chapter

4

Combinational-Circuit Building Blocks

Chapter Objectives

In this chapter you will learn about:

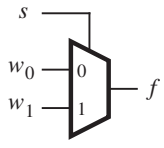
- Commonly used combinational subcircuits
- Multiplexers, which can be used for selection of signals and for implementation of general logic functions
- Circuits used for encoding, decoding, and code-conversion purposes
- Key Verilog constructs used to define combinational circuits

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

4.1 MULTIPLEXERS

Multiplexers were introduced briefly in Chapter 2. 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 4.1 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. Part (*d*) illustrates how it can be constructed with transmission gates which are discussed in Appendix B.

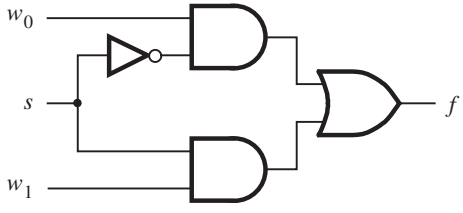
Figure 4.2*a* depicts a larger multiplexer with four data inputs, $w_0, \ldots, w_3$, and two select inputs, s_1 and s_0. As shown in the truth table in part (*b*) of the figure, the two-bit number represented by $s_1 s_0$ selects one of the data inputs as the output of the multiplexer.



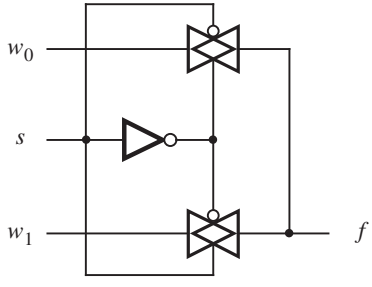
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

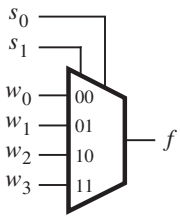


(c) Sum-of-products circuit



(d) Circuit with transmission gates

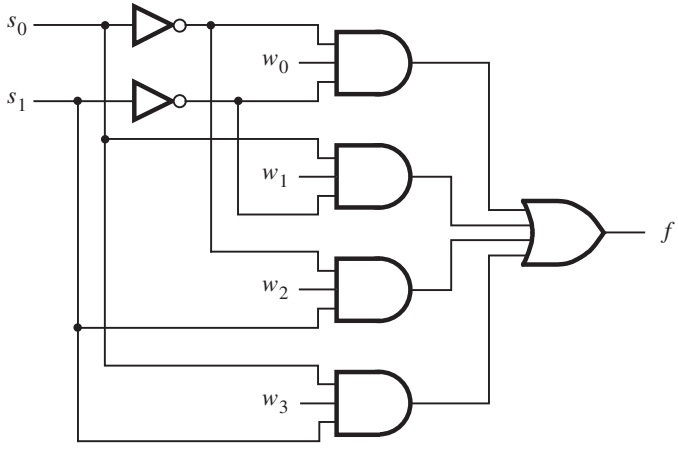
Figure 4.1 A 2-to-1 multiplexer.



(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 4.2 A 4-to-1 multiplexer.

A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 4.2*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 4.3. Figure 4.4 shows how a 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.

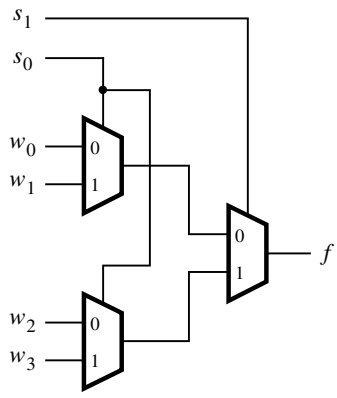


Figure 4.3 Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

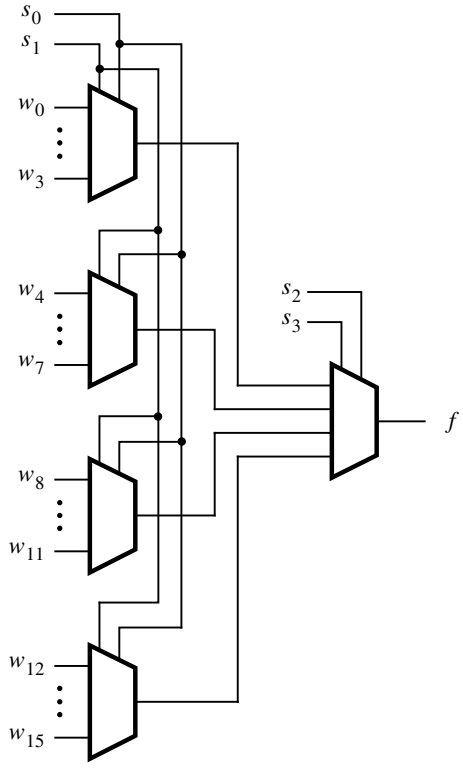
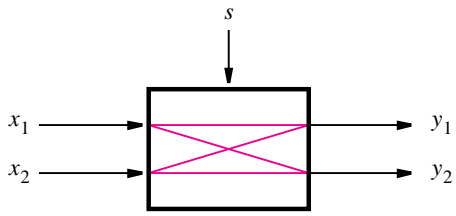
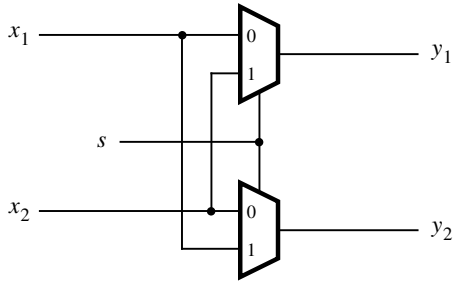


Figure 4.4 A 16-to-1 multiplexer.



(a) A 2x2 crossbar switch



(b) Implementation using multiplexers

Figure 4.5 A practical application of multiplexers.

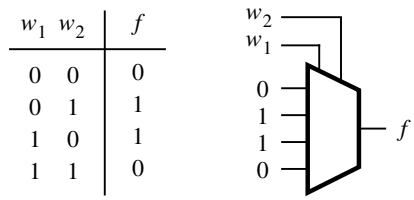
Example 4.1

Figure 4.5 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.

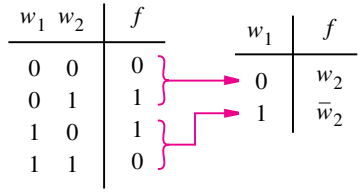
Figure 4.5*b* shows how the 2×2 crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal s. If $s = 0$, the crossbar connects x_1 to y_1 and x_2 to y_2, while if $s = 1$, the crossbar connects x_1 to y_2 and x_2 to y_1. Crossbar switches are useful 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.

4.1.1 SYNTHESIS OF LOGIC FUNCTIONS USING MULTIPLEXERS

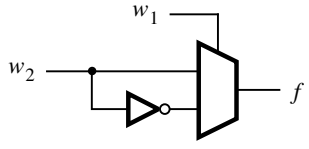
Multiplexers are useful in many practical applications, such as the one described above. They can also be used in a more general way to synthesize logic functions. Consider the



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



(b) Modified truth table

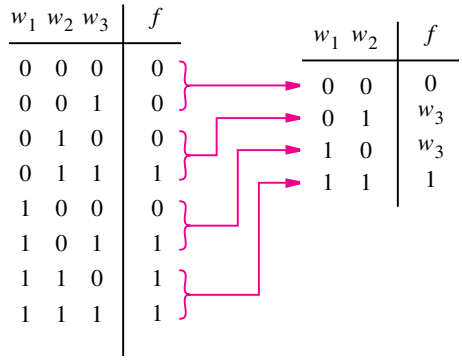


(c) Circuit

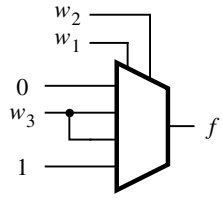
Figure 4.6 Synthesis of a logic function using mutiplexers.

example in Figure 4.6*a*. The truth table defines the function $f = w_1 \oplus w_2$. This function can be implemented by a 4-to-1 multiplexer in which the values of f in each row of the truth table are connected as constants to the multiplexer data inputs. The multiplexer select inputs are driven by w_1 and w_2. Thus for each valuation of $w_1 w_2$, the output f is equal to the function value in the corresponding row of the truth table.

The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 4.6*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 4.6*c*. This procedure can be applied to synthesize a circuit that implements any logic function.



(a) Modified truth table



(b) Circuit

Figure 4.7 Implementation of the three-input majority function using a 4-to-1 multiplexer.

Example 4.2

Figure 4.7*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 4.7*b*.

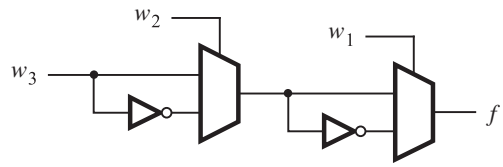
Example 4.3

Figure 4.8*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. Part (*b*) of the figure gives a corresponding circuit. The left multiplexer in the circuit produces $w_2 \oplus w_3$, using the result from Figure 4.6, 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 4.9 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.

w_1	w_2	w_3	f	
0	0	0	0	
0	0	1	1	
0	1	0	1	$w_2 \oplus w_3$
0	1	1	0	
1	0	0	1	
1	0	1	0	
1	1	0	0	$\overline{w_2 \oplus w_3}$
1	1	1	1	

(a) Truth table

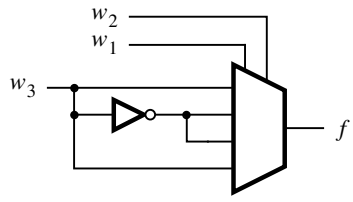


(b) Circuit

Figure 4.8 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 4.9 Three-input XOR implemented with a 4-to-1 multiplexer.

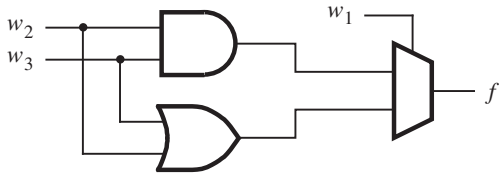
4.1.2 MULTIPLEXER SYNTHESIS USING SHANNON'S EXPANSION

Figures 4.6 through 4.9 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 4.7 using a 2-to-1 multiplexer

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 4.10 The three-input majority function implemented using a 2-to-1 multiplexer.

in this way. Figure 4.10 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 4.10*b*.

This implementation can be derived using algebraic manipulation as follows. The function in Figure 4.10*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 4.10*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].

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 4.9).

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

$$\begin{aligned} f &= \overline{w}_1(0 \cdot w_2 + 0 \cdot w_3 + w_2w_3) + w_1(1 \cdot w_2 + 1 \cdot w_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

which is the expression that we derived above.

For the three-input XOR function, we have

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

which gives the circuit in Figure 4.8*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 $f_{\overline{w}_i}$ denotes $f(w_1, \ldots, w_{i-1}, 0, w_{i+1}, \ldots, w_n)$, 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 4.4.

Example 4.4 For the function $f = \overline{w}_1w_3 + w_2\overline{w}_3$, 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

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2(\overline{w}_1w_3) + w_2(\overline{w}_1w_3 + \overline{w}_3) \\ &= \overline{w}_2(\overline{w}_1w_3) + w_2(\overline{w}_1 + \overline{w}_3) \end{aligned}$$

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, when performing decompositions of this type it is useful to 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.

Example 4.5

Assume that we wish to implement the function

$$f = \overline{w}_1\overline{w}_3 + w_1 w_2 + w_1 w_3$$

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 4.11*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 4.11*b*.

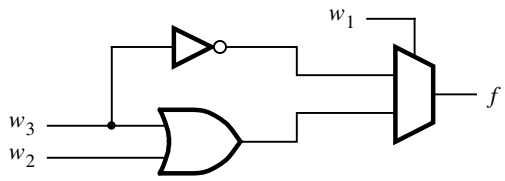
Example 4.6

Consider the three-input majority function

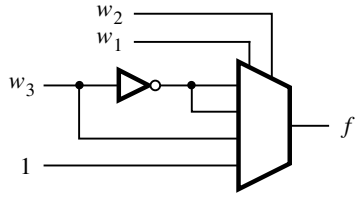
$$f = w_1 w_2 + w_1 w_3 + w_2 w_3$$

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

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



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

Figure 4.11 The circuits synthesized in Example 4.5.

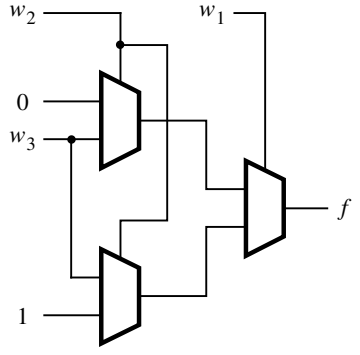


Figure 4.12 The circuit synthesized in Example 4.6.

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

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

The corresponding circuit is shown in Figure 4.12. It is equivalent to the 4-to-1 multiplexer circuit derived using a truth table in Figure 4.7.

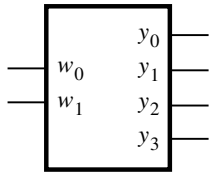
4.2 DECODERS

Consider the logic circuit in Figure 4.13. It has two inputs, w_1 and w_0, and four outputs, y_0, y_1, y_2, and y_3. As shown in the truth table, only one of the outputs is asserted at a time, and each output corresponds to one valuation of the inputs. 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. This type of circuit is called a *binary decoder*. Its inputs represent a binary number, which is decoded to assert the corresponding output. A circuit symbol and logic circuit for this decoder are shown in parts (*b*) and (*c*) of the figure. Each output is driven by an AND gate that decodes the corresponding valuation of w_1w_0.

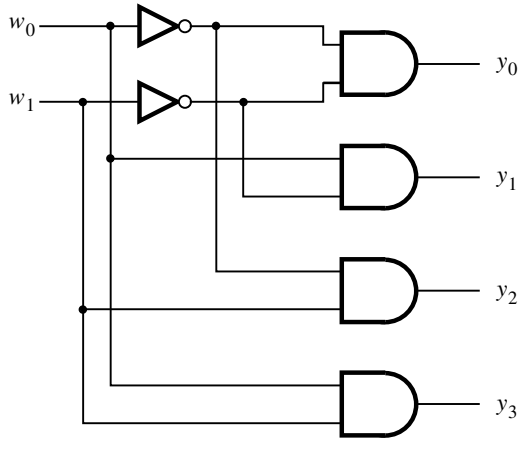
It is useful to include an *enable* input, *En*, in a decoder circuit, as illustrated in Figure 4.14. When enabled by setting $En = 1$ the decoder behaves as presented in Figure 4.13.

w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a) Truth table



(b) Graphical symbol

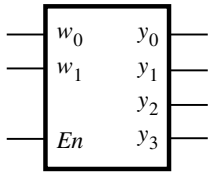


(c) Logic circuit

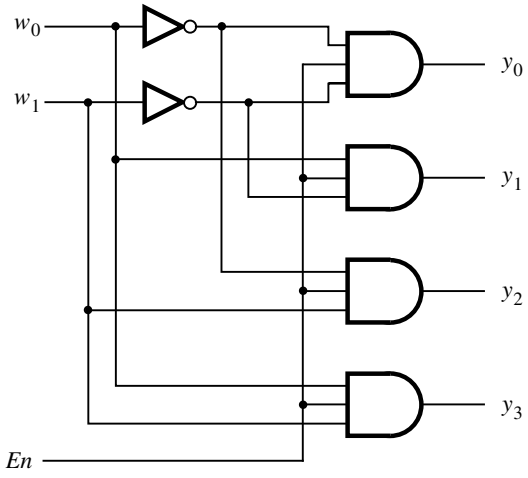
Figure 4.13 A 2-to-4 decoder.

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

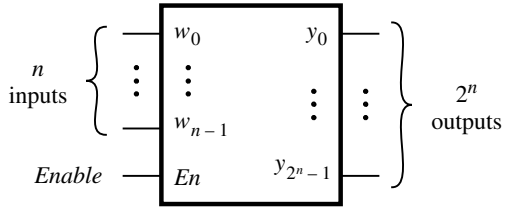
(a) Truth table



(b) Graphical symbol



(c) Logic circuit



(d) An n-to-2^n decoder

Figure 4.14 Binary decoder.

But, if it is disabled by setting $En = 0$, then none of the outputs are asserted. Note that only five rows are shown in the truth table, because if $En = 0$ then all outputs are equal to 0 regardless of the values of w_1 and w_0. The truth table indicates this by showing x when it does not matter whether the variable in question has the value 0 or 1. A graphical symbol for this decoder is given in Figure 4.14*b*. Part (*c*) of the figure shows how the enable capability can be included in the decoder of Figure 4.13*c*. A binary decoder with n inputs has 2^n outputs. A graphical symbol for an n-to-2^n decoder is shown in Figure 4.14*d*.

A k-bit binary code in which exactly one of the bits is set to 1 at a 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 an enabled binary decoder are one-hot encoded.

We should also note that decoders can be designed to have either active-high or active-low outputs. In our discussion, we have assumed that active-high outputs are needed.

Larger decoders can be built using the sum-of-products structure in Figure 4.14*c*, or else they can be constructed from smaller decoders. Figure 4.15 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 4.16 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 4.7

Decoders are useful for many practical purposes. In Figure 4.2*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 4.17. 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.

4.2.1 DEMULTIPLEXERS

We showed in Section 4.1 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 4.14 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 4.14*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.

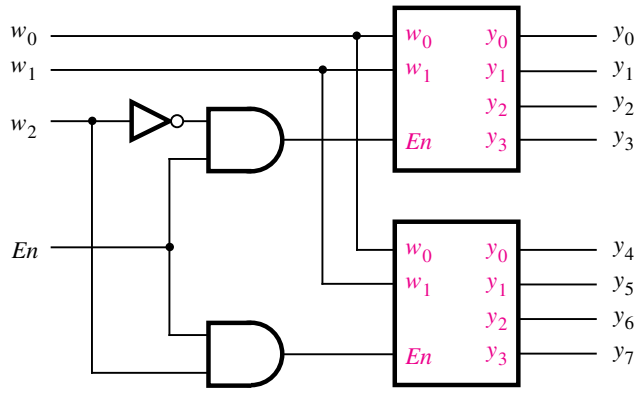


Figure 4.15 A 3-to-8 decoder using two 2-to-4 decoders.

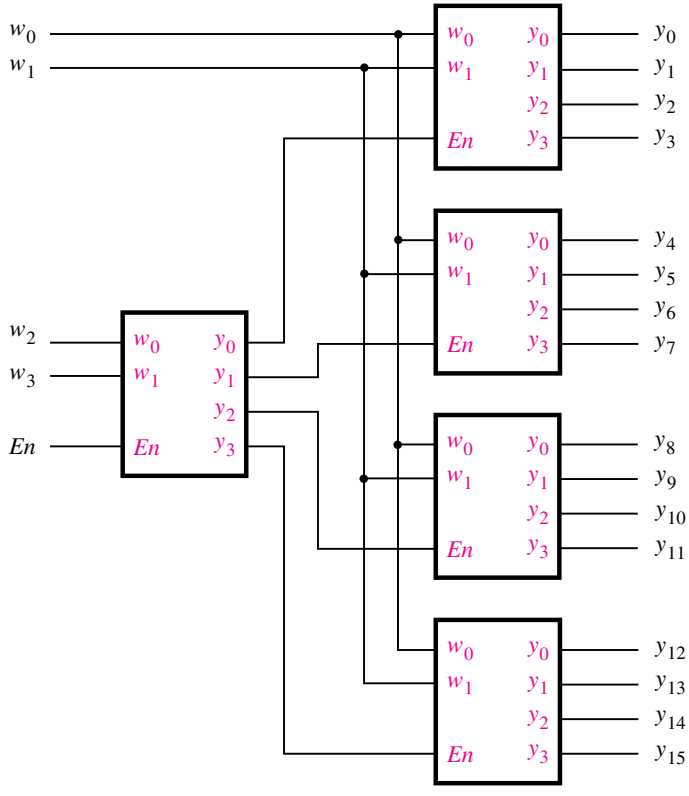


Figure 4.16 A 4-to-16 decoder built using a decoder tree.

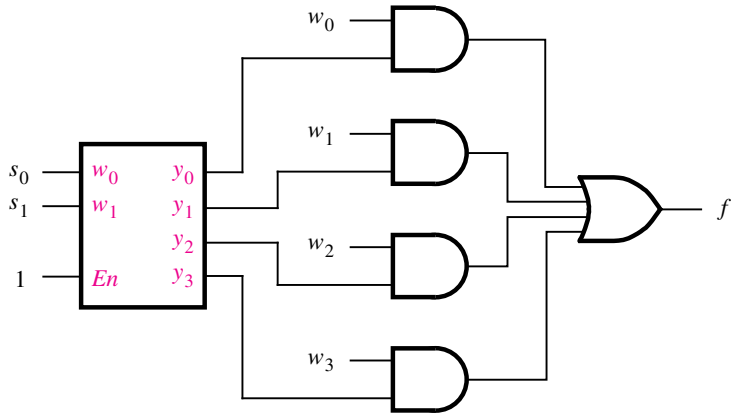


Figure 4.17 A 4-to-1 multiplexer built using a decoder.

4.3 ENCODERS

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

4.3.1 BINARY ENCODERS

A *binary encoder* encodes information from 2^n inputs into an n-bit code, as indicated in Figure 4.18. 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 4.19*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 4.19*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.

4.3.2 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

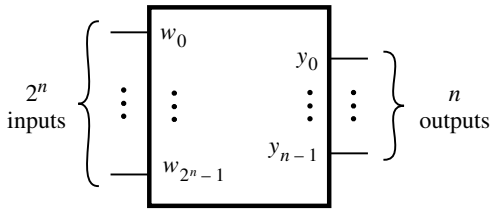
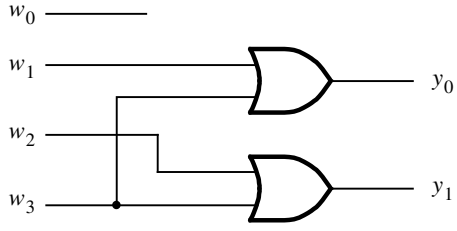


Figure 4.18 A 2^n-to-n binary encoder.

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 4.19 A 4-to-2 binary encoder.

other inputs with lower priority are ignored. The truth table for a 4-to-2 priority encoder is shown in Figure 4.20. 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 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.

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 4.20 Truth table for a 4-to-2 priority encoder.

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 2. 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

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

Using the intermediate signals, the rest of the circuit for the priority encoder has the same structure as the binary encoder in Figure 4.19, namely

$$\begin{aligned} y_0 &= i_1 + i_3 \\ y_1 &= i_2 + i_3 \end{aligned}$$

The output z is given by

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

4.4 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 was introduced in Section 2.14. A similar decoder is often used to display hexadecimal information on seven-segment displays. As explained in Section 3.1.2, long binary numbers are easier to deal with visually if they are represented in the hexadecimal form. A hex-to-7-segment decoder can be implemented as shown in Figure 4.21. Digits 0 to 9 are displayed the same as in the case of the BCD-to-7-segment decoder. Digits 10 to 15 are displayed as A, b, C, d, E, and F.

We should note that although the word *decoder* is traditionally used for such circuits, a more appropriate term is *code converter*. The term *decoder* is more appropriate for circuits that produce one-hot encoded outputs.

4.5 ARITHMETIC COMPARISON CIRCUITS

Chapter 3 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.

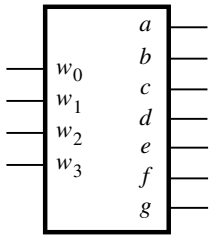
The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of A and B. However, even for moderate values of n, the truth table is large. A better approach is to derive the comparator circuit by considering the bits of A and B in pairs. We can illustrate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

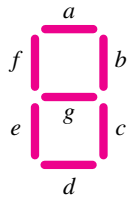
$$AeqB = i_3i_2i_1i_0$$

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

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



(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
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	1	0	1
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	1	1	1

(c) Truth table

Figure 4.21 A hex-to-7-segment display code converter.

The i_k signals ensure that only the first bits, 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 4.22. 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 3.9 in Chapter 3.

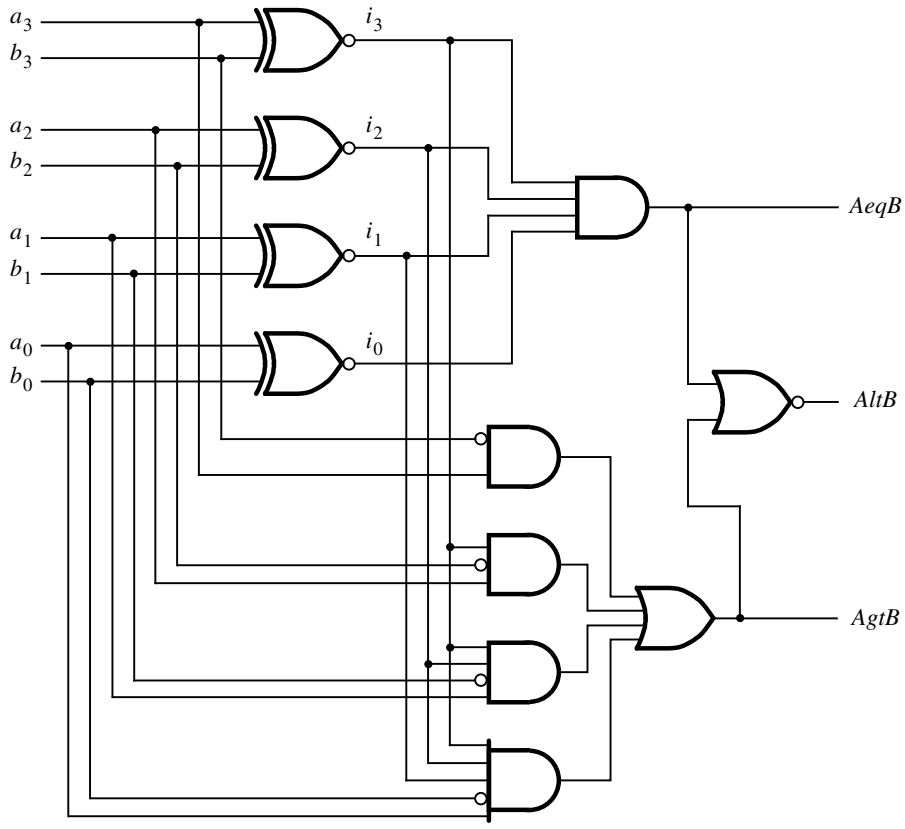


Figure 4.22 A four-bit comparator circuit.

4.6 VERILOG FOR COMBINATIONAL CIRCUITS

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

4.6.1 THE CONDITIONAL OPERATOR

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

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

conditional_expression ? true_expression : false_expression

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

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

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

Example 4.8

A 2-to-1 multiplexer can be defined using the conditional operator in an **assign** statement as shown in Figure 4.23. The module, named *mux2to1*, has the inputs w_0, w_1, and s, and the output f. The signal s is used for the selection criterion. The output f is equal to w_1 if the select input s has the value 1; otherwise, f is equal to w_0. Figure 4.24 shows how the same multiplexer can be defined by using the conditional operator inside an **always** block.

The same approach can be used to define the 4-to-1 multiplexer from Figure 4.2. As seen in the truth table in Figure 4.2*b*, if the select input $s_1 = 1$, then f is set to either w_2 or w_3 based on the value of s_0. Similarly, if $s_1 = 0$, then f is set to either w_0 or w_1. Figure 4.25 shows how nested conditional operators can be used to define this function. The module is called *mux4to1*. Its select inputs are represented by the two-bit vector S. The first conditional expression tests the value of bit s_1. If $s_1 = 1$, then s_0 is tested and f is set to w_3 if $s_0 = 1$ and f is set to w_2 if $s_0 = 0$. This corresponds to the third and fourth rows of the truth table in Figure 4.2*b*. Similarly, if $s_1 = 0$ the conditional operator on the right chooses $f = w_1$ if $s_0 = 1$ and $f = w_0$ if $s_0 = 0$, thus realizing the first two rows of the truth table.

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

   assign f = s ? w1 : w0;

endmodule
```

Figure 4.23 A 2-to-1 multiplexer specified using the conditional operator.

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

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

endmodule
```

Figure 4.24 An alternative specification of a 2-to-1 multiplexer using the conditional operator.

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

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

endmodule
```

Figure 4.25 A 4-to-1 multiplexer specified using the conditional operator.

4.6.2 THE IF-ELSE STATEMENT

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

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

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

Example 4.9 Figure 4.26 shows how the **if-else** statement can be used to describe a 2-to-1 multiplexer. The **if** clause states that f is assigned the value of w_0 when $s = 0$. Else, f is assigned the value of w_1.

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

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

endmodule
```

Figure 4.26 Code for a 2-to-1 multiplexer using the **if-else** statement.

The **if-else** statement can be used to implement larger multiplexers. A 4-to-1 multiplexer is shown in Figure 4.27. The **if-else** clauses set f to the value of one of the inputs $w_0, \ldots, w_3$, depending on the valuation of S.

Another way of defining the same circuit is presented in Figure 4.28. In this case, a four-bit vector W is defined instead of single-bit signals w_0, w_1, w_2, and w_3. Also, the four different values of S are specified as decimal rather than binary numbers.

Example 4.10

Figure 4.4 shows how a 16-to-1 multiplexer can be built by using five 4-to-1 multiplexers. Figure 4.29 presents Verilog code for this circuit using five instantiations of the *mux4to1* module. The data inputs to the *mux16to1* module are the 16-bit vector W, and the select inputs are the four-bit vector S. In the Verilog code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left of Figure 4.4. A four-bit signal named M is used for this purpose. The first multiplexer instantiated, *Mux1*, corresponds to the multiplexer at the top left of Figure 4.4. Its first four ports are driven by the signals $W[0], \ldots, W[3]$. The syntax $S[1:0]$ is used to attach the signals $S[1]$ and $S[0]$ to the two-bit S port of the *mux4to1* module. The $M[0]$ signal is connected to the multiplexer's output port. Similarly, *Mux2, Mux3*, and *Mux4* are instantiations of the next three multiplexers on the left. The multiplexer on the right of Figure 4.4 is instantiated as *Mux5*. The signals $M[0], \ldots, M[3]$ are connected to its data inputs, and bits $S[3]$ and $S[2]$ are attached to the select inputs. The output port generates the *mux16to1* output f. Compiling the code results in the multiplexer function

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

Since the *mux4to1* module is being instantiated in the code of Figure 4.29, it is necessary to either include the code of Figure 4.28 in the same file as the *mux16to1* module or place the *mux4to1* module in a separate file in the same directory, or a directory with a

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

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

endmodule
```

Figure 4.27 Code for a 4-to-1 multiplexer using the **if-else** statement.

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

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

endmodule
```

Figure 4.28 Alternative specification of a 4-to-1 multiplexer.

specified path so that the Verilog compiler can find it. Observe that if the code in Figure 4.27 were used as the required *mux4to1* module, then we would have to list the ports separately, as in $W[0]$, $W[1]$, $W[2]$, $W[3]$, rather than as the vector $W[0:3]$.

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

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

endmodule
```

Figure 4.29 Hierarchical code for a 16-to-1 multiplexer.

4.6.3 THE CASE STATEMENT

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

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

The value of the controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included. Otherwise, the Verilog compiler will synthesize memory elements to deal with the unspecified possibilities; we will discuss this issue in Chapter 5.

Example 4.11 The **case** statement can be used to define a 4-to-1 multiplexer as shown in Figure 4.30. The four values that the select vector *S* can have are given as decimal numbers, but they could also be given as binary numbers.

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

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

endmodule
```

Figure 4.30 A 4-to-1 multiplexer defined using the **case** statement.

Example 4.12 Figure 4.31 shows how a **case** statement can be used to describe the truth table for a 2-to-4 binary decoder. The module is called *dec2to4*. The data inputs are the two-bit vector W, and the enable input is En. The four outputs are represented by the four-bit vector Y.

In the truth table for the decoder in Figure 4.14*a*, the inputs are listed in the order $En\ w_1\ w_0$. To represent these three signals in the controlling expression, the Verilog code uses the concatenate operator to combine the En and W signals into a three-bit vector. The four alternatives in the **case** statement correspond to the truth table in Figure 4.14*a* where $En = 1$, and the decoder outputs have the same patterns as in the first four rows of the truth table. The last clause uses the **default** keyword and sets the decoder outputs to 0000, because it represents all other cases, namely those where $En = 0$.

Example 4.13 The 2-to-4 decoder can be specified using a combination of **if-else** and **case** statements as given in Figure 4.32. If $En = 0$, then all four bits of the output Y are set to the value 0, else the **case** alternatives are evaluated if $En = 1$.

Example 4.14 The tree structure of the 4-to-16 decoder in Figure 4.16 can be defined as shown in Figure 4.33. The inputs are a four-bit vector W and an enable signal En. The outputs are represented by the 16-bit vector Y. The circuit uses five instances of the 2-to-4 decoder defined in either Figure 4.31 or 4.32. The outputs of the left-most decoder in Figure 4.16 are denoted as the four-bit vector M in Figure 4.33.

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

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

endmodule
```

Figure 4.31 Verilog code for a 2-to-4 binary decoder.

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

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

endmodule
```

Figure 4.32 Alternative code for a 2-to-4 binary decoder.

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

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

endmodule
```

Figure 4.33 Verilog code for a 4-to-16 decoder.

Example 4.15 Another example of a **case** statement is given in Figure 4.34. The module, *seg7*, represents the hex-to-7-segment decoder in Figure 4.21. The hexadecimal input is the four-bit vector named *hex*, and the seven outputs are the seven-bit vector named *leds*. The **case** alternatives are listed so that they resemble the truth table in Figure 4.21*c*. Note that there is a comment to the right of the **case** statement, which labels the seven outputs with the letters from *a* to *g*. These labels indicate to the reader the correlation between the bits of the *leds* vector in the Verilog code and the seven segments in Figure 4.21*b*.

Example 4.16 An arithmetic logic unit (ALU) is a logic circuit that performs various Boolean and arithmetic operations on n-bit operands. Table 4.1 specifies the functionality of a simple ALU, known as the 74381 chip, which has been available in the form of a standard chip in the family called the 7400-series. This ALU has 2 four-bit data inputs, A and B, a three-bit select input, S, and a four-bit output, F. As the table shows, F is defined by various arithmetic or Boolean operations on the inputs A and B. In this table $+$ means arithmetic addition, and $-$ means arithmetic subtraction. To avoid confusion, the table uses the words XOR, OR, and AND for the Boolean operations. Each Boolean operation is done in a bitwise fashion. For example, $F = A$ 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 4.35 shows how the functionality of the 74381 ALU can be described in Verilog code. The **case** statement shown corresponds directly to Table 4.1.

The Casex and Casez Statements

Logic circuits that we have considered so far operate using the logic values 0 and 1. When specifying the functionality of such circuits in the form of a truth table, we sometimes encounter cases where it does not matter whether a given logic variable has the value 0 or 1, as seen in Figures 4.14*a* and 4.20. It is customary to use the letter x to denote such cases, where x represents an unknown value.

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

    always @(hex)
        case (hex)    //abcdefg
            0: leds = 7'b1111110;
            1: leds = 7'b0110000;
            2: leds = 7'b1101101;
            3: leds = 7'b1111001;
            4: leds = 7'b0110011;
            5: leds = 7'b1011011;
            6: leds = 7'b1011111;
            7: leds = 7'b1110000;
            8: leds = 7'b1111111;
            9: leds = 7'b1111011;
           10: leds = 7'b1110111;
           11: leds = 7'b0011111;
           12: leds = 7'b1001110;
           13: leds = 7'b0111101;
           14: leds = 7'b1001111;
           15: leds = 7'b1000111;
        endcase

endmodule
```

Figure 4.34 Code for a hex-to-7-segment decoder.

Table 4.1 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

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

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

endmodule
```

Figure 4.35 Code that represents the functionality of the 74381 ALU chip.

It is also possible to implement circuits that can produce three different types of output signals. In addition to the usual 0 and 1 values, there is a third value that indicates that the output line is not connected to any defined voltage level. In this state the output behaves like an open circuit, as explained in Appendix B. We say that the output is in the *high-impedance* state, which is usually denoted by using the letter z.

In Verilog, a signal can have four possible values: 0, 1, z, or x. The z and x values can also be denoted by the capital letters Z and X. In the **case** statement it is possible to use the logic values 0, 1, z, and x in the **case** alternatives. A bit-by-bit comparison is used to determine the match between the expression and one of the alternatives.

Verilog provides two variants of the **case** statement that treat the z and x values in a different way. The **casez** statement treats all z values in the case alternatives and the controlling expression as don't cares. The **casex** statement treats all z and x values as don't cares.

Example 4.17 Figure 4.36 gives Verilog code for the priority encoder defined in Figure 4.20. The desired priority scheme is realized by using a **casex** statement. The first alternative specifies that the output is set to $y_1y_0 = 3$ if the input w_3 is 1. This assignment does not depend on the values of inputs w_2, w_1, or w_0; hence their values do not matter. The other alternatives in the **casex** statement are evaluated only if $w_3 = 0$. The second alternative states that if w_2 is 1, then $y_1y_0 = 2$. If $w_2 = 0$, then the next alternative results in $y_1y_0 = 1$ if $w_1 = 1$. If $w_3 = w_2 = w_1 = 0$ and $w_0 = 1$, then the fourth alternative results in $y_1y_0 = 0$.

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

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

endmodule
```

Figure 4.36 Verilog code for a priority encoder.

The priority encoder's output z must be set to 1 whenever at least one of the data inputs is 1. This output is set to 1 outside the **casex** statement in the **always** block. If none of the four alternatives matches the value of W, then the **default** clause overrides the value of z and sets it to 0. The **default** clause also indicates that the Y output can be set to any pattern because it will be ignored.

4.6.4 THE FOR LOOP

If the structure of a desired circuit exhibits a certain regularity, it may be convenient to define the circuit using a **for** loop. We introduced the **for** loop in Section 3.5.4, where it was useful in a generic specification of a ripple-carry adder. The **for** loop has the syntax

for (initial_index; terminal_index; increment) statement;

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

Unlike **for** loops in high-level programming languages, the Verilog **for** loop does not specify changes that take place in time through successive loop iterations. Instead, during each iteration it specifies a different subcircuit. In Figure 3.25 the **for** loop was used to define a cascade of full-adder subcircuits to form an n-bit ripple-carry adder. The **for** loop can be used to define many other structures as illustrated by the next two examples.

Example 4.18 Figure 4.37 shows how the **for** loop can be used to specify a 2-to-4 decoder circuit. The effect of the loop is to repeat the **if-else** statement four times, for $k = 0, \ldots, 3$. The first loop iteration sets $y_0 = 1$ if $W = 0$ and $En = 1$. Similarly, the other three iterations set the values of y_1, y_2, and y_3 according to the values of W and En.

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

Example 4.19 The priority encoder of Figure 4.20 can be defined by the Verilog code in Figure 4.38. In the **always** block, the output bits y_1 and y_0 are first set to the don't-care state and z is cleared to 0. Then, if one or more of the four inputs $w_3, \ldots, w_0$ is equal to 1, the **for** loop will set the valuation of y_1y_0 to match the index of the highest priority input that has the value 1. Note that each successive iteration through the loop corresponds to a higher priority. Verilog semantics specify that a signal that receives multiple assignments in an **always** block retains the last assignment. Thus the iteration that corresponds to the highest priority input that is equal to 1 will override any setting of Y established during the previous iterations.

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

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

endmodule
```

Figure 4.37 A 2-to-4 binary decoder specified using the **for** loop.

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

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

endmodule
```

Figure 4.38 A priority encoder specified using the **for** loop.

4.6.5 VERILOG OPERATORS

In this section we discuss the Verilog operators that are useful for synthesizing logic circuits. Table 4.2 lists these operators in groups that reflect the type of operation performed. A more complete listing of the operators is given in Table A.1.

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

Bitwise Operators

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

```
C = ~A;
```

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.

Most bitwise operators operate on pairs of bits. The statement

```
C = A & B;
```

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

Table 4.2 Verilog operators.

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

$$\mathrm{C = A \sim\!\wedge\ B;}$$

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

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

$$\mathrm{f = A \wedge B;}$$

yields $f = a_0 \oplus b_0$.

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

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

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

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

Figure 4.39 Truth tables for bitwise operators.

The bitwise operations may involve operands that include the unknown logic value *x*. Then the operations are performed according to the truth tables in Figure 4.39. For example, if P = 4'b101x and Q = 4'b1001, then P & Q = 4'b100x while P | Q = 4'b1011.

Logical Operators

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

```
f = !A;
```

then f will be equal to 1 (true) only if all bits of A are equal to 0 (false). Hence, $f = \overline{a_2 + a_1 + a_0}$.

The && operator implements the AND operation such that

```
f = A && B;
```

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

```
f = A || B;
```

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

Reduction Operators

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

```
f = &A;
```

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

```
f = ^A;
```

gives $f = a_2 \oplus a_1 \oplus a_0$, and so on.

Arithmetic Operators

We have already encountered the arithmetic operators in Chapter 3. They perform standard arithmetic operations. Thus

$$\mathrm{C} = \mathrm{A} + \mathrm{B};$$

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

$$\mathrm{C} = \mathrm{A} - \mathrm{B};$$

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

$$\mathrm{C} = -\mathrm{A};$$

places the 2's complement of A into C.

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

Relational Operators

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

Example 4.20 The use of relational operators in the **if-else** statement is illustrated in Figure 4.40. The defined circuit is the four-bit comparator described in Section 4.5.

Equality Operators

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

Shift Operators

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

$$\mathrm{B} = \mathrm{A} << 1;$$

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

$$\mathrm{B} = \mathrm{A} >> 2;$$

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

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

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

endmodule
```

Figure 4.40 Verilog code for a four-bit comparator.

Concatenate Operator

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

```
D = {A, B};
```

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

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

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

Replication Operator

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

The replication operator may be used in conjunction with the concatenate operator. For instance, {2{A}, 3{B}} is equivalent to {A, A, B, B, B}. We introduced the concatenate and replication operators in Sections 3.5.6 and 3.5.8, respectively, and illustrated their use in specifying the adder circuits.

Table 4.3 Precedence of Verilog operators.

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

Conditional Operator

The conditional operator is discussed fully in Section 4.6.1.

Operator Precedence

The Verilog operators are assumed to have the precedence indicated in Table 4.3. The order of precedence is from top to bottom; operators in the top row have the highest precedence and those in the bottom row have the lowest precedence. The operators listed in the same row have the same precedence.

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

4.6.6 THE GENERATE CONSTRUCT

In Section 3.5.4 we introduced the **generate** loop capability which can be used to create multiple instances of subcircuits. A subcircuit may be defined in a block of statements delineated by the **generate** and **endgenerate** keywords. The subcircuit is instantiated multiple times using a generate-index variable. This variable is defined using the **genvar** keyword and it can have only positive integer values. It is not possible to use an index declared as a normal **integer** variable.

Example 4.21 Figure 4.41 shows how the **generate** construct can be used to specify an n-bit ripple-carry adder. The subcircuit is a full-adder defined structurally in terms of primitive gates as introduced in Figure 3.18. The **for** loop causes the full-adder block to be instantiated n times.

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

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

endmodule
```

Figure 4.41 Using the **generate** loop to define an n-bit ripple-carry adder.

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

4.6.7 TASKS AND FUNCTIONS

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

Verilog Task

A task is declared by the keyword **task** and it comprises a block of statements that ends with the keyword **endtask**. The task must be included in the module that calls it. It may

have input and output ports. These are not the ports of the module that contains the task, which are used to make external connections to the module. The task ports are used only to pass values between the module and the task.

Example 4.22 In Figure 4.29 we showed the Verilog code for a 16-to-1 multiplexer that instantiates five copies of a 4-to-1 multiplexer circuit given in a separate module named *mux4to1*. The same circuit can be specified using the task approach as shown in Figure 4.42. Observe the key differences. The task *mux4to1* is included in the module *mux16to1*. It is called from an **always** block by means of an appropriate **case** statement. The output of a task must be a variable, hence *g* is of **reg** type.

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

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

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

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

endmodule
```

Figure 4.42 Use of a task in Verilog code.

Verilog Function

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

Example 4.23

Figure 4.43 shows how the code in Figure 4.42 can be written to use a function. The Verilog compiler essentially inserts the body of the function at each place where it is called. Hence the clause

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

becomes

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

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

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

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

endmodule
```

Figure 4.43 The code from Figure 4.42 using a function.

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

The function serves as a convenience that makes the *mux16to1* module more compact.

A Verilog function can invoke another function but it cannot call a Verilog task. A task may call another task and it may invoke a function. In Figure 4.42 we defined the task after the **always** block that calls it. In contrast, in Figure 4.43 we defined the function before the **always** block that invokes it. Both possibilities are allowed in the Verilog standard for both tasks and functions. However, some tools require functions to be defined before the statements that invoke them.

4.7 CONCLUDING REMARKS

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

Verilog is not a programming language, and Verilog code should not be written as if it were a computer program. The statements discussed in this chapter can be used to create large, complex circuits. A good way to design such circuits is to construct them using well-defined modules, in the manner that we illustrated for the multiplexers, decoders, encoders, and so on. Additional examples using the Verilog statements introduced in this chapter are given in Chapters 5 and 6. In Chapter 7 we provide a number of examples of using Verilog code to describe larger digital systems. For more information on Verilog, the reader can consult more specialized books [2–8].

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

4.8 EXAMPLES OF SOLVED PROBLEMS

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

Example 4.24

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 4.44.

Example 4.25

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

Solution: The truth table for the encoder is shown in Figure 4.45. 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 4.26

Problem: Implement the function

$$f(w_1, w_2, w_3, w_4, w_5) = \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

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, w_4, and w_5 are available.

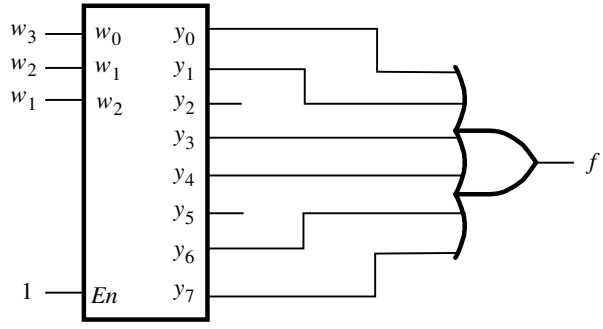


Figure 4.44 Circuit for Example 4.24.

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

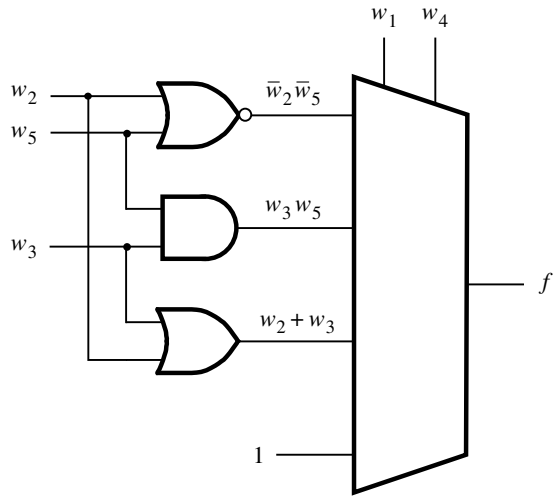


Figure 4.46 Circuit for Example 4.26.

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

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

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 4.46.

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 4.47 Binary to Gray code coversion.

Example 4.27

Problem: In Chapter 2 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 4.47 depicts the conversion between three-bit binary and Gray codes. Design a circuit that can convert a binary code into Gray code according to the figure.

Solution: From the figure it follows that

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

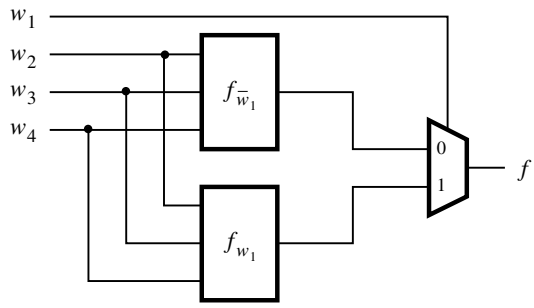
Example 4.28

Problem: In Section 4.1.2 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

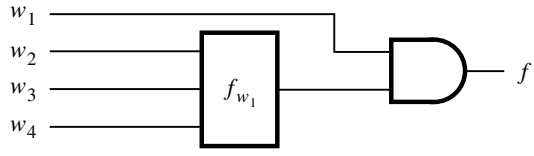
$$f(w_1, \ldots, w_4) = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

A circuit that implements this expression is given in Figure 4.48*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 $f_{w_1} = 1$.

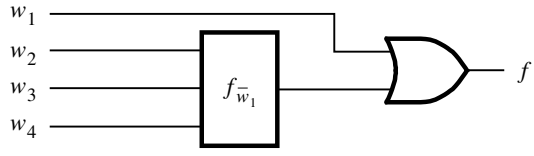
Solution: The desired circuits are shown in parts (*b*) and (*c*) of Figure 4.48.



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



(b) Solution for part *a*.



(c) Solution for part *b*.

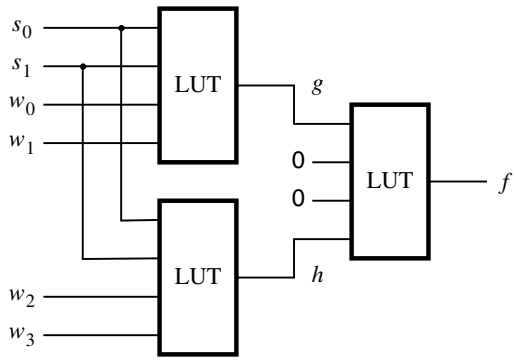
Figure 4.48 Circuits for Example 4.28.

Example 4.29 **Problem:** In Section 2.17 we said that field-programmable gate arrays (FPGAs) contain lookup tables (LUTs) that are used to implement logic functions. Each LUT can be programmed to implement any logic function of its inputs. FPGAs are discussed in detail in Appendix B. Many commercial FPGAs contain four-input lookup tables (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?

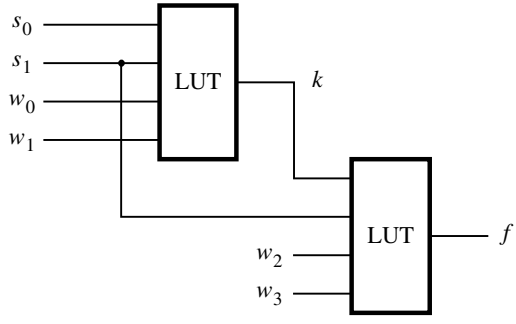
Solution: A straightforward attempt is to use directly the expression that defines the 4-to-1 multiplexer

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

Let $g = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1$ and $h = s_1\bar{s}_0w_2 + s_1s_0w_3$, so that $f = g + h$. This decomposition leads to the circuit in Figure 4.49*a*, which requires three LUTs.



(a) Using three LUTs



(b) Using two LUTs

Figure 4.49 Circuits for Example 4.29.

When designing logic circuits, one can sometimes come up with a clever idea which leads to a superior implementation. Figure 4.49*b* shows how it is possible to implement the multiplexer with just two LUTs, based on the following observation. The truth table in Figure 4.2*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, with the output k. The second LUT must make the choice between w_2 and w_3 when $s_1 = 1$. But, the choice can be made only by knowing the value of s_0. Since it is impossible to have five inputs in the LUT, more information has to be passed from the first to the second LUT. Observe that when $s_1 = 1$ the output f will be equal to either w_2 or w_3, in which case it is not necessary to know the values of w_0 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

$$k = \overline{s}_1(\overline{s}_0 w_0 + s_0 w_1) + s_1 s_0$$

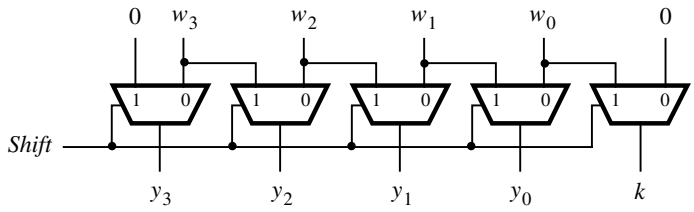


Figure 4.50 A shifter circuit.

Then, the second LUT performs the function

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

Example 4.30 **Problem:** In digital systems it is often necessary to have circuits that can shift the bits of a vector by one or more bit positions to the left or right. Design a circuit that can shift a four-bit vector $W = w_3 w_2 w_1 w_0$ 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 $Y = y_3 y_2 y_1 y_0$ 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 4.50. The *Shift* signal is used as the select input to each multiplexer.

Example 4.31 **Problem:** The shifter circuit in Example 4.30 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 4.51*a*. The barrel shifter can be implemented with four 4-to-1 multiplexers as shown in Figure 4.51*b*. The control signals s_1 and s_0 are used as the select inputs to the multiplexers.

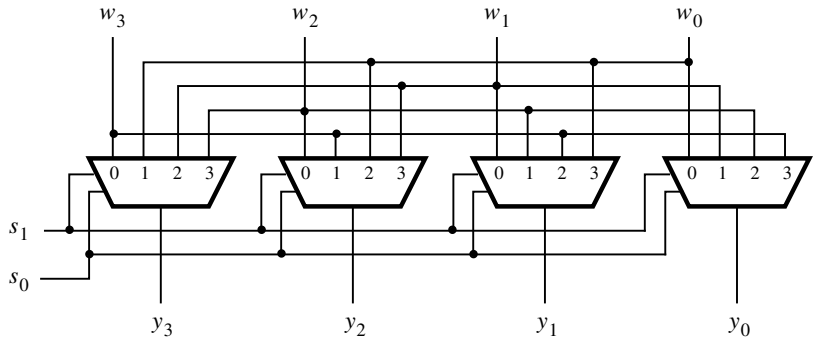
Example 4.32 **Problem:** Write Verilog code that represents the circuit in Figure 4.17. Use the *dec2to4* module in Figure 4.31 as a subcircuit in your code.

Solution: The code is shown in Figure 4.52. Note that the *dec2to4* module can be included in the same file as we have done in the figure, but it can also be in a separate file in the project directory.

---

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

(a) Truth table



(b) Circuit

Figure 4.51 A barrel shifter circuit.

Example 4.33

Problem: Write Verilog code that represents the shifter circuit in Figure 4.50.

Solution: One possibility is to specify the structure of this circuit as shown in Figure 4.53. The **if-else** construct is used to define the desired shifting of individual bits. A typical Verilog compiler will implement this code with 2-to-1 multiplexers as depicted in Figure 4.50.

An alternative is to make use of the shift operator defined in Section 4.6.5, as indicated in Figure 4.54.

Example 4.34

Problem: Write Verilog code that defines the barrel shifter in Figure 4.51.

Solution: The code in Figure 4.55 is a possible solution. The rotate function is accomplished by concatenating two copies of the input vector W and shifting the obtained 8-bit vector to the right by the number of bit positions specified as the input S. The four least-significant bits of the resulting 8-bit vector are the desired output Y.

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

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

endmodule

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

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

endmodule
```

Figure 4.52 Verilog code for Example 4.32.

Example 4.35 The concept of *parity* is widely used in digital systems for error-checking purposes. When digital information is transmitted from one point to another, perhaps by long wires, it is possible for some bits to become corrupted during the transmission process. For example, the sender may transmit a bit whose value is equal to 1, but the receiver observes a bit whose value is 0. Suppose that a data item consists of n bits. A simple error-checking mechanism can be implemented by including an extra bit, p, which indicates the parity of the n-bit item. Two kinds of parity can be used. For *even parity* the p bit is given the value such that the total number of 1s in the $n + 1$ transmitted bits (comprising the n-bit data and the parity bit p) is even. For *odd parity* the p bit is given the value that makes the total number of 1s odd. The sender generates the p bit based on the n-bit data item that is to be transmitted. The receiver checks whether the parity of the received item is correct.

Parity generating and checking circuits can be realized with XOR gates. For example, for a four-bit data item consisting of bits $x_3x_2x_1x_0$, the even parity bit can be generated as

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

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

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

endmodule
```

Figure 4.53 Verilog code for the circuit in Figure 4.50.

At the receiving end the checking is done using

$$c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

If $c = 0$, then the received item shows the correct parity. If $c = 1$, then an error has occurred. Note that observing $c = 0$ is not a guarantee that the received item is correct. If two or any even number of bits have their values inverted during the transmission, the parity of the data item will not be changed; hence the error will not be detected. But if an odd number of bits are corrupted, then the error will be detected.

Problem: The ASCII code, discussed in Section 1.5.3, uses seven-bit patterns to represent characters. In computer applications it is common to use one byte per character. The eighth bit, b_7, is usually set to 0 for use in digital processing. But, if the the character data is to be transmitted from one digital system to another, it may be prudent to use bit b_7 as a parity bit. Write Verilog code that specifies a circuit that accepts an input byte (where $b_7 = 0$) and produces an output byte where b_7 is the even parity bit.

Solution: Let X and Y be the input and output bytes, respectively. Then, the desired solution is given in Figure 4.56.

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

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

endmodule
```

Figure 4.54 Alternative Verilog code for the circuit in Figure 4.50.

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

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

endmodule
```

Figure 4.55 Verilog code for the barrel shifter.

```
module parity (X, Y);
    input  [7:0] X;
    output [7:0] Y;

    assign Y = {^X[6:0], X[6:0]};

endmodule
```

Figure 4.56 Verilog code for Example 4.35.

PROBLEMS

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

4.1 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.

4.2 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.

***4.3** 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.

4.4 Repeat Problem 4.3 for the function $f = \overline{w}_2\overline{w}_3 + w_1 w_2$.

***4.5** 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.

4.6 Repeat Problem 4.5 for the function $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$.

4.7 Consider the function $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1 w_3$. Show how repeated application of Shannon's expansion can be used to derive the minterms of f.

4.8 Repeat Problem 4.7 for $f = w_2 + \overline{w}_1\overline{w}_3$.

4.9 Prove Shannon's expansion theorem presented in Section 4.1.2.

***4.10** Section 4.1.2 shows Shannon's expansion in sum-of-products form. Using the principle of duality, derive the equivalent expression in product-of-sums form.

***4.11** Consider the function $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1 w_2 w_3$. The cost of this 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.

4.12 Use multiplexers to implement the circuit for stage 0 of the carry-lookahead adder in Figure 3.15 (included in the right-most shaded area).

***4.13** Derive minimal sum-of-products expressions for the outputs a, b, and c of the 7-segment display in Figure 4.21.

4.14 Derive minimal sum-of-products expressions for the outputs d, e, f, and g of the 7-segment display in Figure 4.21.

4.15 For the function, f, in Example 4.26 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 4.46?

4.16 Consider the multiplexer-based circuit illustrated in Figure P4.1. Show how the function $f = w_2\overline{w}_3 + w_1 w_3 + \overline{w}_2 w_3$ can be implemented using only one instance of this circuit.

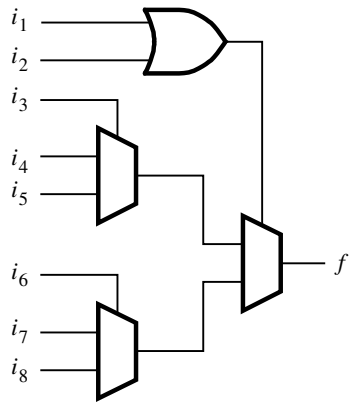


Figure P4.1 A multiplexer-based circuit.

4.17 Show how the function $f = w_1\overline{w}_3 + \overline{w}_1 w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$ can be realized using one or more instances of the circuit in Figure P4.1. Note that there are no NOT gates in the circuit; hence complements of signals have to be generated using the multiplexers in the logic block.

***4.18** Consider the Verilog code in Figure P4.2. What type of circuit does the code represent? Comment on whether or not the style of code used is a good choice for the circuit that it represents.

```
module problem4_18 (W, En, y0, y1, y2, y3);
   input [1:0] W;
   input En;
   output reg y0, y1, y2, y3;

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

endmodule
```

Figure P4.2 Code for Problem 4.18.

4.19 Write Verilog code that represents the function in Problem 4.2, using a **case** statement.

4.20 Write Verilog code for a 4-to-2 binary encoder.

4.21 Write Verilog code for an 8-to-3 binary encoder.

4.22 Figure P4.3 shows a modified version of the code for a 2-to-4 decoder in Figure 4.37. This code is almost correct but contains one error. What is the error?

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

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

endmodule
```

Figure P4.3 Code for Problem 4.22.

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

4.24 Using a **casex** statement, write Verilog code for an 8-to-3 priority encoder.

4.25 Repeat Problem 4.24, using a **for** loop.

4.26 Create a Verilog module named *if2to4* that represents a 2-to-4 binary decoder using an **if-else** statement. Create a second module named *h3to8* that represents the 3-to-8 binary decoder in Figure 4.15 using two instances of the *if2to4* module.

4.27 Create a Verilog module named *h6to64* that represents a 6-to-64 binary decoder. Use the treelike structure in Figure 4.16, in which the 6-to-64 decoder is built using nine instances of the *h3to8* decoder created in Problem 4.26.

4.28 Write Verilog code that represents the circuit in Figure 4.17. Use the *dec2to4* module in Figure 4.31 as a subcircuit in your code.

4.29 Design a shifter circuit, similar to the one in Figure 4.50, 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.

4.30 Design a circuit that can multiply an eight-bit number, $A = a_7, \ldots, a_0$, by 1, 2, 3 or 4 to produce the result A, $2A$, $3A$ or $4A$, respectively.

4.31 Write Verilog code that implements the task in Problem 4.30.

4.32 Figure 4.47 depicts the relationship between the binary and Gray codes. Design a circuit that can convert Gray code into binary code.

4.33 Example 4.35 and Figure 4.56 show how a circuit that generates an ASCII byte suitable for sending over a communications link may be defined. Write Verilog code for its counterpart at the receiving end, where byte Y (which includes the parity bit) has to be converted into byte X in which the bit x_7 has to be 0. An *error* signal has to be produced, which is set to 0 or 1 depending on whether the parity check indicates correct or erroneous transmission, respectively.

REFERENCES

1. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits," *Transactions AIEE* 57 (1938), pp. 713–723.
2. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
3. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
4. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).
5. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
6. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
7. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
8. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).