax s(3 DOWNTO 2), are attached to the select inputs. The output port generates the *mux16to1* output f. Compiling the code results in the multiplexer function

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

Example 6.14 The selected signal assignments can also be used to describe other types of circuits. Figure 6.30 shows how a selected signal assignment can be used to describe the truth table for a 2-to-4 binary decoder. The entity is called *dec2to4*. The data inputs are the two-bit signal

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   LIBRARY work ;
4   USE work.mux4to1_package.all ;

5   ENTITY mux16to1 IS
6       PORT ( w  : IN    STD_LOGIC_VECTOR(0 TO 15) ;
7              s  : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
8              f  : OUT   STD_LOGIC ) ;
9   END mux16to1 ;

10  ARCHITECTURE Structure OF mux16to1 IS
11      SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
12  BEGIN
13      Mux1: mux4to1 PORT MAP
            ( w(0), w(1), w(2), w(3), s(1 DOWNTO 0), m(0) ) ;
14      Mux2: mux4to1 PORT MAP
            ( w(4), w(5), w(6), w(7), s(1 DOWNTO 0), m(1) ) ;
15      Mux3: mux4to1 PORT MAP
            ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
16      Mux4: mux4to1 PORT MAP
            ( w(12), w(13), w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
17      Mux5: mux4to1 PORT MAP
            ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
18  END Structure ;
```

Figure 6.29 Hierarchical code for a 16-to-1 multiplexer.

named w, and the enable input is En. The four outputs are represented by the four-bit signal y.

In the truth table for the decoder in Figure 6.16*a*, the inputs are listed in the order $En\ w_1w_0$. To represent these three signals, the VHDL code defines the three-bit signal named *Enw*. The statement *Enw* <= *En* & *w* uses the VHDL concatenate operator, which was discussed in section 5.5.4, to combine the *En* and *w* signals into the *Enw* signal. Hence $Enw(2) = En$, $Enw(1) = w_1$, and $Enw(0) = w_0$. The *Enw* signal is used as the selection signal in the selected signal assignment statement. It describes the truth table in Figure 6.16*a*. In the first four WHEN clauses, $En = 1$, and the decoder outputs have the same patterns as in the first four rows of the truth table. The last WHEN clause uses the OTHERS keyword and sets the decoder outputs to 0000, because it represents the cases where $En = 0$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <=  "1000" WHEN "100",
              "0100" WHEN "101",
              "0010" WHEN "110",
              "0001" WHEN "111",
              "0000" WHEN OTHERS ;
END Behavior ;
```

Figure 6.30 VHDL code for a 2-to-4 binary decoder.

6.6.3 CONDITIONAL SIGNAL ASSIGNMENT

Similar to the selected signal assignment, a conditional signal assignment allows a signal to be set to one of several values. Figure 6.31 shows a modified version of the 2-to-1 multiplexer entity from Figure 6.27. It uses a conditional signal assignment to specify that f is assigned the value of w_0 when $s = 0$, or else f is assigned the value of w_1. Compiling

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    f <= w0 WHEN s = '0' ELSE w1 ;
END Behavior ;
```

Figure 6.31 Specification of a 2-to-1 multiplexer using a conditional signal assignment.

the code generates the same circuit as the code in Figure 6.27. In this small example the conditional signal assignment has only one WHEN clause. A more complex example, which better illustrates the features of the conditional signal assignment, is given in Example 6.15.

Example 6.15

Figure 6.24 gives the truth table for a 4-to-2 priority encoder. VHDL code that describes this truth table is shown in Figure 6.32. The inputs to the encoder are represented by the four-bit signal named w. The encoder has the outputs y, which is a two-bit signal, and z.

The conditional signal assignment specifies that y is assigned the value 11 when input $w(3) = 1$. If this condition is true, then the other WHEN clauses that follow the ELSE keyword do not affect the value of f. Hence the values of $w(2)$, $w(1)$, and $w(0)$ do not matter, which implements the desired priority scheme. The second WHEN clause states that when $w(2) = 1$, then y is assigned the value 10. This can occur only if $w(3) = 0$. Each successive WHEN clause can affect y only if none of the conditions associated with the preceding WHEN clauses are true. Figure 6.32 includes a second conditional signal assignment for the output z. It states that when all four inputs are 0, z is assigned the value 0; else z is assigned the value 1.

The priority level associated with each WHEN clause in the conditional signal assignment is a key difference from the selected signal assignment, which has no such priority scheme. It is possible to describe the priority encoder using a selected signal assignment, but the code is more awkward. One possibility is shown by the architecture in Figure 6.33. The first WHEN clause sets y to 00 when w_0 is the only input that is 1. The next two clauses state that y should be 01 when $w_3 = w_2 = 0$ and $w_1 = 1$. The next four clauses specify that y should be 10 if $w_3 = 0$ and $w_2 = 1$. Finally, the last WHEN clause states that y should be 1 for all other input valuations, which includes all valuations for which w_3 is 1. Note that

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    y <= "11" WHEN w(3) = '1' ELSE
         "10" WHEN w(2) = '1' ELSE
         "01" WHEN w(1) = '1' ELSE
         "00" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figure 6.32 VHDL code for a priority encoder.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    WITH w SELECT
        y <= "00" WHEN "0001",
             "01" WHEN "0010",
             "01" WHEN "0011",
             "10" WHEN "0100",
             "10" WHEN "0101",
             "10" WHEN "0110",
             "10" WHEN "0111",
             "11" WHEN OTHERS ;
    WITH w SELECT
        z <= '0' WHEN "0000",
             '1' WHEN OTHERS ;
END Behavior ;
```

Figure 6.33 Less efficient code for a priority encoder.

the OTHERS clause includes the input valuation 0000. This pattern results in $z = 0$, and the value of y does not matter in this case.

Example 6.16 We derived the circuit for a comparator in Figure 6.26. Figure 6.34 shows how this circuit can be described with VHDL code. Each of the three conditional signal assignments determines the value of one of the comparator outputs. The package named *std_logic_unsigned* is included in the code because it specifies that STD_LOGIC_VECTOR signals, namely, A and B, can be used as unsigned binary numbers with VHDL relational operators. The relational operators provide a convenient way of specifying the desired functionality.

The circuit generated from the code in Figure 6.34 is similar, but not identical, to the circuit in Figure 6.26. The VHDL compiler instantiates a predefined module to implement each of the comparison operations. In Quartus II the modules that are instantiated are from the LPM library, which was introduced in section 5.5.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY compare IS
    PORT ( A, B                 : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           AeqB, AgtB, AltB     : OUT   STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
    AeqB <= '1' WHEN A = B ELSE '0' ;
    AgtB <= '1' WHEN A > B ELSE '0' ;
    AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figure 6.34 VHDL code for a four-bit comparator.

Instead of using the *std_logic_unsigned* library, another way to specify that the generated circuit should use unsigned numbers is to include the library named *std_logic_arith.* In this case the signals *A* and *B* should be defined with the type UNSIGNED, rather than STD_LOGIC_VECTOR. If we want the circuit to work with signed numbers, signals *A* and *B* should be defined with the type SIGNED. This code is given in Figure 6.35.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY compare IS
    PORT ( A, B                 : IN    SIGNED(3 DOWNTO 0) ;
           AeqB, AgtB, AltB     : OUT   STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
    AeqB <= '1' WHEN A = B ELSE '0' ;
    AgtB <= '1' WHEN A > B ELSE '0' ;
    AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figure 6.35 The code from Figure 6.34 for signed numbers.

6.6.4 GENERATE STATEMENTS

Figure 6.29 gives VHDL code for a 16-to-1 multiplexer using five instances of a 4-to-1 multiplexer subcircuit. The regular structure of the code suggests that it could be written in a more compact form using a loop. VHDL provides a feature called the FOR GENERATE statement for describing regularly structured hierarchical code.

Figure 6.36 shows the code from Figure 6.29 rewritten using a FOR GENERATE statement. The generate statement must have a label, so we have used the label *G1* in the code. The loop instantiates four copies of the *mux4to1* component, using the loop index i in the range from 0 to 3. The variable i is not explicitly declared in the code; it is automatically defined as a local variable whose scope is limited to the FOR GENERATE statement. The first loop iteration corresponds to the instantiation statement labeled *Mux1* in Figure 6.29. The * operator represents multiplication; hence for the first loop iteration the VHDL compiler translates the signal names $w(4 * i)$, $w(4 * i + 1)$, $w(4 * i + 2)$, and $w(4 * i + 3)$ into signal names $w(0)$, $w(1)$, $w(2)$, and $w(3)$. The loop iterations for $i = 1$, $i = 2$, and $i = 3$ correspond to the statements labeled *Mux2*, *Mux3*, and *Mux4* in Figure 6.29. The statement labeled *Mux5* in Figure 6.29 does not fit within the loop, so it is included as a separate statement in Figure 6.36. The circuit generated from the code in Figure 6.36 is identical to the circuit produced by using the code in Figure 6.29.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.mux4to1_package.all ;

ENTITY mux16to1 IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(0 TO 15) ;
           s  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f  : OUT  STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Muxes: mux4to1 PORT MAP (
            w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
    END GENERATE ;
    Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

Figure 6.36 Code for a 16-to-1 multiplexer using a generate statement.

Example 6.17

In addition to the FOR GENERATE statement, VHDL provides another type of generate statement called IF GENERATE. Figure 6.37 illustrates the use of both types of generate statements. The code shown is a hierarchical description of the 4-to-16 decoder given in Figure 6.18, using five instances of the *dec2to4* component defined in Figure 6.30. The decoder inputs are the four-bit signal *w*, the enable is *En*, and the outputs are the 16-bit signal *y*.

Following the component declaration for the *dec2to4* subcircuit, the architecture defines the signal *m*, which represents the outputs of the 2-to-4 decoder on the left of Figure 6.18. The five copies of the *dec2to4* component are instantiated by the FOR GENERATE statement. In each iteration of the loop, the statement labeled *Dec_ri* instantiates a *dec2to4* component that corresponds to one of the 2-to-4 decoders on the right side of Figure 6.18. The first loop iteration generates the *dec2to4* component with data inputs w_1 and w_0, enable input m_0, and outputs y_0, y_1, y_2, y_3. The other loop iterations also use data inputs w_1w_0, but use different bits of *m* and *y*.

The IF GENERATE statement, labeled *G2*, instantiates a *dec2to4* component in the last loop iteration, for which the condition $i = 3$ is true. This component represents the 2-to-4 decoder on the left of Figure 6.18. It has the two-bit data inputs w_3 and w_2, the enable *En*, and

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec4to16 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 15) ) ;
END dec4to16 ;

ARCHITECTURE Structure OF dec4to16 IS
    COMPONENT dec2to4
        PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En  : IN    STD_LOGIC ;
               y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
    END COMPONENT ;
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Dec_ri: dec2to4 PORT MAP ( w(1 DOWNTO 0), m(i), y(4*i TO 4*i+3) );
        G2: IF i=3 GENERATE
            Dec_left: dec2to4 PORT MAP ( w(i DOWNTO i-1), En, m ) ;
        END GENERATE ;
    END GENERATE ;
END Structure ;
```

Figure 6.37 Hierarchical code for a 4-to-16 binary decoder.

the outputs m_0, m_1, m_2, and m_3. Note that instead of using the IF GENERATE statement, we could have instantiated this component outside the FOR GENERATE statement. We have written the code as shown simply to give an example of the IF GENERATE statement.

The generate statements in Figures 6.36 and 6.37 are used to instantiate components. Another use of generate statements is to generate a set of logic equations. An example of this use will be given in Figure 7.73.

6.6.5 CONCURRENT AND SEQUENTIAL ASSIGNMENT STATEMENTS

We have introduced several types of assignment statements: simple assignment statements, which involve logic or arithmetic expressions, selected assignment statements, and conditional assignment statements. All of these statements share the property that the order in which they appear in VHDL code does not affect the meaning of the code. Because of this property, these statements are called the *concurrent assignment statements*.

VHDL also provides a second category of statements, called *sequential assignment statements*, for which the ordering of the statements may affect the meaning of the code. We will discuss two types of sequential assignment statements, called if-then-else statements and case statements. VHDL requires that the sequential assignment statements be placed inside another type of statement, called a process statement.

6.6.6 PROCESS STATEMENT

Figures 6.27 and 6.31 show two ways of describing a 2-to-1 multiplexer, using the selected and conditional signal assignments. The same circuit can also be described using an if-then-else statement, but this statement must be placed inside a process statement. Figure 6.38 shows such code. The process statement, or simply *process*, begins with the PROCESS keyword, followed by a parenthesized list of signals, called the *sensitivity list*. For a combinational circuit like the multiplexer, the sensitivity list includes all input signals that are used inside the process. The process statement is translated by the VHDL compiler into logic equations. In the figure the process consists of the single if-then-else statement that describes the multiplexer function. Thus the sensitivity list comprises the data inputs, w_0 and w_1, and the select input s.

In general, there can be a number of statements inside a process. These statements are considered as follows. Using VHDL jargon, we say that when there is a change in the value of any signal in the process's sensitivity list, then the process becomes *active*. Once active, the statements inside the process are evaluated in sequential order. Any assignments made to signals inside the process are not visible outside the process until all of the statements in the process have been evaluated. If there are multiple assignments to the same signal, only the last one has any visible effect. This is illustrated in Example 6.18.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN   STD_LOGIC ;
           f          : OUT  STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.38 A 2-to-1 multiplexer specified using the if-then-else statement.

Example 6.18

The code in Figure 6.39 is equivalent to the code in Figure 6.38. The first statement in the process assigns the value of w_0 to f. This provides a *default* value for f but the assignment does not actually take place until the end of the process. In VHDL jargon we say that the assignment is *scheduled* to occur after all of the statements in the process have been evaluated. If another assignment to f takes place while the process is active, the default assignment will be overridden. The second statement in the process assigns the value of w_1 to f if the value of s is equal to 1. If this condition is true, then the default assignment is overridden. Thus if $s = 0$, then $f = w_0$, and if $s = 1$, then $f = w_1$, which defines the 2-to-1 multiplexer. Compiling this code results in the same circuit as for Figures 6.27, 6.31, and 6.38, namely, $f = \bar{s}w_0 + sw_1$.

The process statement in Figure 6.39 illustrates that the ordering of the statements in a process can affect the meaning of the code. Consider reversing the order of the two statements so that the if-then-else statement is evaluated first. If $s = 1$, f is assigned the value of w_1. This assignment is scheduled and does not take place until the end of the process. However, the statement $f <= w_0$ is evaluated last. It overrides the first assignment, and f is assigned the value of w_0 regardless of the value of s. Hence instead of describing a multiplexer, when the statements inside the process are reversed, the code represents the trivial circuit $f = w_0$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN     STD_LOGIC ;
           f          : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        f <= w0 ;
        IF s = '1' THEN
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.39 Alternative code for the 2-to-1 multiplexer using an if-then-else statement.

Example 6.19 Figure 6.40 gives an example that contains both a concurrent assignment statement and a process statement. It describes a priority encoder and is equivalent to the code in Figure 6.32. The process describes the desired priority scheme using an if-then-else statement. It specifies that if the input w_3 is 1, then the output is set to $y = 11$. 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 clauses in the if-then-else statement are evaluated only if $w_3 = 0$. The first ELSIF clause states that if w_2 is 1, then $y = 10$. If $w_2 = 0$, then the next ELSIF clause results in $y = 01$ if $w_1 = 1$. If $w_3 = w_2 = w_1 = 0$, then the ELSE clause results in $y = 00$. This assignment is done whether or not w_0 is 1; Figure 6.24 indicates that y can be set to any pattern when $w = 0000$ because z will be set to 0 in this case.

The priority encoder's output z must be set to 1 whenever at least one of the data inputs is 1. This output is defined by the conditional assignment statement at the end of Figure 6.40. The VHDL syntax does not allow a conditional assignment statement (or a selected assignment statement) to appear inside a process. An alternative would be to specify the value of z by using an if-then-else statement inside the process. The reason that we have written the code as given in the figure is to illustrate that concurrent assignment statements can be used in conjunction with process statements. The process statement serves the purpose of separating the sequential statements from the concurrent statements. Note that the ordering of the process statement and the conditional assignment statement does not matter. VHDL stipulates that while the statements inside a process are sequential statements, the process statement itself is a concurrent statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        IF w(3) = '1' THEN
            y <= "11" ;
        ELSIF w(2) ='1' THEN
            y <= "10" ;
        ELSIF w(1) ='1' THEN
            y <= "01" ;
        ELSE
            y <= "00" ;
        END IF ;
    END PROCESS ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figure 6.40 A priority encoder specified using the if-then-else statement.

Example 6.20

Figure 6.41 shows an alternative style of code for the priority encoder, using if-then-else statements. The first statement in the process provides the default value of 00 for y_1y_0. The second statement overrides this if w_1 is 1, and sets y_1y_0 to 01. Similarly, the third and fourth statements override the previous ones if w_2 or w_3 are 1, and set y_1y_0 to 10 and 11, respectively. These four statements are equivalent to the single if-then-else statement in Figure 6.40 that describes the priority scheme. The value of z is specified using a default assignment statement, followed by an if-then-else statement that overrides the default if $w = 0000$. Although the examples in Figures 6.40 and 6.41 are equivalent, the meaning of the code in Figure 6.40 is probably easier to understand.

Example 6.21

Figure 6.34 specifies a four-bit comparator that produces the three outputs *AeqB*, *AgtB*, and *AltB*. Figure 6.42 shows how such specification can be written using if-then-else statements. For simplicity, one-bit numbers are used for the inputs *A* and *B*, and only the code for the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        y <= "00" ;
        IF w(1) = '1' THEN y <= "01" ; END IF ;
        IF w(2) = '1' THEN y <= "10" ; END IF ;
        IF w(3) = '1' THEN y <= "11" ; END IF ;

        z <= '1' ;
        IF w = "0000" THEN z <= '0' ; END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.41 Alternative code for the priority encoder.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
    PORT ( A, B  : IN   STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END compare1 ;

ARCHITECTURE Behavior OF compare1 IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        AeqB <= '0' ;
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.42 Code for a one-bit equality comparator.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B  : IN   STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.43 An example of code that results in implied memory.

AeqB output is shown. The process assigns the default value of 0 to *AeqB* and then the if-then-else statement changes *AeqB* to 1 if *A* and *B* are equal. It is instructive to consider the effect on the semantics of the code if the default assignment statement is removed, as illustrated in Figure 6.43.

With only the if-then-else statement, the code does not specify what value *AeqB* should have if the condition $A = B$ is not true. The VHDL semantics stipulate that in cases where the code does not specify the value of a signal, the signal should retain its current value. For the code in Figure 6.43, once *A* and *B* are equal, resulting in $AeqB = 1$, then *AeqB* will remain set to 1 indefinitely, even if *A* and *B* are no longer equal. In the VHDL jargon, the *AeqB* output is said to have *implied memory* because the circuit synthesized from the code will "remember," or store the value $AeqB = 1$. Figure 6.44 shows the circuit synthesized from the code. The XOR gate produces a 1 when *A* and *B* are equal, and the OR gate ensures that *AeqB* remains set to 1 indefinitely.

The implied memory that results from the code in Figure 6.43 is not useful, because it generates a comparator circuit that does not function correctly. However, we will show

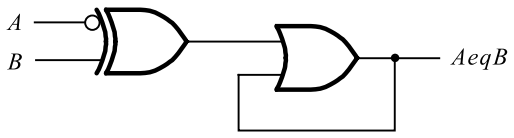


Figure 6.44 The circuit generated from the code in Figure 6.43.

in Chapter 7 that the semantics of implied memory are useful for other types of circuits, which have the capability to store logic signal values in memory elements.

6.6.7 CASE STATEMENT

A case statement is similar to a selected signal assignment in that the case statement has a selection signal and includes WHEN clauses for various valuations of this selection signal. Figure 6.45 shows how the case statement can be used as yet another way of describing the 2-to-1 multiplexer circuit. The case statement begins with the CASE keyword, which specifies that s is to be used as the selection signal. The first WHEN clause specifies, following the => symbol, the statements that should be evaluated when $s = 0$. In this example the only statement evaluated when $s = 0$ is $f <= w_0$. The case statement must include a WHEN clause for all possible valuations of the selection signal. Hence the second WHEN clause, which contains $f <= w_1$, uses the OTHERS keyword.

Example 6.22 Figure 6.30 gives the code for a 2-to-4 decoder. A different way of describing this circuit, using sequential assignment statements, is shown in Figure 6.46. The process first uses an if-then-else statement to check the value of the decoder enable signal En. If $En = 1$, the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN     STD_LOGIC ;
           f          : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        CASE s IS
            WHEN '0' =>
                f <= w0 ;
            WHEN OTHERS =>
                f <= w1 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.45 A case statement that represents a 2-to-1 multiplexer.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN   STD_LOGIC ;
           y   : OUT  STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
    PROCESS ( w, En )
    BEGIN
        IF En = '1' THEN
            CASE w IS
                WHEN "00" =>
                    y <= "1000" ;
                WHEN "01" =>
                    y <= "0100" ;
                WHEN "10" =>
                    y <= "0010" ;
                WHEN OTHERS =>
                    y <= "0001" ;
            END CASE ;
        ELSE
            y <= "0000" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.46 A process statement that describes a 2-to-4 binary decoder.

case statement sets the output y to the appropriate value based on the input w. The case statement represents the first four rows of the truth table in Figure 6.16*a*. If $En = 0$, the ELSE clause sets y to 0000, as specified in the bottom row of the truth table.

Example 6.23

Another example of a case statement is given in Figure 6.47. The entity is named *seg7*, and it represents the BCD-to-7-segment decoder in Figure 6.25. The BCD input is represented by the four-bit signal named *bcd*, and the seven outputs are the seven-bit signal named *leds*. The case statement is formatted so that it resembles the truth table in Figure 6.25*c*. Note that there is a comment to the right of the case statement, which labels the seven outputs

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY seg7 IS
    PORT ( bcd  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           leds : OUT STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;

ARCHITECTURE Behavior OF seg7 IS
BEGIN
    PROCESS ( bcd )
    BEGIN
        CASE bcd IS                    --    abcdefg
            WHEN "0000"  => leds <= "1111110" ;
            WHEN "0001"  => leds <= "0110000" ;
            WHEN "0010"  => leds <= "1101101" ;
            WHEN "0011"  => leds <= "1111001" ;
            WHEN "0100"  => leds <= "0110011" ;
            WHEN "0101"  => leds <= "1011011" ;
            WHEN "0110"  => leds <= "1011111" ;
            WHEN "0111"  => leds <= "1110000" ;
            WHEN "1000"  => leds <= "1111111" ;
            WHEN "1001"  => leds <= "1110011" ;
            WHEN OTHERS => leds <= "-------" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.47 Code that represents a BCD-to-7-segment decoder.

with the letters from *a* to *g*. These labels indicate to the reader the correlation between the seven-bit *leds* signal in the VHDL code and the seven segments in Figure 6.25*b*. The final WHEN clause in the case statement sets all seven bits of *leds* to −. Recall that − is used in VHDL to denote a don't-care condition. This clause 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 6.24 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 standard chip called the 74381. Table 6.1 specifies the functionality of this chip. It has 2 four-bit data inputs, named *A* and *B*; a three-bit select input *s*; and a four-bit output *F*. As the table shows,

Table 6.1 The functionality of the 74381 ALU.

Operation	Inputs $s_2\ s_1\ s_0$	Outputs 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 XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

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 bit-wise fashion. For example, $F = A$ 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 6.48 shows how the functionality of the 74381 ALU can be described using VHDL code. The std_logic_unsigned package, introduced in section 5.5.4, is included so that the STD_LOGIC_VECTOR signals A and B can be used in unsigned arithmetic operations. 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 CPLD. An example of a timing simulation is illustrated in Figure 6.49. For each valuation of s, the circuit generates the appropriate Boolean or arithmetic operation.

6.6.8 VHDL OPERATORS

In this section we discuss the VHDL operators that are useful for synthesizing logic circuits. Table 6.2 lists these operators in groups that reflect the type of operation performed.

To illustrate the results produced by the various operators, we will use three-bit vectors A(2 DOWNTO 0), B(2 DOWNTO 0), and C(2 DOWNTO 0).

Logical Operators

The logical operators can be used with bit and boolean types of operands. The operands can be either single-bit scalars or multibit vectors. For example, the statement

```
C <= NOT A;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY alu IS
    PORT ( s      : IN    STD_LOGIC_VECTOR(2 DOWNTO 0) ;
           A, B   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           F      : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END alu ;

ARCHITECTURE Behavior OF alu IS
BEGIN
    PROCESS ( s, A, B )
    BEGIN
        CASE s IS
            WHEN "000" =>
                F <= "0000" ;
            WHEN "001" =>
                F <= B - A ;
            WHEN "010" =>
                F <= A - B ;
            WHEN "011" =>
                F <= A + B ;
            WHEN "100" =>
                F <= A XOR B ;
            WHEN "101" =>
                F <= A OR B ;
            WHEN "110" =>
                F <= A AND B ;
            WHEN OTHERS =>
                F <= "1111" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.48 Code that represents the functionality of the 74381 ALU chip.

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

The statement

```
C <= A AND B;
```

generates $c_2 = a_2 \cdot b_2$, $c_1 = a_1 \cdot b_1$, and $c_0 = a_0 \cdot b_0$. The other operators lead to similar evaluations.

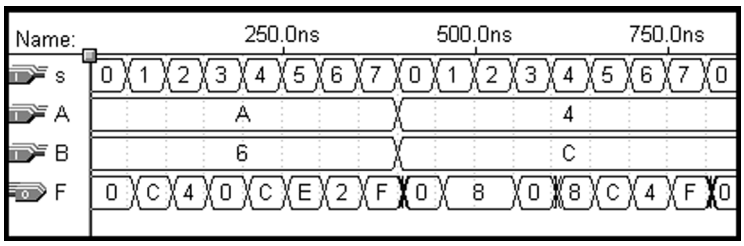


Figure 6.49 Timing simulation for the code in Figure 6.48.

Table 6.2 VHDL operators (used for synthesis).

Operator category	Operator symbol	Operation performed
Logical	AND	AND
	OR	OR
	NAND	Not AND
	NOR	Not OR
	XOR	XOR
	XNOR	Not XOR
	NOT	NOT
Relational	=	Equality
	/=	Inequality
	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Arithmetic	+	Addition
	−	Subtraction
	*	Multiplication
	/	Division
Concatenation	&	Concatenation
Shift and Rotate	SLL	Shift left logical
	SRL	Shift right logical
	SLA	Shift left arithmetic
	SRA	Shift right arithmetic
	ROL	Rotate left
	ROR	Rotate right

Relational Operators

The relational operators are used to compare expressions. The result of the comparison is TRUE or FALSE. The expressions that are compared must be of the same type. For example, if $A = 011$ and $B = 010$ then A > B evaluates to TRUE, and B /= "010" evaluates to FALSE.

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 VHDL compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

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 <= "111" & A & "00";
```

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

Shift and Rotate 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 SLL 1;
```

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

```
B <= A SRL 2;
```

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

The arithmetic shift left, SLA, has the same effect as SLL. But, the arithmetic shift right, SRA, performs the sign extension by replicating the sign bit into the positions left vacant after shifting. Hence

```
B <= A SRA 1;
```

gives $b_2 = a_2$, $b_1 = a_2$, and $b_0 = a_1$.

An operand can also be rotated, in which case the bits shifted out from one end are placed into the vacated positions at the other end. For example,

```
B <= A ROR 2;
```

produces $b_2 = a_1$, $b_1 = a_0$, and $b_0 = a_2$.

Operator Precedence

Operators in different categories have different precedence. Operators in the same category have the same precedence, and are evaluated from left to right in a given expression. It is a good practice to use parentheses to indicate the desired order of operations in the expression. To illustrate this point, consider the statement

```
S <= A + B + C + D;
```

which defines the addition of four vector operands. The VHDL compiler will synthesize a circuit as if the expression was written in the form ((A + B) + C) + D, which gives a cascade of three adders so that the final sum will be available after a propagation delay through three adders. By writing the statement as

```
S <= (A + B) + (C + D);
```

the synthesized circuit will still have three adders, but since the sums $A + B$ and $C + D$ are generated in parallel, the final sum will be available after a propagation delay through only two adders.

Table 6.2 groups the operators informally according to their functionality. It shows only those operators that are used to synthesize logic circuits. The VHDL Standard specifies additional operators, which are useful for simulation and documentation purposes. All operators are grouped into different classes, with a defined precedence ordering between classes. We discuss this issue in Appendix A, section A.3.

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 VHDL 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 a selected signal assignment can also be described using a case statement. Circuits that fit well with conditional signal assignments are also well-suited to if-then-else statements. In general, there are no clear rules that dictate when one type of assignment statement 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.

VHDL is not a programming language, and VHDL code should not be written as if it were a computer program. The concurrent and sequential assignment 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 VHDL statements introduced in this chapter are given in Chapters 7 and 8. In Chapter 10 we provide a number of examples of using VHDL code to describe larger digital systems. For more information on VHDL, the reader can consult more specialized books [5–10].

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 6.25 **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 6.50.

Example 6.26 **Problem:** Derive a circuit that implements an 8-to-3 binary encoder.

Solution: The truth table for the encoder is shown in Figure 6.51. 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

$$y_2 = w_4 + w_5 + w_6 + w_7$$
$$y_1 = w_2 + w_3 + w_6 + w_7$$
$$y_0 = w_1 + w_3 + w_5 + w_7$$

Example 6.27 **Problem:** Implement the function

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

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.

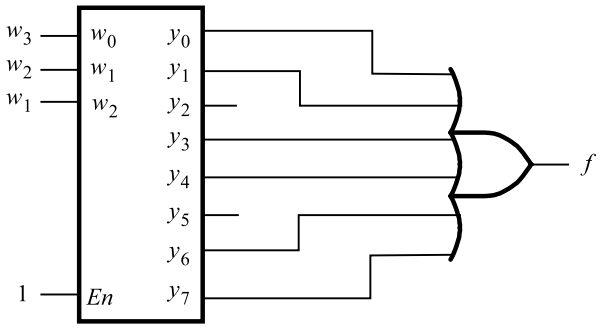


Figure 6.50 Circuit for Example 6.25.

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 6.51 Truth table for an 8-to-3 binary encoder.

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

$$\begin{aligned} f &= \overline{w}_1\overline{w}_4 f_{\overline{w}_1\overline{w}_4} + \overline{w}_1 w_4 f_{\overline{w}_1 w_4} + w_1\overline{w}_4 f_{w_1\overline{w}_4} + w_1 w_4 f_{w_1 w_4} \\ &= \overline{w}_1\overline{w}_4(\overline{w}_2\overline{w}_5) + \overline{w}_1 w_4(w_3 w_5) + w_1\overline{w}_4(w_2 + w_3) + w_1 w_2(1) \end{aligned}$$

We can use a NOR gate to implement $\overline{w}_2\overline{w}_5 = \overline{w_2 + w_5}$. We also need an AND gate and an OR gate. The complete circuit is presented in Figure 6.52.

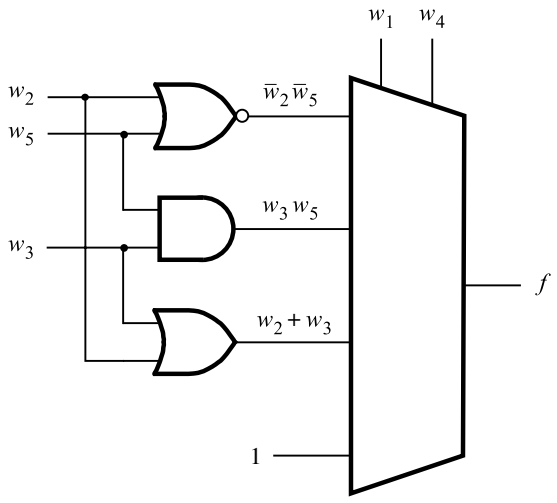


Figure 6.52 Circuit for Example 6.27.

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 6.53 Binary to Gray code coversion.

Example 6.28 **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 6.53 depicts the conversion between three-bit binary and Gray codes. Design a circuit that can convert a binary code into a Gray according the figure.

Solution: From the figure it follows that

$$\begin{aligned} g_2 &= b_2 \\ g_1 &= b_1\overline{b}_2 + \overline{b}_1 b_2 \\ &= b_1 \oplus b_2 \\ g_0 &= b_0\overline{b}_1 + \overline{b}_0 b_1 \\ &= b_0 \oplus b_1 \end{aligned}$$

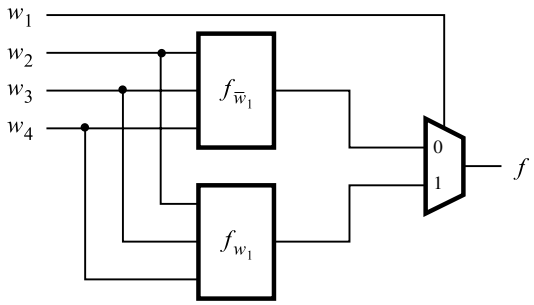
Example 6.29 **Problem:** In section 6.1.2 we showed that any logic function can be decomposed using Shannon's expansion theorem. For a four-variable function, $f(w_1, \ldots, w_4)$, the expansion with respect to w_1 is

$$f(w_1, \ldots, w_4) = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

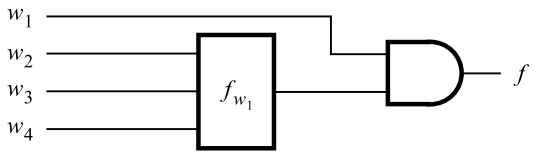
A circuit that implements this expression is given in Figure 6.54*a*.
(a) If the decomposition yields $f_{\overline{w}_1} = 0$, then the multiplexer in the figure can be replaced by a single logic gate. Show this circuit.
(b) Repeat part (*a*) for the case where $f_{w_1} = 1$.

Solution: The desired circuits are shown in parts (*b*) and (*c*) of Figure 6.54.

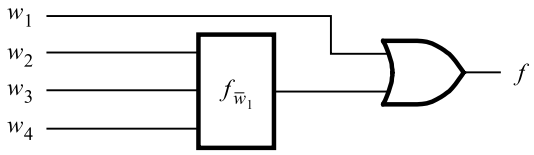
Example 6.30 **Problem:** In several commercial FPGAs the logic blocks are 4-LUTs. What is the minimum number of 4-LUTs needed to construct a 4-to-1 multiplexer with select inputs s_1 and s_0 and data inputs w_3, w_2, w_1, and w_0?



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



(b) Solution for part *a*



(c) Solution for part *b*

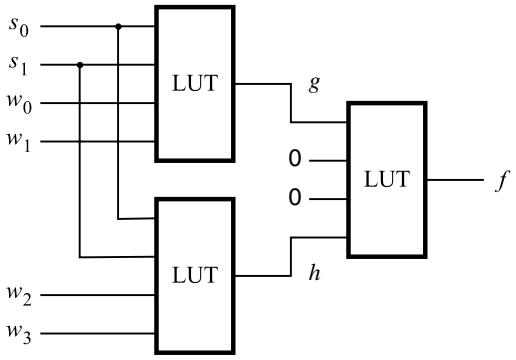
Figure 6.54 Circuits for Example 6.29.

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

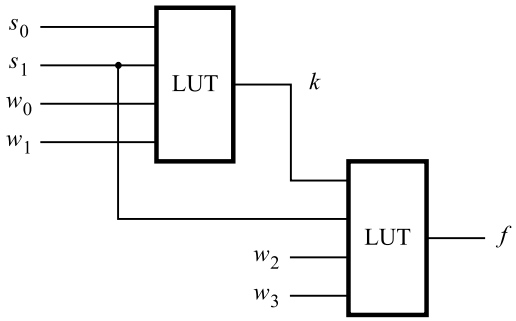
$$f = \overline{s}_1\overline{s}_0 w_0 + \overline{s}_1 s_0 w_1 + s_1\overline{s}_0 w_2 + s_1 s_0 w_3$$

Let $g = \overline{s}_1\overline{s}_0 w_0 + \overline{s}_1 s_0 w_1$ and $h = s_1\overline{s}_0 w_2 + s_1 s_0 w_3$, so that $f = g + h$. This decomposition leads to the circuit in Figure 6.55*a*, which requires three LUTs.

When designing logic circuits, one can sometimes come up with a clever idea which leads to a superior implementation. Figure 6.55*b* shows how it is possible to implement the multiplexer with just two LUTs, based on the following observation. The truth table in Figure 6.2*b* 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



(a) Using three LUTs



(b) Using two LUTs

Figure 6.55 Circuits for Example 6.30.

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 and 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

$$k = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1 + s_1s_0$$

Then, the second LUT performs the function

$$f = \bar{s}_1k + s_1\bar{k}w_3 + s_1kw_4$$

Example 6.31 **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

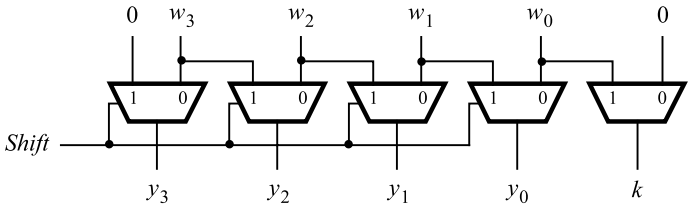


Figure 6.56 A shifter circuit.

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 $k = 0$.

Solution: The required circuit can be implemented with five 2-to-1 multiplexers as shown in Figure 6.56. The *Shift* signal is used as the select input to each multiplexer.

Example 6.32

Problem: The shifter circuit in Example 6.31 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.

Solution: The required action is given in Figure 6.57*a*. The barrel shifter can be implemented with four 4-to-1 multiplexers as shown in Figure 6.57*b*. The control signals s_1 and s_0 are used as the select inputs to the multiplexers.

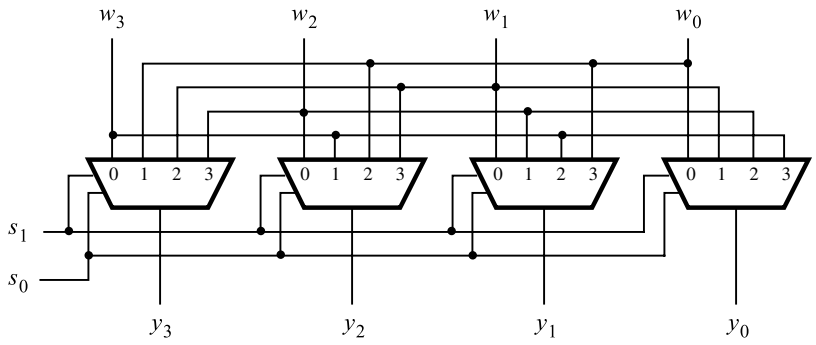
Example 6.33

Problem: Write VHDL code that represents the circuit in Figure 6.19. Use the *dec2to4* entity in Figure 6.30 as a subcircuit in your code.

Solution: The code is shown in Figure 6.58. Note that the *dec2to4* entity 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 6.57 A barrel shifter circuit.

Example 6.34 **Problem:** Write VHDL code that represents the shifter circuit in Figure 6.56.

Solution: There are two possible approaches: structural and behavioral. A structural description is given in Figure 6.59. The IF construct is used to define the desired shifting of individual bits. A typical VHDL compiler will implement this code with 2-to-1 multiplexers as depicted in Figure 6.56.

A behavioral specification is given in Figure 6.60. It makes use of the shift operator SRL. Since the shift and rotate operators are supported in the *ieee.numeric_std.all* library, this library must be included in the code. Note that the vectors *w* and *y* are defined to be of UNSIGNED type.

Example 6.35 **Problem:** Write VHDL code that defines the barrel shifter in Figure 6.57.

Solution: The easiest way to specify the barrel shifter is by using the VHDL rotate operator. The complete code is presented in Figure 6.61.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( s   : IN    STD_LOGIC_VECTOR( 1 DOWNTO 0 ) ;
           w   : IN    STD_LOGIC_VECTOR( 3 DOWNTO 0 ) ;
           f   : OUT   STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Structure OF mux4to1 IS
    COMPONENT dec2to4
        PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En  : IN    STD_LOGIC ;
               y   : OUT   STD_LOGIC_VECTOR(0 TO 3) );
    END COMPONENT;
    SIGNAL High : STD_LOGIC ;
    SIGNAL y : STD_LOGIC_VECTOR( 3 DOWNTO 0 ) ;
BEGIN
    decoder: dec2to4 PORT MAP ( s, '1', y ) ;
    f <= (w(0) AND y(0)) OR (w(1) AND y(1)) OR
         (w(2) AND y(2)) OR w(3) AND y(3) ) ;
END Structure ;


LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <=  "1000" WHEN "100",
              "0100" WHEN "101",
              "0010" WHEN "110",
              "0001" WHEN "111",
              "0000" WHEN OTHERS ;
END Behavior ;
```

Figure 6.58 VHDL code for Example 6.38.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shifter IS
    PORT ( w      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Shift  : IN  STD_LOGIC ;
           y      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           k      : OUT STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = '1' THEN
            y(3) <= '0' ;
            y(2 DOWNTO 0) <= w(3 DOWNTO 1) ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= '0' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.59 Structural VHDL code that specifies the shifter circuit in Figure 6.56.

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 $f = \overline{w}_1\overline{w}_3 + w_2\overline{w}_3 + \overline{w}_1 w_2$. 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 $f = \overline{w}_2\overline{w}_3 + w_1 w_2$.

***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.

6.6 Repeat problem 6.5 for the function $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;

ENTITY shifter IS
    PORT ( w     : IN  UNSIGNED(3 DOWNTO 0) ;
           Shift : IN  STD_LOGIC ;
           y     : OUT UNSIGNED(3 DOWNTO 0) ;
           k     : OUT STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = "1" THEN
            y <= w SRL 1 ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= "0" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure 6.60 Behavioral VHDL code that specifies the shifter circuit in Figure 6.56.

6.7 Consider the function $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1w_3$. Show how repeated application of Shannon's expansion can be used to derive the minterms of f.

6.8 Repeat problem 6.7 for $f = w_2 + \overline{w}_1\overline{w}_3$.

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 $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1w_2w_3$. 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.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;

ENTITY barrel IS
    PORT ( w  : IN   UNSIGNED(3 DOWNTO 0) ;
           s  : IN   UNSIGNED(1 DOWNTO 0) ) ;
           y  : OUT  UNSIGNED(3 DOWNTO 0) ) ;
END barrel ;

ARCHITECTURE Behavior OF barrel IS
BEGIN
    PROCESS (s, w)
    BEGIN
        CASE s IS
            WHEN "00" =>
                y <= w ;
            WHEN "01" =>
                y <= w ROR 1 ;
            WHEN "10" =>
                y <= w ROR 2 ;
            WHEN OTHERS =>
                y <= w ROR 3 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure 6.61 VHDL code that specifies the barrel shifter circuit in Figure 6.57.

***6.14** Give two examples of logic functions with five inputs, $w_1, \ldots, w_5$, that can be realized using 2 four-input LUTs.

6.15 For the function, f, in Example 6.27 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 6.52?

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 $f = w_2\overline{w}_3 + w_1w_3 + \overline{w}_2w_3$ can be implemented using only one Act 1 logic block.

6.17 Show how the function $f = w_1\overline{w}_3 + \overline{w}_1w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$ 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.

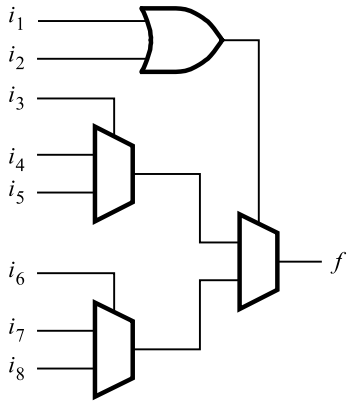


Figure P6.1 The Actel Act 1 logic block.

***6.18** Consider the VHDL 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.

6.19 Write VHDL code that represents the function in problem 6.1, using one selected signal assignment.

6.20 Write VHDL code that represents the function in problem 6.2, using one selected signal assignment.

6.21 Using a selected signal assignment, write VHDL code for a 4-to-2 binary encoder.

6.22 Using a conditional signal assignment, write VHDL code for an 8-to-3 binary encoder.

6.23 Derive the circuit for an 8-to-3 priority encoder.

6.24 Using a conditional signal assignment, write VHDL code for an 8-to-3 priority encoder.

6.25 Repeat problem 6.24, using an if-then-else statement.

6.26 Create a VHDL entity named *if2to4* that represents a 2-to-4 binary decoder using an if-then-else statement. Create a second entity named *h3to8* that represents the 3-to-8 binary decoder in Figure 6.17, using two instances of the *if2to4* entity.

6.27 Create a VHDL entity 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 five instances of the *h3to8* decoder created in problem 6.26.

6.28 Write VHDL code for a BCD-to-7-segment code converter, using a selected signal assignment.

***6.29** Derive minimal sum-of-products expressions for the outputs a, b, and c of the 7-segment display in Figure 6.25.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( w              : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En             : IN   STD_LOGIC ;
           y0, y1, y2, y3 : OUT  STD_LOGIC ) ;
END problem ;

ARCHITECTURE Behavior OF problem IS
BEGIN
    PROCESS (w, En)
    BEGIN
        y0 <= '0' ; y1 <= '0' ; y2 <= '0' ; y3 <= '0' ;
        IF En = '1' THEN
            IF w = "00" THEN y0 <= '1' ;
            ELSIF w = "01" THEN y1 <= '1' ;
            ELSIF w = "10" THEN y2 <= '1' ;
            ELSE y3 <= '1' ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure P6.2 Code for problem 6.18.

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 Design a shifter circuit, similar to the one in Figure 6.56, which can shift a four-bit input vector, $W = w_3w_2w_1w_0$, one bit-position to the right when the control signal *Right* is equal to 1, and one bit-position to the left when the control signal *Left* is equal to 1. When *Right* = *Left* = 0, the output of the circuit should be the same as the input vector. Assume that the condition *Right* = *Left* = 1 will never occur.

6.32 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 P6.3. 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 4×4 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.

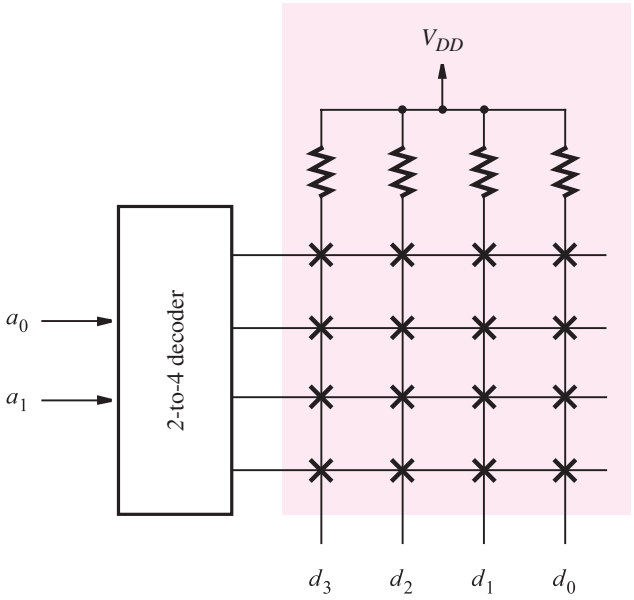


Figure P6.3 A 4 × 4 ROM circuit.

6.33 Show the complete circuit for a ROM using the storage cells designed in Part (*a*) of problem 6.33 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. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2nd ed. (McGraw-Hill: New York, 1998).
6. J. Bhasker, *A VHDL Primer*, 3rd ed. (Prentice-Hall: Englewood Cliffs, NJ, 1998).
7. D. L. Perry, *VHDL*, 3rd ed. (McGraw-Hill: New York, 1998).
8. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
9. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, 1997).
10. D. J. Smith, *HDL Chip Design* (Doone Publications: Madison, AL, 1996).