COMBINATIONAL BASICS 2

In this chapter, we look at combinational circuits in some detail. We start with some of the theory underpinning combinational circuits, and show how circuits of gates correspond to formulas in the theory. Next, we show how information can be represented in binary form for processing by digital circuits. We then survey a range of components that can be used as building blocks in larger combinational circuits. Finally, we return to our design methodology and discuss verification of combinational circuits.

2.1 BOOLEAN FUNCTIONS AND BOOLEAN ALGEBRA

In Chapter 1, we showed how a digital signal can be used to represent information with two possible values, such as the truth or falsehood of a logical condition. We will now expand on that discussion and show how the laws of logic can be used to analyze and design digital systems that use binary representation. The theoretical foundation that we will use is called *Boolean algebra*, named after the nineteenth century British mathematician, George Boole, who invented the mathematical theory that deals with logical propositions.

2.1.1 BOOLEAN FUNCTIONS

According to our abstract view, a digital logic circuit has inputs and outputs, each of which has a low or high voltage at any given time. We think of these two voltage levels as electrical implementations of two *Boolean values*, 0 and 1, respectively. We could choose other names for the Boolean values, such as F and T, corresponding to falsehood and truth of logical conditions. However, that would make them harder to distinguish from the names of variables that we also introduce. Use of 0 and 1 is equally valid, less confusing, and closer to the way we express Boolean values in hardware description languages.

The combinational circuits that we mentioned in Chapter 1 have outputs that depend only on the current input values. In such circuits, each output value is a *Boolean function* of one or more inputs. This means that, for each possible combination of Boolean input values, the output takes on a specified Boolean value. This is analogous to functions on other sets of values, such as addition on numbers, where for each possible combination of operand numbers, a function yields a result number.

The most direct way of defining a Boolean function is simply to list the result values for each combination of input values. We call a table containing such a list a *truth table*. Table 2.1 shows truth tables for three basic Boolean functions that we will denote with the symbols "+", "·" and the overbar notation ("¯"). The "+" function is the *logical OR* of its two operands, and the "·" function is the *logical AND* of its operands. We use these operator symbols because the functions have many properties in common with arithmetic addition and multiplication. However, there are some differences, as we will see. The function denoted by the overbar notation is the logical negation (logical "NOT") of its single operand.

Another way of defining a Boolean function is to use a *Boolean expression*, in which we combine the literal values 0 and 1 and Boolean variables with Boolean operators. We will use alphanumeric names such as x, y and z for variables. Each variable represents a Boolean value, such as the value of a signal in a digital circuit or the value of a logical condition. Note that the column headings in Table 2.1 are simple Boolean expressions. More generally, we can include an arbitrary number of literals, variables and operators, and can use parentheses to specify an order of evaluation. We adopt the convention of giving "·" higher precedence than "+", allowing us to omit parentheses in expressions such as $(a \cdot b) + c$, giving the equivalent expression $a \cdot b + c$.

In practical terms, the literal values 0 and 1 are usually implemented as low-voltage and high-voltage digital signal values, respectively. The operator "+" is implemented as an *OR gate*, "·" as an *AND gate*, and "¯" as an *inverter*. (We introduced these basic gates in Chapter 1.) Named variables in Boolean expressions are implemented by digital signals of the same name. A complete Boolean expression is implemented by a circuit of interconnected gates, in which there is one gate corresponding to each operator in the expression. We can also write a *Boolean equation* in which one

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	$\bar{x}$
0	0
1	0

TABLE 2.1 Truth tables for the logical OR, AND and negation functions.

Boolean expression is defined to be equal to another. A Boolean equation in which a single variable of a given name is defined to be equal to a Boolean expression is implemented by the circuit for the expression yielding an output with the given name. For example, the Boolean equation

$$f = (x + y) \cdot \overline{z}$$

is implemented by the digital logic circuit shown in Figure 2.1.

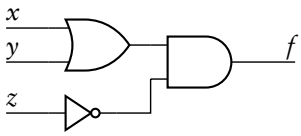

FIGURE 2.1 Circuit implementing a Boolean equation.

We can show that truth tables and Boolean expressions are equally valid ways of specifying Boolean functions. For any Boolean expression, we can write a truth table with a column for each variable mentioned in the expression and a column for the expression value. We systematically fill in a row for each combination of variable values. For an expression with n distinct variables, there are 2^n combinations, so we need 2^n rows. For each combination, we substitute the variable values into the expression and evaluate the result. We write the result in the same row as the variable values, under the expression column.

EXAMPLE 2.1 Derive the truth table corresponding to the Boolean expression $(x + y) \cdot \overline{z}$.

SOLUTION There are three distinct variables in the expression, namely, x, y and z, so we will need $2^3 = 8$ rows in our truth table, as shown in Table 2.2. The easiest way to systematically fill in the variable values is to start with the value 0 for x in the first half of the table and 1 in the second half. Then, in each half, fill in the value 0 for y in the first half of that half and 1 in the second half of that half. In general, keep on filling in columns to the right, reducing the number of successive 0s and 1s by half each time, until single 0s and 1s alternate in the column for the last variable. Now evaluate the expression for the first row, substituting 0 values for x, y and z, to get the result 0. For the second row, substitute 0 for x and y and 1 for z, also giving the result 0. Continue in this way until all rows are filled in.

---

x	y	z	$(x + y) \cdot \overline{z}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

TABLE 2.2 Truth table for a Boolean expression.

We can also work in the reverse direction and derive a Boolean expression for a function represented by a truth table. We do this by examining the rows for which the expression has the value 1. For each such row, we form the logical AND of those variables for which the input value is 1, together with the negation of those variables for which the input value is 0. Such a conjunction is called a *minterm* of the function. For example, the third row of Table 2.2 gives us the minterm $\overline{x} \cdot y \cdot \overline{z}$. The complete expression for the function is then the logical OR of all the minterms for which the function value is 1. Thus, for the function of Table 2.2, the expression is

$$\overline{x} \cdot y \cdot \overline{z} + x \cdot \overline{y} \cdot \overline{z} + x \cdot y \cdot \overline{z}$$

Note that this is not the same expression as $(x + y) \cdot \bar{z}$, but it does have the same value for all combinations of input values. We say that the two expressions are *equivalent*, denoting the same function, and write the Boolean equation

$$(x + y) \cdot \bar{z} = \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot \bar{z}$$

The right-hand expression in this equation is in *sum-of-products* form, meaning that it is the "sum" (logical OR) of a number of "product"(logical AND) terms, or *p-terms*, of variables. Note that each term in a sum-of-products expression need not be a minterm; that is, it need not include every variable that is mentioned in the expression. For example, another sum-of-products expression that is equivalent to the above expression is

$$\bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{z}$$

An implication of equivalence of Boolean expressions is that digital circuits corresponding to equivalent expressions also implement the same function. For example, the two circuits shown in Figure 2.2, corresponding to the equivalent expressions $(x + y) \cdot \bar{z}$ and $\bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{z}$, are functionally equivalent. This is a very important idea, as it means we can choose among the various equivalent circuits to implement a given function in order to satisfy nonfunctional constraints. Making such choices is a form of *optimization*, and is central to digital logic design. Note that a circuit with the minimal number of logic gates may not be the best choice in all circumstances. It depends on the particular constraints that apply. For example, if we are constrained to implement the function in certain kinds of programmable logic device, the circuit on the left may actually have more delay than the circuit on the right. We will return to the idea of constraint-dependent optimization many times throughout this book. In particular, in Section 2.1.2, we will look at some ways in which we can determine equivalent circuits for a given Boolean function.

An interesting thing about the logical OR, AND and negation operators is that any Boolean function can be written as an expression involving just these operators. One way to see the truth of this statement is to recognize that any function can be written as a truth table, and from there as a sum of products of minterms. Such an expression only involves the basic operators. A corollary is that any Boolean function can

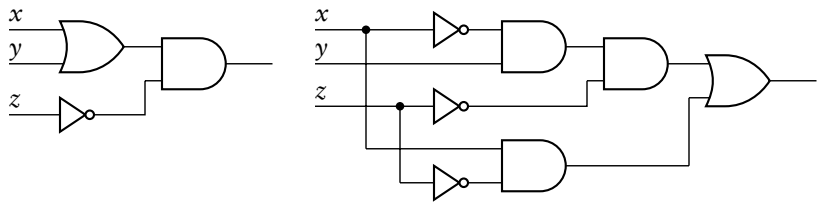

FIGURE 2.2 Two equivalent digital circuits.

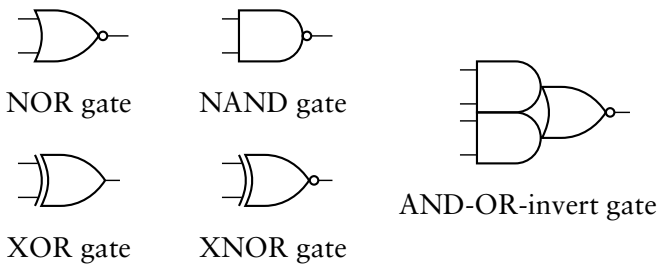

FIGURE 2.3 Complex logic gates.

be implemented using only OR gates, AND gates and inverters. However, such an implementation may not be optimal or even meet constraints. In fact, in most implementation fabrics, these gates are not the most simple that we can use. Figure 2.3 shows a number of other gates. They are often called *complex gates*, as their functions are combinations of the basic logical operations. The NOR, NAND and AND-OR-invert gates are of particular interest, since their internal circuitry in many implementation fabrics is very simple, and hence fast. Use of those gates can often lead to smaller and faster circuits for a given Boolean function than circuits involving OR and AND gates.

The function implemented by the *NOR* gate is the negation of the OR operation. Similarly, the function implemented by the *NAND* gate is the negation of the AND operation. The term *XOR* is short for *exclusive OR*, denoted by the operator "⊕" in Boolean expressions. The result of the exclusive OR operator is 1 if either, but not both, of the inputs is 1; and is 0 if both inputs are 0 or both inputs are 1. This is closer to what we usually mean when we say "or" informally in English. For example, when we're asked if we'd like ice cream or cake for dessert, we usually don't expect both! The function implemented by the *XNOR* gate is the negation of the exclusive OR operation. It is 1 when both inputs are the same and 0 when the inputs differ. For this reason, it is also called an *equivalence* gate. Finally, the *AND-OR-invert* gate performs the logical AND on each of two pairs of inputs, then performs a NOR operation on the two results. While it may look overly complicated to be called a single gate, the electrical implementation as a transistor circuit is surprisingly simple, which is why we include it here. The truth table for the functions implemented by the two-input gates are shown in Table 2.3. The truth table for the AND-OR-invert gate is left as an exercise.

a	b	$\overline{a+b}$	$\overline{a \cdot b}$	$a \oplus b$	$\overline{a \oplus b}$
0	0	1	1	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	0	0	0	1

TABLE 2.3 Truth table for functions implemented by complex gates.

EXAMPLE 2.2 Use truth tables to show that the following two Boolean functions are equivalent. Design a circuit using NOR and NAND gates for the first function, and a circuit using OR and AND gates and inverters for the second.

$$f_1 = \overline{\overline{a \cdot b} + c} \quad \text{and} \quad f_2 = (a \cdot b) \cdot \bar{c}$$

SOLUTION The truth table for f_1 is shown in Table 2.4, and that for f_2 is shown in Table 2.5. For each combination of input values, both functions have the same result value, so they are equivalent.

a	b	c	$\overline{a \cdot b}$	$\overline{a \cdot b} + c$	f_1
0	0	0	1	1	0
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	1	1	0
1	0	0	1	1	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	0	1	0

TABLE 2.4 Truth table for the first function.

a	b	c	$a \cdot b$	$\bar{c}$	f_2
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	0	0	0
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	0

TABLE 2.5 Truth table for the second function.

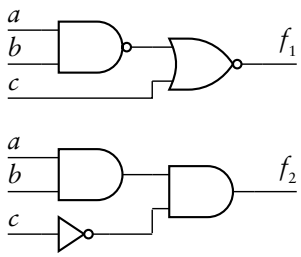

FIGURE 2.4 Two equivalent gate circuits.

The function f_1 involves the NAND operation applied to a and b, followed by the NOR operation applied to the result and c. The circuit implementing this function is shown at the top of Figure 2.4. The function f_2 involves the AND operation applied to a and b, followed by the AND operation applied to the result and the negation of c. The circuit for this function is shown at the bottom of Figure 2.4. Note that, since NAND and NOR gates are considerably simpler

and faster in most implementation fabrics, the circuit at the top would be the preferred implementation.

There is one further Boolean function that we need to consider, namely, the *identity* function. This function has one input, and the function's value is just the value of the input. The simplest implementation of the identity function is a piece of wire. However, there is also a gate component, called a *buffer*, that implements the identity function. The symbol for a buffer is shown in Figure 2.5.

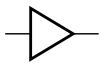

FIGURE 2.5 Symbol for a buffer.

It might seem strange to waste precious circuit area and power on a component that doesn't do anything. However, if we recall our discussion in Chapter 1 of static and capacitive loading of component outputs, we realize that buffer components are useful when we need to connect a given output to many inputs. If we just connect the output directly to the inputs, the output may be overloaded, affecting its ability to drive proper logic levels or to change between logic levels with acceptable rise and fall times. By inserting buffers between the output and the inputs, as shown in Figure 2.6, we can reduce the loading on the output to just that of the buffer inputs. Furthermore, each buffer output is now driving a fraction of the original inputs. When the number of inputs to be driven is very large, we can buffer the outputs of the buffers, and so on, forming a *buffer tree*, as shown in Figure 2.7. This is a two-level buffer tree, meaning that the original output drives each of the original inputs through two intervening buffers. If we extrapolate this arrangement, we can see that the number of inputs that can be driven from an output increases exponentially with the number of levels in the buffer tree.

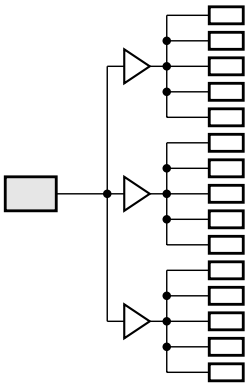

FIGURE 2.6 Using buffers to reduce loading on a component.

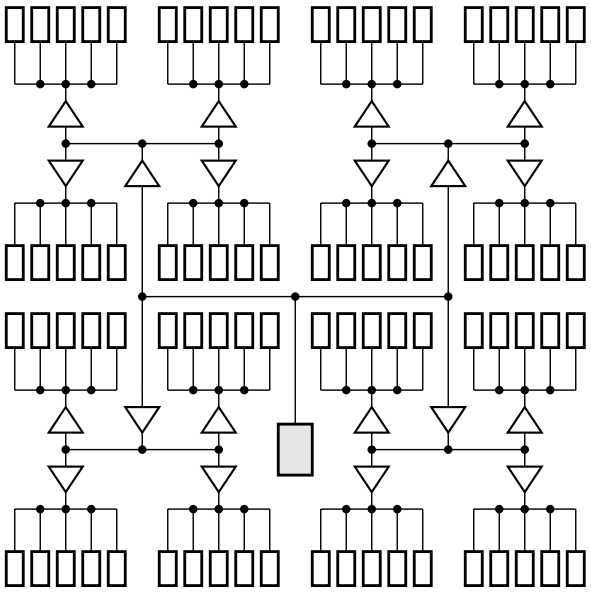

FIGURE 2.7 A two-level buffer tree.

As we shall see later, one important use for buffer trees is for connecting a clock signal from a clock-generator circuit to all of the flip-flops in a system. Meanwhile, however, we just need to be aware that buffers and buffer trees can be used in combinational circuits where many inputs are to be driven from a single output.

Don't Care Notation

While truth tables provide a systematic way to completely define a Boolean function, they can be cumbersome, particularly when the function has a lot of inputs. In many such cases, we can write the truth table in more compact form using the *don't care* notation for function inputs. In this book, we use the notation "−" for don't care, but "X" is another commonly used notation. Use of the don't care notation takes advantage of the property of many Boolean functions that, if some inputs have given values, the values of other inputs don't affect the result value. This is illustrated in Table 2.6, which shows the complete truth table and the compacted truth table for the function

$$z = \bar{s} \cdot a + s \cdot b$$

This is a Boolean equation for the multiplexer component that we introduced in Chapter 1. The input s represents the select input, and a and b represent the two data inputs: a is selected when $s = 0$ and b is selected when $s = 1$.

Note that, for this function, when $s = 0$, we don't care what value b has, and the output is the same as a. Similarly, when $s = 1$, we don't care what value a has, and the output is the same as b. This is shown in the compacted form using the dash symbol to denote an input whose value we don't care about. This simple expedient reduces the table to half the size, while still specifying the same information about the function.

s	a	b	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

s	a	b	z
0	0	−	0
0	1	−	1
1	−	0	0
1	−	1	1

TABLE 2.6 Complete and compacted truth tables for the multiplexer function.

In some designs, we can also use the don't care notation for the result of a function. We can do this if the design only requires a *partial function*, that is, if the function result need only be specified for some combinations of inputs and not for others. Usually, the input combinations for which we don't care about the result are those combinations that cannot arise during operation of the circuit; the combinations are logically impossible, given the functionality of the system of which the circuit is a part. However, any real circuit that we design will yield some value, either 0 or 1, for all possible input combinations. The benefit of specifying "don't care" for the impossible combinations, rather than arbitrarily choosing 0 or 1 as the function result, is that it gives us more scope for optimizing the circuit. We might be able to identify two candidate circuits that both produce the required outputs for the combinations we do care about, but that differ in their output for the "don't care" combinations. If one of the candidates better meets constraints than the other, we would choose it, accepting whatever result it yields for the "don't care" combinations.

EXAMPLE 2.3 The truth table in Table 2.7 has two don't care entries for the function f, since a result of 0 or 1 is equally acceptable for those two "impossible" input combinations. Compare the circuits that result from choosing 0 or 1 as the actual function result for both of the don't care combinations.

a	b	c	f	f_1	f_2
0	0	0	–	0	1
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	0	0	0
1	0	0	–	0	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	0	0	0

TABLE 2.7 Truth table for a function with "don't care" results, and two realizations of the function.

SOLUTION If a value of 0 is chosen for both of the input combinations, the resulting function can be expressed as the sum of two minterms $f_1 = \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot c$, and can be implemented by the circuit shown at the top of Figure 2.8. If a value of 1 is chosen for the combinations, the resulting function has more minterms, but can be reduced to the sum of products $f_2 = a \cdot \overline{b} + \overline{a} \cdot \overline{c}$,

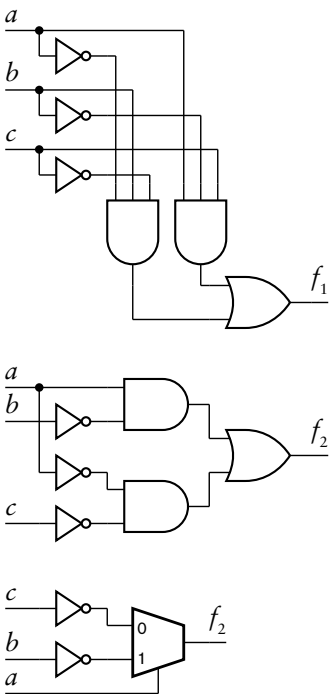

FIGURE 2.8 Realizations of a partial function.

implemented by either of the middle or bottom circuits in Figure 2.8. Our choice among these circuits may depend on the implementation fabric to be used. If we are simply concerned with minimizing the number of gate inputs, we would choose the middle circuit, yielding a result of 1 for the impossible input combinations. If our implementation fabric is based on sum-of-product circuit, and the minterms can also be shared as part of other functions in the system, we would choose the first, yielding a result of 0 for the impossible input combinations. Some implementation fabrics are based on multiplexers, introduced in Chapter 1, as the primitive circuit elements. If we were using such a fabric, we would choose the bottom circuit.

2.1.2 BOOLEAN ALGEBRA

The mathematical abstraction that we use as the foundation for digital design is *Boolean algebra*. It deals with Boolean expressions containing symbols that denote Boolean values, variables and operations. We can interpret the symbols as representing digital signals and gates.

Boolean algebra is based on a number of *axioms*. These are just Boolean equations that we take as given without requiring proof. The axioms of Boolean algebra are:

- Commutative laws:

$$x + y = y + x \tag{2.1}$$

$$x \cdot y = y \cdot x \tag{2.2}$$

- Associative laws:

$$(x + y) + z = x + (y + z) \tag{2.3}$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{2.4}$$

- Distributive laws:

$$x + (y \cdot z) = (x + y) \cdot (x + z) \tag{2.5}$$

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \tag{2.6}$$

- Identity laws:

$$x + 0 = x \tag{2.7}$$

$$x \cdot 1 = x \tag{2.8}$$

- Complement laws:

$$x + \overline{x} = 1 \tag{2.9}$$

$$x \cdot \overline{x} = 0 \tag{2.10}$$

Although we don't have to prove these laws, we can see that they make sense, since any consistent substitution of 0 and 1 values for variables in

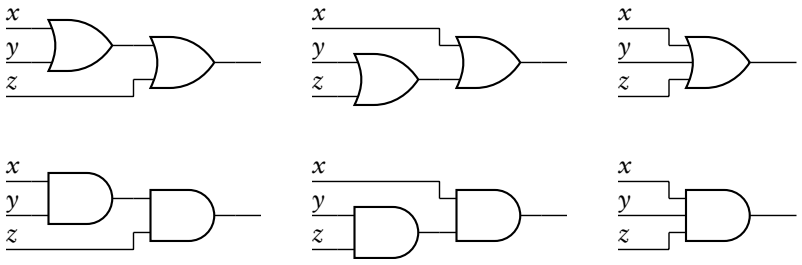

FIGURE 2.9 Circuits whose equivalence follows from the associative laws.

each law demonstrates the equality. The laws also suggest ways in which we can transform digital circuits while maintaining functional equivalence. For example, the commutative laws tell us that it doesn't matter which way around we connect the two inputs of an OR gate or an AND gate; we will get the same result either way. Similarly, the associative laws tell us that we don't need the parentheses when forming the logical OR or logical AND of three values, and that the circuits in each row in Figure 2.9 are equivalent. The distributive laws suggest how we can transform a circuit into sum-of-products form. This can be very useful, since many implementation fabrics allow efficient implementation of sum-of-product circuits.

Notice that we have presented the axioms in pairs, with each axiom being similar in form to the other in the pair. Each axiom is called the *dual* of the other in the pair. The *duality principle* of Boolean algebra states that we can take any Boolean equation and form its dual by interchanging the "+" and "·" operators and interchanging occurrences of 0 and 1; the dual is then a valid Boolean equation.

Given the axioms of Boolean algebra listed above, we can derive a number of further useful theorems:

- Idempotence laws:

$$x + x = x \tag{2.11}$$

$$x \cdot x = x \tag{2.12}$$

- Further identity laws:

$$x + 1 = 1 \tag{2.13}$$

$$x \cdot 0 = 0 \tag{2.14}$$

- Absorption laws:

$$x + (x \cdot y) = x \tag{2.15}$$

$$x \cdot (x + y) = x \tag{2.16}$$

- DeMorgan laws:

$$\overline{(x + y)} = \overline{x} \cdot \overline{y} \tag{2.17}$$

$$\overline{(x \cdot y)} = \overline{x} + \overline{y} \tag{2.18}$$

EXAMPLE 2.4 Prove the idempotence laws using just the axioms.

SOLUTION To prove law 2.11:

$$\begin{aligned} x + x &= (x + x) \cdot 1 && \text{by identity law 2.8} \\ &= (x + x) \cdot (x + \overline{x}) && \text{by complement law 2.9} \\ &= x + (x \cdot \overline{x}) && \text{by distributive law 2.5} \\ &= x + 0 && \text{by complement law 2.10} \\ &= x && \text{by identity law 2.7} \end{aligned}$$

Law 2.12 immediately follows, since it is the dual of law 2.11.

EXAMPLE 2.5 Suppose we are to implement the following Boolean function using AND and OR gates and inverters:

$$f = (x + y \cdot \overline{z}) \cdot \overline{(y \cdot z)}$$

If we were to implement it directly, as shown in Figure 2.10, the longest path through the circuit is four gates. Show how the Boolean equation for f can be transformed into sum-of-products form, thus reducing the length of the longest path.

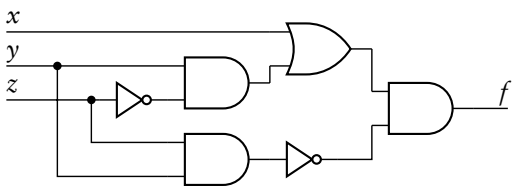

FIGURE 2.10 A circuit that directly implements a Boolean function.

SOLUTION We can transform the Boolean equation as follows:

$$\begin{aligned} f &= (x + y \cdot \overline{z}) \cdot \overline{(y \cdot z)} \\ &= (x + y \cdot \overline{z}) \cdot (\overline{y} + \overline{z}) && \text{DeMorgan law 2.18} \\ &= x \cdot (\overline{y} + \overline{z}) + (y \cdot \overline{z}) \cdot (\overline{y} + \overline{z}) && \text{distributive law 2.6} \\ &= x \cdot \overline{y} + x \cdot \overline{z} + y \cdot \overline{z} \cdot \overline{y} + y \cdot \overline{z} \cdot \overline{z} && \text{distributive law 2.6 twice} \\ &= x \cdot \overline{y} + x \cdot \overline{z} + 0 \cdot \overline{z} + y \cdot \overline{z} \cdot \overline{z} && \text{complement law 2.10} \\ &= x \cdot \overline{y} + x \cdot \overline{z} + 0 + y \cdot \overline{z} \cdot \overline{z} && \text{identity law 2.14} \\ &= x \cdot \overline{y} + x \cdot \overline{z} + 0 + y \cdot \overline{z} && \text{idempotence law 2.12} \\ &= x \cdot \overline{y} + x \cdot \overline{z} + y \cdot \overline{z} && \text{identity law 2.7} \end{aligned}$$

This reduced sum-of-products form of the Boolean equation can be implemented by the circuit of Figure 2.11, in which the longest path is reduced to three gates. Moreover, the circuit may be more efficiently implemented in this form in many fabrics.

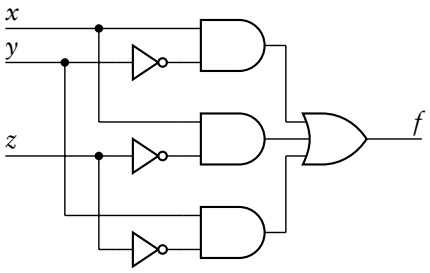

FIGURE 2.11 A circuit that implements the sum-of-products form.

The laws of Boolean algebra can be used to transform Boolean equations and their corresponding circuits, and to verify equivalence of Boolean expressions and circuits. However, they don't provide a recipe for finding an optimal circuit. That's mainly because the criteria for optimization depend on many different factors, including the implementation fabric to be used, power consumption constraints, physical packaging requirements, design resources available, and others. Optimization procedures, such as use of Karnaugh maps and the Quine-McClusky procedure, are described in many textbooks on digital logic design. They and other more involved procedures are founded on the laws of Boolean algebra. Given the complexity of the Boolean equations in real-world systems and the fact that computer aided design tools are needed to make optimization tractable, we won't go into the detail of the procedures in this book. Rather, we will focus on identifying the constraints that apply so that we can bring appropriate tools to bear on design problems.

2.1.3 VERILOG MODELS OF BOOLEAN EQUATIONS

In the design methodology described in Chapter 1, we focused on the use of models expressed in an HDL such as Verilog. Modern CAD tools are very good at analyzing, verifying and synthesizing Boolean functions expressed in an HDL. In this section, we will see how to express Boolean equations in Verilog. Later, as we introduce more complex combinational components and circuits, we will also show how they can be expressed in Verilog.

As we mentioned earlier, a Boolean equation in which a name is defined to be equal to a Boolean expression can be implemented by the circuit for the expression yielding an output with the given name. We can write a Boolean equation directly in Verilog using an *assignment statement* within a module. We use the keyword `assign`, then write the name of a net or port on the left hand side of the assignment symbol, "=", and a Verilog expression corresponding to the Boolean expression on the right hand side.

EXAMPLE 2.6 Develop a Verilog model for a circuit that implements the Boolean equation of Example 2.5.

SOLUTION The equation refers to three inputs, x, y and z, and one output, f. We represent them as input and output ports in the module definition. The module contains an assignment statement that represents the Boolean equation, as follows:

```
module circuit ( output f,
                 input  x, y, z );

  assign f = (x | (y & ~z)) & ~(y & z);

endmodule
```

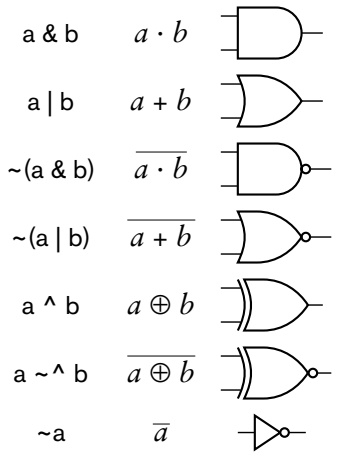

FIGURE 2.12 Verilog operators and their corresponding Boolean operations and gates.

In order to write arbitrary Boolean equations in Verilog, we need to know how to form Verilog expressions that mean the same as Boolean expressions. The example above uses the Verilog operators &, |, and ~, corresponding to the Boolean operators "·", "+" and the overbar notation, respectively. Verilog also provides the ^ and ~^ operators, corresponding to the XOR and XNOR operations and the XOR and XNOR gates that we introduced in Section 2.1.1. However, Verilog does not provide separate operators for the NAND and NOR operations. Instead, we model those operations using the ~ operator together with & and |. For example, we would write the NAND of a and b as ~(a & b). The Verilog operators, the Boolean expressions they represent and the corresponding gates are summarized in Figure 2.12. Note that Verilog makes the same assumptions about precedence of logical operations that we have made for Boolean expressions. The ~ operators are evaluated first, then & operators, and finally | operators. However, we can include parentheses in Verilog expressions, as we did in the assignment in Example 2.6, to clarify or force the order of evaluation of operators.

When we write Verilog models for combinational circuits, we should generally not try to rearrange the Boolean expressions to imply any particular circuit of gates or other components. Rather, we should express the Boolean equations in the way that makes them most readily understood, then let our CAD tools synthesize and optimize a circuit based on constraints and our chosen implementation fabric. CAD tools can usually do a much better job at this than we could do manually. Where a CAD tool requires us to rearrange an expression to enable an optimization, we should clearly document the change and the reason for it using comments in the model code.

EXAMPLE 2.7 Develop a Verilog model for a combinational circuit that implements the following three Boolean equations, representing part of the control logic for an air conditioner:

$$\text{heater\_on} = \text{temp\_low} \cdot \text{auto\_temp} + \text{manual\_heat}$$

$$\text{cooler\_on} = \text{temp\_high} \cdot \text{auto\_temp} + \text{manual\_cool}$$

$$\text{fan\_on} = \text{heater\_on} + \text{cooler\_on} + \text{manual\_fan}$$

SOLUTION The module definition defines the input and output ports and contains assignment statements for the Boolean equations, as follows:

```
module aircon ( output heater_on, cooler_on, fan_on,
                input  temp_low, temp_high, auto_temp,
                input  manual_heat, manual_cool, manual_fan );

  assign heater_on = (temp_low  & auto_temp) | manual_heat;
  assign cooler_on = (temp_high & auto_temp) | manual_cool;
  assign fan_on    =  heater_on | cooler_on  | manual_fan;

endmodule
```

A straightforward synthesis of a digital circuit from this model is shown at the top of Figure 2.13. There are two subcircuits, one each for heater_on and cooler_on. The outputs of these circuits then drive the third subcircuit for fan_on. For some implementation fabrics, however, CAD tools might transform the circuit as shown at the bottom of Figure 2.13. The logical OR operations

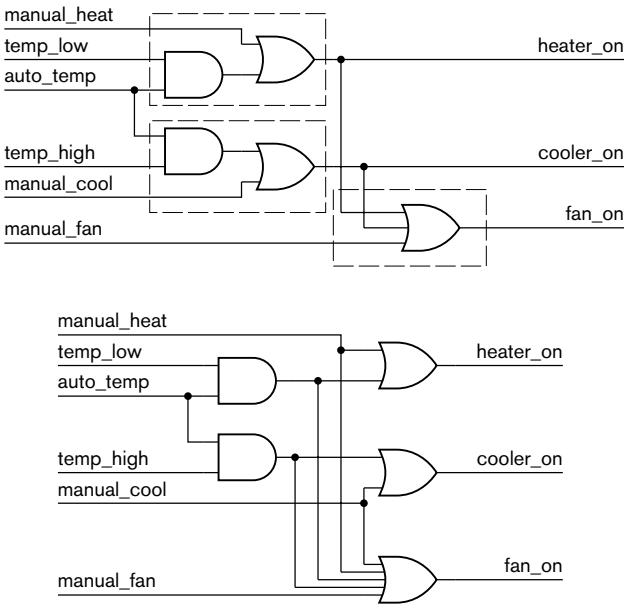

FIGURE 2.13 Circuits corresponding to the assignment statements for the air conditioner control logic.

that produce the heater_on and cooler_on outputs are replicated and merged with the logical OR operation for fan_on. This circuit would fit well in a sum-of-products implementation fabric, and would have reduced propagation delay in that fabric.

KNOWLEDGE TEST QUIZ

1. Write a truth table for the Boolean function $f = a \cdot \overline{b} + \overline{c}$.
2. Use truth tables to show that the Boolean expression $\overline{a \cdot b}$ is equivalent to $\overline{a} + \overline{b}$.
3. What is meant by a Boolean expression being in *sum-of-products* form?
4. Write the truth table for the AND-OR-invert gate shown in Figure 2.3.
5. Why are buffers used in digital circuits?
6. Use the "don't care" notation for inputs to compact the truth table for the function f_1 shown in Table 2.4.
7. What is the benefit of using the "don't care" notation for outputs in a truth table?
8. What is the dual of the following Boolean equation?
$$\overline{a + b \cdot c} = \overline{a} \cdot \overline{b} + \overline{a} \cdot \overline{c}$$
9. Write a Verilog assignment statement to model the Boolean equation $f = a \cdot \overline{b} + \overline{c}$.
10. Why should we generally not try to optimize Boolean equations manually when modeling them in Verilog?

2.2 BINARY CODING

Thus far, we have looked at digital representation of information that has two possible values and shown how we can use Boolean algebra as the formal basis for circuits that deal with such information. We now extend our discussion to dealing with information involving more than two values. An obvious example is numeric information. However, since representation and computation of numeric information is such an important and extensive topic, it deserves a chapter of its own (Chapter 3). First, we will look at more general principles that underlie digital representation of all forms of information.

We saw in Chapter 1 that we can represent two-valued information with two distinct voltage levels in a circuit. Using our digital abstraction, we called the levels "low" and "high," but then refined them to ranges

of voltages for pragmatic reasons. If we need to represent information that can take on N possible values, we could choose N distinct voltage levels (or voltage ranges, with intermediate thresholds). However, designing electronic circuits that can distinguish between more than two levels is extremely complex, and we would lose many of the benefits of binary digital circuits.

A better approach is to use multiple binary signals to represent a multivalued piece of information. Since each individual signal is binary, we can continue to use binary logic gates in our circuits with all of the advantages that they afford. We will use the values 0 and 1, as we did when discussing Boolean algebra, as the abstract values for each binary signal. We will continue to use the term *bit* to refer to these values.

Suppose that we have two signals, a_1 and a_0, available for representing some information. There are four possible combinations of binary values for the pair (a_1, a_0), namely, (0, 0), (0, 1), (1, 0) and (1, 1). Each possible combination is called a *code word*, and the set of all of the code words is called a *binary code*. Since a two-bit code has four possible code words, we can use a two-bit code to represent information with any number of values up to and including four. We just need to specify which code word corresponds to which value of the information. We say that a code word *encodes* the corresponding value.

EXAMPLE 2.8 Devise a binary code for the state of a road traffic light. The possible states are red, yellow and green.

SOLUTION Since there are three possible values to represent, we can use a two-bit binary code with one code word unused. One possible code is

red: (0, 0) yellow: (0, 1) green: (1, 0)

In this case, the code word (1, 1) is unused.

If two bits, with four possible code words, are not sufficient for the information we need to represent, we can just use more bits. In general an n-bit code has 2^n possible code words, so an n-bit code can represent information with up to 2^n values. Conversely, if we need to represent information with N values, we need at least $\lceil \log_2 N \rceil$ bits in our code. (The notation $\lceil x \rceil$ is called *ceiling* of x, and denotes the smallest integer that is greater than or equal to x.) We might choose a longer code, for a variety of reasons that we will explore, in which case there will be more unused code words.

EXAMPLE 2.9 Many ink-jet printers have six cartridges for different colored ink: black, cyan, magenta, yellow, light cyan and light magenta. A multi-bit signal in such a printer indicates selection of one of the colors. Devise a minimal length code for the signal.

SOLUTION Since there are six values to encode, the minimal length code is $\lceil \log_2 6 \rceil = 3$ bits long. There are $2^3 = 8$ possible code words, so two will remain unused. One possible code is

black: (0, 0, 1)	cyan: (0, 1, 0)	magenta: (0, 1, 1)
yellow: (1, 0, 0)	light cyan: (1, 0, 1)	light magenta: (1, 1, 0)

While it might make sense in some cases to use the shortest code, in other cases a longer code is better. A particular case of a non–minimal-length code is a *one-hot* code, in which the code length is the number of values to be encoded. Each code word has exactly one 1 bit with the remaining bits 0. The advantage of a one-hot code becomes clear when we want to test whether the encoded multibit signal represents a given value; we just test the single-bit signal corresponding to the 1 bit in the code word for that value.

EXAMPLE 2.10 Devise a one-hot code for the state of the traffic light described in a preceding example.

SOLUTION Since there are three values to encode, we need a 3-bit one-hot code. A possible code is

red: (1, 0, 0) yellow: (0, 1, 0) green: (0, 0, 1)

With this code, the left-most bit can be used to activate the red light, the middle bit to activate the yellow light, and the right-most bit to activate the green light. No additional circuitry is needed to decode the encoded signals to determine which light to activate.

2.2.1 USING VECTORS FOR BINARY CODES

Since a collection of binary coded bits conceptually represents a single piece of information, it would be convenient to be able to represent it as a single net in Verilog. We can do so using a *vector* net instead of using several individual nets. For example, if we need a net `w` to carry a 5-bit binary coded value, we could declare it as

```
wire [4:0] w;
```

This defines w to be a collection of five nets, w[4], w[3], w[2], w[1] and w[0], each of which is a single bit. Apart from condensing the declaration of the nets quite considerably, using vectors for encoded values gives us many other benefits, as we shall see throughout this book.

When we declare a vector net or port, the part in brackets (4:0 in the above example) specifies the index range for the elements of the vector. The first value is the index of the left-most element, and the second value is the index of the right-most element. If we want to number elements in descending order, we make the left-most index greater than the right-most index, as in the above example. We can also number elements in ascending order by making the left-most index less than the right-most index, as in the following:

```
wire [1:3] a;
```

Here, the elements from left to right are w[1], w[2] and w[3]. The choice between ascending and descending order is often a question of style, and may be addressed by coding guidelines used in an organization. This example also shows that we don't have to use 0 for the least index value; it can be any number.

EXAMPLE 2.11 Assume that the one-hot code for the traffic lights in Example 2.10 is represented using a 3-element vector with element 1 corresponding to red, 2 to yellow and 3 to green. Develop a Verilog model for a light controller that has an encoded input, an encoded output, and a single-bit input that enables the lights. When the enable input is 1, the encoded output is the same as the encoded input. When the enable input is 0, all bits of the output are 0.

SOLUTION One approach is to control each bit of the output by "AND-ing" the corresponding input with the enable bit. A module that does this is

```
module light_controller_and_enable
  ( output [1:3] lights_out,
    input  [1:3] lights_in,
    input        enable );
  assign lights_out[1] = lights_in[1] & enable;
  assign lights_out[2] = lights_in[2] & enable;
  assign lights_out[3] = lights_in[3] & enable;

endmodule
```

An alternative approach is to use the enable input to select whether to assign the input to the output (when enable is 1) or to set the output to all 0 bits otherwise. A module that takes this approach is

```
module light_controller_conditional_enable
  ( output [1:3] lights_out,
    input  [1:3] lights_in,
    input        enable );

  assign lights_out = enable ? lights_in : 3'b000;

endmodule
```

The assignment statement in this module uses the ? : operator to select between the alternatives. Note that we use the notation `3'b000` to form a literal vector value of three 0 bits. The notation `'b` specifies that a binary code word follows, and the number before `'b` specifies how many bits in the vector.

2.2.2 BIT ERRORS

While digital circuits are much more immune to noise than analog electrical circuits, they are not completely immune from interference. The effect of interference is occasionally to change the value of a signal from 0 to 1 or from 1 to 0. We sometimes prosaically call this a *bit flip*. If the signal is a single bit representing a logical condition, the rest of the circuit continues operating on the incorrect value, possibly causing erroneous outputs. If the signal is one of several bits in a binary-coded representation of some information, there are two possibilities. The flipped bit results in the code word being changed either to another valid code word or to a bit combination that is not a valid code word. If the result is a valid code word, the rest of the circuit operates on the incorrect value, as in the single-bit case, possibly producing erroneous outputs. If the result is an invalid code word, operation of the circuit depends on how we deal with invalid codes in the design.

One design approach is to consider invalid code words as "impossible" inputs, and not to specify the behavior of circuits that operate on invalid inputs. If we adopt this approach, the actual behavior of the circuits will depend on the implementation for the valid-code-word cases and on optimizations performed by CAD tools. It may be acceptable not to care about the circuit output values for invalid code words, particularly if cost reduction is a driving constraint. For example, in a mass-produced consumer toy, no one really cares about a once-a-year glitch, particularly if fixing it would increase the cost from $1.00 to $1.05.

If, on the other hand, the application demands more deterministic outputs, we can adopt a "fail safe" design approach. We can design our circuit to produce correct outputs for valid code words, and to produce known safe outputs should an invalid code word arise due to interference. For example, in our ink-jet printer of Example 2.9, if interference caused the signal for selecting the color to take on the code word (1, 1, 1), we could deliberately select no color, rather than spoiling a printout with incorrect colors or damaging the mechanism by trying to select more than one color at once.

EXAMPLE 2.12 In Example 2.10, we suggested that the bits of the one-hot-coded signal could be used to activate the red, yellow and green lights, respectively. However, an error in the three-bit signal could cause multiple lights to activate, or no light to activate. Design a circuit that causes the three lights to activate normally for valid one-hot code words, and for the red light to be activated alone for invalid code words.

SOLUTION Let us represent the three-bit signal with the bits *s_red*, *s_yellow* and *s_green*. The green light should be activated only when *s_green* is 1 and *s_yellow* and *s_red* are both 0. The Boolean equation is

$$green = \overline{s\_red} \cdot \overline{s\_yellow} \cdot s\_green$$

Similarly, the yellow light should be activated when *s_yellow* is 1 and *s_green* and *s_red* are both 0, giving the Boolean equation

$$yellow = \overline{s\_red} \cdot s\_yellow \cdot \overline{s\_green}$$

The red light should be activated when *s_red* is 1 and *s_yellow* and *s_green* are both 0, but it should also be activated in all other cases when neither the green nor yellow light is activated. The Boolean equation is

$$red = s\_red \cdot \overline{s\_yellow} \cdot \overline{s\_green} + \overline{(green + yellow)}$$

There are many other ways we could write this last Boolean equation, for example, by substituting for *green* and *yellow* and using the laws of Boolean algebra to rearrange it. However, we can leave that to a CAD tool, and simply enter the equations in the form above as part of a Verilog model.

A third design approach to dealing with errors introduced by interference is to have the circuit detect when they occur and then to take exceptional action. This is, in a sense, an extension of the "fail safe" approach. However, rather than producing a safe "normal" output, the circuit produces an "exceptional" output that indicates the circuit's function has not been performed correctly. An example of this approach is seen in modern

cars that include digital circuits to manage the engine. If an error arises, the circuit detects the error and illuminates a warning light in the instrument panel as its exceptional output. Detecting that interference has flipped a bit in a code word requires that the code include unused code words, and that the bit flip change a valid code word to one of the invalid code words. Circuits that use the encoded information can check for invalid code words and take action, such as suppressing outputs or activating an error signal. Of course, if interference causes a valid code word to change to a different valid code word, the error would not be detected.

One technique that is often used for error detection is *parity*, which refers to the number of bits that are 1 in a code word. Parity error checking involves increasing the code length by one bit, called the *parity bit*. In the *even parity* scheme, the parity bit in each augmented code word is set to 0 or 1 to ensure that the total number of 1 bits is even. For example, if the original code word is 1011, the augmented code word is 10111. (The converse *odd parity* scheme sets the parity bit to ensure that the total number of 1 bits is odd.) In an even parity scheme, valid augmented code words have even parity, and invalid augmented code words have odd parity. If interference causes a 0 bit to change to 1, the number of 1 bits is increased by one, making the parity odd. Similarly, if interference changes a 1 bit to 0, the number of 1 bits is decreased by one, again making the parity odd. So to check whether a bit has flipped, we simply count the number of 1 bits, including the parity bit. If the count is odd, parity has been reversed, so an error has occurred. If the count is even, either no error has occurred, or an even number of bits have been flipped, which we can't detect. In many applications, the probability of two or more bits flipping is much lower than the probability of one bit flipping, so it is acceptable not to be able to detect an even number of bit flips.

a_1	a_0	p
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 2.8 Truth table for the parity bit of a code of original length 2, giving even parity for the augmented code.

Counting the number of bits in a code word might, at first, seem a rather complicated function to perform. However, since we're only interested in whether the total is even, the task is much simpler. For a code of original length 2, the function p to generate the parity bit so that the augmented code has even parity is shown in the truth table in Table 2.8. As we can see, this function is equivalent to the exclusive-OR function. So we can use an exclusive-OR gate to generate the parity bit to augment a 2-bit code. We can extend this to augment a 3-bit code by taking the exclusive OR of the parity of two bits with the third bit. In general, for a code of any length, we can just take the exclusive OR of all of the bits. Since the exclusive-OR function is commutative and associative, the order in which we apply the exclusive OR to the bits of the code doesn't matter. A common approach is to use a parity tree, as shown in Figure 2.14, since it keeps the overall propagation delay small and avoids using gates with large numbers of inputs. The tree at the left of the figure generates the parity bit to augment an 8-bit code, creating a code of nine bits with even

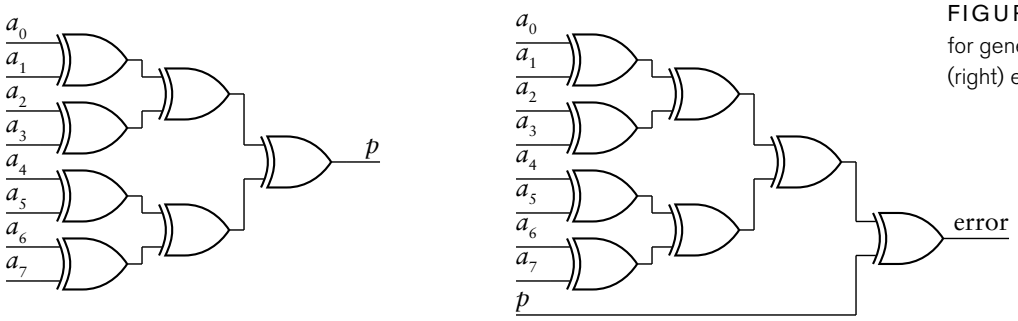

FIGURE 2.14 Parity trees for generating (left) and checking (right) even parity.

parity. The tree at the right checks the augmented code and yields a 1 if there is a parity error.

There are two problems with parity schemes. First, if interference flips two bits, parity is preserved, so we miss that error. The same applies if four, six, or any even number of bits are flipped. In many applications, however, the probability of multiple bits being flipped is extremely low, so the cost of a more elaborate error detection scheme is not warranted. The second problem is that for any given invalid code word, there are several possible bit flips from a valid code word that could yield the invalid code word. So while we can detect occurrence of a single-bit error, we can't tell which bit is in error. If detection of errors and taking some exceptional action is sufficient for the application, parity is a good choice. However, if corrective action is needed, the approach can be extended by including sufficient invalid code words in the code that a flip of any given bit yields a distinct invalid code word. When that invalid code word is detected, it indicates that the given bit has been flipped. So correcting the error is simply a matter of flipping it back, that is, using the negation of that bit's value. This kind of code is called an *error correcting code* (ECC).

The design of codes to provide for error detection and correction is a very broad topic area. We will return to it as part of our discussion of storage in Chapter 5, since that is one place where errors can arise. Meanwhile, when we design circuits that operate on binary coded information, we should think about how they should behave when interference produces bit errors.

KNOWLEDGE TEST QUIZ

1. How many code words are possible with a code of 5 bits?

2. What is the minimum number of bits needed to encode information with 12 possible values?

3. Devise a one-hot code to represent the days of the week (Monday through Sunday).

4. Write a Verilog declaration for a net, named w, representing an 8-bit binary coded value.

5. Write a Verilog assignment that drives each bit of w with a 0 value.

6. Does a single bit flip in a one-hot code word produce an invalid code word always, never, or sometimes?

7. How does extending a code with a parity bit to ensure odd parity enable detection of single-bit errors?

8. Can parity checking be used to correct the effect of a bit flip? If so, how? If not, why not?

2.3 COMBINATIONAL COMPONENTS AND CIRCUITS

In this section, we will introduce a number of combinational circuit components that are used as building blocks in larger digital systems. While these components can, themselves, be constructed from gates, it is generally not useful to do so. Instead, we will work at a higher level of abstraction. We will think of these components as basic blocks that, together with gates, are used to construct complex combinational circuits. We will rely on synthesis tools to refine our descriptions of such circuits into implementations using gates or other elements provided by the target implementation fabric. We will also return to the notion of negative logic, briefly mentioned in Chapter 1. The material presented in this section will form the basis for our consideration of larger-scale digital systems in later chapters.

2.3.1 DECODERS AND ENCODERS

In Section 2.2, we described how information can be binary coded. In many designs, we need to derive a number of control signals from a binary coded signal, with one control signal corresponding to each valid code word. When the encoded signal takes on a given code value, the corresponding control signal is activated. We call a circuit that derives the control signals in this way a *decoder*. For an n-bit code, if every code word is valid, the decoder will have 2^n outputs. As we shall see in Chapter 5, decoders are an important building block in memory designs.

We can derive the Boolean equation for each output of a decoder by looking at the corresponding code word. To illustrate, suppose we have an encoded 4-bit input signal (a_3, a_2, a_1, a_0), and we need to determine the Boolean equation for the output corresponding to the code word 1011. The output is 1 only when $a_3 = 1$, $a_2 = 0$, $a_1 = 1$ and $a_0 = 1$. Thus, the output is the value of the expression

$$a_3 \cdot \overline{a_2} \cdot a_1 \cdot a_0$$

A similar argument applies for other outputs. Each is the logical AND of the input bits, either directly (for bits that are 1 in the corresponding code word) or negated (for bits that are 1 in the corresponding code word).

EXAMPLE 2.13 Develop a Verilog model for a decoder for use in the ink-jet printer described in Example 2.9. The decoder has three input bits representing the choice of color cartridge and six output bits, one to select each cartridge.

SOLUTION A module with assignment statements representing the Boolean equations for the outputs is

```
module ink_jet_decoder ( output black, cyan, magenta, yellow,
                                 light_cyan, light_magenta,
                         input  color2, color1, color0 );

  assign black         = ~color2 & ~color1 &  color0;
  assign cyan          = ~color2 &  color1 & ~color0;
  assign magenta       = ~color2 &  color1 &  color0;
  assign yellow        =  color2 & ~color1 & ~color0;
  assign light_cyan    =  color2 & ~color1 &  color0;
  assign light_magenta =  color2 &  color1 & ~color0;

endmodule
```

If an invalid code occurs on the input bits, none of the outputs is activated. This can be considered a "fail safe" design.

The inverse of a decoder is called an *encoder*. It has, as inputs, a number of single-bit signals, and as outputs, a collection of signals representing the bits of an encoded value. We will assume for the moment that at most one of the inputs is 1 at any time, and the others are all 0. The code word at the output corresponds to the particular input that is 1.

We can derive the Boolean equation for each bit of the output by identifying those inputs for which the output bit is 1. The output bit is then the logical OR of those inputs. However, we need to take account of the possibility that none of the inputs is 1, since that would cause our encoder to output a code word of all 0 bits. If that code word is invalid, we can use it to imply that no inputs are 1, essentially extending the code. Alternatively, if the all-0s code word is valid and corresponds to one of the inputs being 1, we need to have a separate output that indicates when any of the inputs is 1. When this output is 0, we ignore the code word produced by the encoder.

EXAMPLE 2.14 Design an encoder for use in a domestic burglar alarm that has sensors for each of eight zones. Each sensor signal is 1 when an intrusion is detected in that zone, and 0 otherwise. The encoder has three bits of output, encoding the zone as follows:

Zone 1: 000 Zone 2: 001 Zone 3: 010 Zone 4: 011
Zone 5: 100 Zone 6: 101 Zone 7: 110 Zone 8: 111

SOLUTION Since all code words are used, we need a separate output to indicate when there is a valid code-word output. The module definition is

```
module alarm_eqn ( output [2:0] intruder_zone,
                   output       valid,
                   input  [1:8] zone );

  assign intruder_zone[2] = zone[5] | zone[6] |
                            zone[7] | zone[8];
  assign intruder_zone[1] = zone[3] | zone[4] |
                            zone[7] | zone[8];
  assign intruder_zone[0] = zone[2] | zone[4] |
                            zone[6] | zone[8];
  assign valid = zone[1] | zone[2] | zone[3] | zone[4] |
                 zone[5] | zone[6] | zone[7] | zone[8];

endmodule
```

The left-most bit of the output code is 1 when any of the zone 5 through zone 8 inputs is 1, so the equation for that output is the logical OR of those zone inputs. The equations for the other two output code bits are derived similarly. The valid output is the logical OR of all of the zone inputs.

Now let's consider the possibility of more than one input to an encoder being 1 at a time. The design we described above would produce an incorrect output, possibly an invalid code word. The solution is to assign priorities to the inputs, so that if multiple inputs are 1, the encoder outputs the code word corresponding to the input with highest priority. Such an encoder is called, not surprisingly, a *priority encoder*. One application of priority encoders is to prioritize interrupts in embedded systems. (We describe interrupts in Chapter 8.)

EXAMPLE 2.15 Revise the encoder for the burglar alarm to be a priority encoder, with zone 1 having highest priority, down to zone 8 having lowest priority.

SOLUTION The port list is unchanged, since we need the same inputs and outputs for the encoder. The truth table for the priority encoder is shown in

Table 2.9. From this, we can derive the Boolean equations for each bit of the output. A revised module definition is shown below.

```
module alarm_priority ( output [2:0] intruder_zone,
                        output       valid,
                        input  [1:8] zone );

  wire [1:8] winner;

  assign winner[1] = zone[1];
  assign winner[2] = zone[2] & ~zone[1];
  assign winner[3] = zone[3] & ~(zone[2] | zone[1]);
  assign winner[4] = zone[4] & ~(zone[3] | zone[2] | zone[1]);
  assign winner[5] = zone[5] & ~(zone[4] | zone[3] | zone[2] |
                                 zone[1]);
  assign winner[6] = zone[6] & ~(zone[5] | zone[4] | zone[3] |
                                 zone[2] | zone[1]);
  assign winner[7] = zone[7] & ~(zone[6] | zone[5] | zone[4] |
                                 zone[3] | zone[2] | zone[1]);
  assign winner[8] = zone[8] & ~(zone[7] | zone[6] | zone[5] |
                                 zone[4] | zone[3] | zone[2] |
                                 zone[1]);

  assign intruder_zone[2] = winner[5] | winner[6] |
                            winner[7] | winner[8];
  assign intruder_zone[1] = winner[3] | winner[4] |
                            winner[7] | winner[8];
  assign intruder_zone[0] = winner[2] | winner[4] |
                            winner[6] | winner[8];

  assign valid = zone[1] | zone[2] | zone[3] | zone[4] |
                 zone[5] | zone[6] | zone[7] | zone[8];
endmodule
```

zone								intruder_zone			
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(2)	(1)	(0)	valid
1	–	–	–	–	–	–	–	0	0	0	1
0	1	–	–	–	–	–	–	0	0	1	1
0	0	1	–	–	–	–	–	0	1	0	1
0	0	0	1	–	–	–	–	0	1	1	1
0	0	0	0	1	–	–	–	1	0	0	1
0	0	0	0	0	1	–	–	1	0	1	1
0	0	0	0	0	0	1	–	1	1	0	1
0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0	–	–	–	0

TABLE 2.9 Truth table for a priority encoder for a burglar alarm.

In this module, each element of the internal net winner indicates when the corresponding zone is 1 and has not lost to a higher priority zone. The encoder then uses the elements of the internal net instead of the zone inputs directly to generate the output code word. Another way of expressing this in Verilog is shown in the following module:

```
module alarm_priority_1 ( output [2:0] intruder_zone,
                          output       valid,
                          input  [1:8] zone );

  assign intruder_zone = zone[1] ? 3'b000 :
                         zone[2] ? 3'b001 :
                         zone[3] ? 3'b010 :
                         zone[4] ? 3'b011 :
                         zone[5] ? 3'b100 :
                         zone[6] ? 3'b101 :
                         zone[7] ? 3'b110 :
                         zone[8] ? 3'b111 :
                         3'b000;

  assign valid = zone[1] | zone[2] | zone[3] | zone[4] |
                 zone[5] | zone[6] | zone[7] | zone[8];
endmodule
```

The conditional assignment in this module tests a series of conditions to determine the value to assign to the net intruder_zone. First the zone 1 input is tested, and the result assigned 000 if the zone 1 input is 1. Otherwise, the zone 2 input is tested, and the result assigned 001 if the zone 2 input is 1. Testing continues in this way, with priority implied by the order of testing the conditions. This form of assignment for priority encoding is much easier to understand, and leaves the hard work of determining and optimizing the Boolean equations to the synthesis CAD tool.

BCD Code and 7-Segment Decoders

One form of information that we might wish to encode is numeric information. As we mentioned earlier, we will look at this topic in detail in Chapter 3. However, in this section, we will look at a particular form of numeric coding called *binary coded decimal* (BCD). If we consider just a single decimal digit, the ten possible values are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. We need at least 4 bits in a binary code for these values. There are a large number of possible codes, but BCD is the most common, having the following code words:

0: 0000 1: 0001 2: 0010 3: 0011 4: 0100
5: 0101 6: 0110 7: 0111 8: 1000 9: 1001

If we have more than one decimal digit of information to represent, we simply use groups of four bits, with each group corresponding to one decimal digit. For example, a system that deals with three-digit numbers would use a 12-bit code. The number 493 would be encoded as 0100 1001 0011.

Many digital systems display decimal numbers using 7-segment displays. Each display digit consists of seven separate lights, arranged as shown in Figure 2.15. If we have a digit encoded using BCD and we need to display the digit on a 7-segment display, we need a *7-segment decoder*. Strictly speaking, we should call it a "7-segment code converter," since it converts from a BCD code input to a 7-segment code output. However, the term "7-segment decoder" is widely used. Assuming a segment is lit if its input is 1, we need a 7-bit code for representing the digits 0 through 9. The code word for each digit has a 1 bit corresponding to each segment that is lit and a 0 bit corresponding to each segment that is not lit. A 7-segment decoder then converts between BCD and this 7-bit code. One possible code is shown in Figure 2.16, with the bits corresponding left to right with segments g through a.

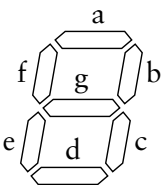

FIGURE 2.15 A 7-segment display digit. The segments are named "a" through "g," as shown.

FIGURE 2.16 A 7-segment code for decimal digits. In each code word, the bits correspond to segments g through a in left-to-right order.

EXAMPLE 2.16 Develop a Verilog model for a 7-segment decoder. Include an additional input, blank, that overrides the BCD input and causes all segments not to be lit.

SOLUTION We could determine the BCD code words that result in each segment being lit, and so derive Boolean equations for each segment output. However, that would make the model hard to understand. A better approach is to list the 7-bit code word corresponding to each BCD code word, as we did in Figure 2.16. A module that does this is

```
module seven_seg_decoder ( output [7:1] seg,
                           input  [3:0] bcd,
                           input        blank );
reg [7:1] seg_tmp;
```

(*continued*)

```
  always @*
    case (bcd)
      4'b0000: seg_tmp = 7'b0111111;  // 0
      4'b0001: seg_tmp = 7'b0000110;  // 1
      4'b0010: seg_tmp = 7'b1011011;  // 2
      4'b0011: seg_tmp = 7'b1001111;  // 3
      4'b0100: seg_tmp = 7'b1100110;  // 4
      4'b0101: seg_tmp = 7'b1101101;  // 5
      4'b0110: seg_tmp = 7'b1111101;  // 6
      4'b0111: seg_tmp = 7'b0000111;  // 7
      4'b1000: seg_tmp = 7'b1111111;  // 8
      4'b1001: seg_tmp = 7'b1101111;  // 9
      default: seg_tmp = 7'b1000000;  // "-" for invalid code
    endcase

  assign seg = blank ? 7'b0000000 : seg_tmp;

endmodule
```

We have written the list of code-word values in a *case statement* contained within an *always block*. (An always block is one kind of *procedural block*; we shall return to the other kind in Section 2.4). For a combinational function, the always block starts with an *event list* of the form @*, indicating that the block responds to any change of value on any of the inputs to the function. The case statement includes an expression in parentheses whose value is used to select among the alternatives. Each alternative lists a possible value of the expression (before the : character) and has an assignment to seg_tmp. The default alternative in the case statement deals with values not explicitly listed. In this module, the default alternative deals with invalid codes. Note that Verilog requires the target of an assignment within a procedural block to be declared as a *variable*, in this case using the keyword reg, instead of as a net using the keyword wire. The difference is that a variable retains the value assigned within a block, whereas a net continuously gains its value from an assignment statement (written outside a block using the assign keyword) or from a connection to an instance. The final assignment statement within the module uses the blank input to determine whether to drive the encoded output with all 0s, causing all segments not to be lit, or to copy the value decoded from the BCD input to the output.

2.3.2 MULTIPLEXERS

Multiplexers are an important building block in many digital systems. We introduced a simple multiplexer in Section 1.2. It has two data inputs, one data output, and a select input that determines which input value is used for the output value. We can expand on this simple multiplexer along two dimensions. First, we can add more data inputs, which also requires adding

further select inputs to encode the choice of input to drive the output. Second, we can use multiplexers in parallel to select between two sources of multibit encoded data. Let's look at the alternatives in more detail.

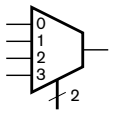

FIGURE 2.17 A 4-to-1 multiplexer.

Suppose that, instead of selecting between two input bits, we need to select between four input bits. Since there are four input sources, we need to have four values for the select input. We can encode the select input using two bits. Figure 2.17 shows a 4-to-1 multiplexer. The select input is drawn as a thicker line to indicate that it is a multibit encoded input. In this book, we will mostly use line thickness to distinguish between single-bit and multibit signals. Occasionally, where we want to emphasis that a signal is multibit, we will add a stroke across the line and show the number of bits, as in Figure 2.17. The code for the select input is

00: input 0 01: input1 10: input 2 11: input 3

We could describe a gate circuit to implement the multiplexer, but there is little point, for two reasons. First, a synthesis tool would probably optimize the circuit, changing it from what we specify. Second, in a number of implementation fabrics, multiplexers can be constructed from individual transistors more efficiently than as a circuit of gates. Multiplexers would be considered primitive elements in those fabrics. So instead of a gate-level circuit, we will just consider how to express a multiplexer function in Verilog.

EXAMPLE 2.17 Develop a Verilog model for a 4-to-1 multiplexer.

SOLUTION The module definition is

```
module multiplexer_4_to_1 ( output reg       z,
                            input      [3:0] a,
                            input            sel );

  always @*
    case (sel)
      2'b00: z = a[0];
      2'b01: z = a[1];
      2'b10: z = a[2];
      2'b11: z = a[3];
    endcase

endmodule
```

The case statement in the always block uses the value of the sel input to determine which input bit to copy to the output. This example illustrates a further point about using always blocks to model combinational functions. As we

mentioned in Example 2.16, the target of the assignments in the block must be declared as a variable, using the keyword reg in this case. When the target is a port of the module, the reg declaration can be combined with the output port declaration.

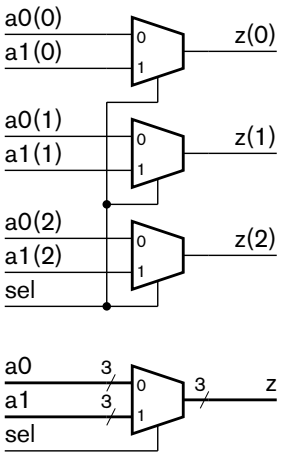

FIGURE 2.18 A circuit for a 2-to-1 multiplexer for 3-bit data sources (top), and a symbol for the multiplexer (bottom).

We can further expand this multiplexer to have eight data inputs, which would require a 3-bit select input. The number of data inputs need not be a power of 2. If it is not, then the select input code will have unused code words. We must then ensure that an invalid code word is never presented to the select input. In general, a multiplexer having N input bits needs $\lceil \log_2 N \rceil$ bits for the select input, since the select input carries a binary code requiring N values.

Now let's consider using multiplexers to select between two sources of encoded data. If the code length is m (that is, each code word has m bits), we can use m two-input multiplexers, one for each bit of the two data sources. This is illustrated in Figure 2.18 for selecting between two sources each of three bits. The circuit at the top of the figure shows the three separate 2-to-1 multiplexers. At the bottom of the figure is a symbol that represents a 2-to-1 multiplexer operating on the 3-bit encoded data inputs and output.

EXAMPLE 2.18 Develop a Verilog model for the 3-bit 2-to-1 multiplexer.

SOLUTION The module definition is

```
module multiplexer_3bit_2_to_1 ( output [2:0] z,
                                 input  [2:0] a0, a1,
                                 input        sel );

  assign z = sel ? a1 : a0;

endmodule
```

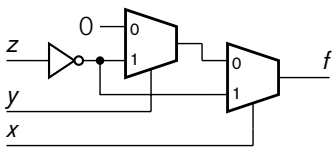

FIGURE 2.19 Implementing a Boolean function using multiplexers.

We can, of course, combine these two forms of expansion. If we need to select between N sources of data, each of which is encoded with m bits, we simply use m lots of N-to-1 multiplexers. The details are left as an exercise.

Before we leave the topic of multiplexers, it is interesting to note that all Boolean functions can be expressed in terms of multiplexers combined with negation. To illustrate, consider the function that we examined earlier, $f = (x + y) \cdot \bar{z}$ whose truth table is shown in Table 2.2. This function can be implemented using the circuit shown in Figure 2.19. Note the use of a literal 0 value for one input. This can be implemented by hard wiring

the input to the 0V ground. We won't go into the general principles of how to implement Boolean functions using multiplexers here. We raise the topic since multiplexers can be very efficiently implemented in some fabrics. As an example, the basic circuit elements in FPGAs manufactured by Actel Corporation consist of two multiplexers and a small number of other associated components. However, the details of mapping arbitrary Boolean equations to multiplexers are generally handled by CAD tools.

2.3.3 ACTIVE-LOW LOGIC

Thus far, we have focused on circuits in which a low logic level represents the falsehood of some condition and a high logic level represents truth of the condition. In Chapter 1, we identified this convention as *positive logic*, or *active-high logic*. In principle, the correspondence of low with falsehood and high with truth is largely arbitrary. We could just as well represent falsehood with a high logic level and truth with a low logic level, a convention that we referred to in Chapter 1 as *negative logic*, or *active-low logic*. Note that "positive" and "negative" in this context don't refer to the voltage polarity, but simply distinguish between the two conventions. We will use the terms "active high" and "active low" to avoid the confusion. We will also maintain the convention of associating 0 with a low logic level and 1 with a high logic level.

In a circuit that mixes both active-low and active-high logic, we could get confused about which convention is used for which signal. We should still label signals with the conditions they represent so that we can understand the intended function of the circuit. A commonly adopted approach is to label an active-low signal with the negation of the condition it represents. For example, an active-low signal representing the condition that a lamp is lit would be labeled $\overline{\text{lamp\_lit}}$, since the signal is 1 when the condition is false and 0 when the condition is true.

One reason for using active-low logic is that some kinds of digital circuits are able to sink more current when driving an output low than they can source when driving the output high. If such an output is used to activate some condition for which current flow is required, it would be better to use a low logic level rather than a high logic level.

EXAMPLE 2.19 Revise the night-light circuit from Figure 1.3 in Chapter 1 by connecting the lamp to the positive power supply instead of to ground.

SOLUTION To make current flow in the lamp and light it, we need to drive the controlling signal low. Thus, we must use an active-low signal to implement the "lamp lit" condition. This is shown in Figure 2.20, in which the controlling signal is labeled $\overline{\text{lamp\_lit}}$. The gate performs the logical AND function of the lamp_enabled and dark signals, but its output must be negated to match the

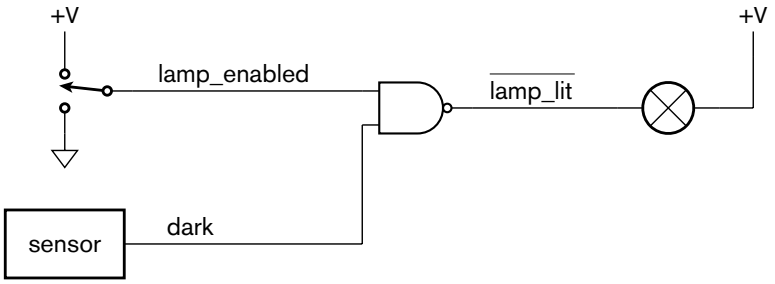

FIGURE 2.20 The night-light circuit using an active-low signal.

negation of the "lamp lit" condition. Hence, we use a NAND gate in place of the AND gate in the original circuit.

In general, this approach to dealing with active-low logic involves matching negation "bubbles" on components with active-low signals. When we do that, no negation of the logical condition represented by the signal is implied. Thus, we can interpret the circuit of Figure 2.20 as saying, "The lamp is lit when the lamp is enabled and it is dark." If we connect an active-low signal to a component without a bubble at the connection point, we are implying negation of the logical condition represented by the symbol.

EXAMPLE 2.20 Returning to the original night-light circuit from Figure 1.3 in Chapter 1, think of the sensor as having an active-low output representing the condition "it is light." Redraw the circuit to take account of this change.

SOLUTION As shown in Figure 2.21, we label the signal connected to the sensor $\overline{\text{light}}$ to show that it is active-low. We draw a bubble on the sensor output to indicate it is an active-low output. There is no negation implied by the connection at the sensor output, since we have a bubble output connected to an active-low signal. However, since there is no bubble on the AND gate input, logical negation is implied for its connection to $\overline{\text{light}}$. Thus, we can interpret the circuit as saying, "The lamp is lit when the lamp is enabled and it is not light."

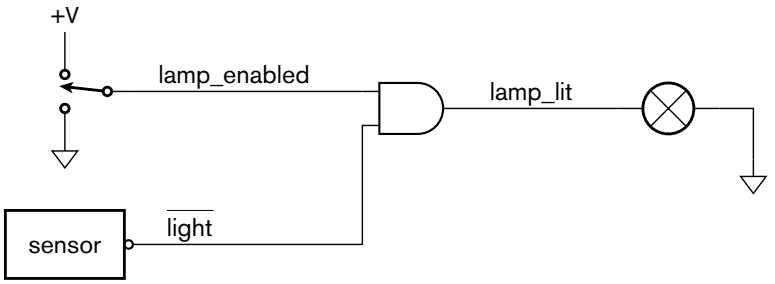

FIGURE 2.21 The night-light circuit with negation implied by connecting an active-low signal to an active-high input.

When we draw gate circuits for Boolean functions, it is important to use AND and OR gates as appropriate for the logical operations applied to conditions represented by signals. If any of those signals are active-low, and no implicit negation is intended, we should "draw a bubble" where the signal connects to a gate. We can make use of DeMorgan's laws to derive alternate views of gates. For example, Equation 2.18 tells us that the component that we have called a NAND gate when operating on active-high inputs can also perform an OR function upon conditions represented by active-low inputs. We can draw two distinct symbols for the gate component, as shown in Figure 2.22. It is important to realize, however, that both symbols represent the same circuit of interconnected transistors!

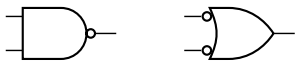

FIGURE 2.22 Alternate logic symbols for a gate.

One of the problems we encounter when modeling designs with active-low signals in Verilog is that we don't have a way of drawing a negation bar over a signal name or drawing a bubble on a port. Instead, we usually adopt a textual naming convention, such as appending the suffix "_N" to a name, to indicate which signals and ports are active-low. For example, a Verilog model might give the active-low output of the sensor in Figure 2.21 the name light_N. A Verilog model for the sensor would assign 0 to the light_N signal when it is light and 1 when it is dark. The model for the AND gate assigns 1 to its output when both inputs are 1, and 0 to its output otherwise. Thus, the lamp_lit signal is assigned 1 when lamp_enabled is 1 ("the lamp is enabled") and light_N is 1 ("it is not light"). When we're dealing with active-low logic in Verilog models, we need to think carefully about which Verilog value represents truth or falsehood of each condition, and design accordingly.

KNOWLEDGE TEST QUIZ

1. For a decoder with inputs (a_2, a_1, a_0), write the Boolean equation for the output corresponding to the code word 100.
2. What would be the output of the encoder in Example 2.14 if both the Zone 2 and Zone 3 inputs were 1 at the same time? Would this output be correct?
3. What problem would arise if we did not include the valid output from the encoder in Example 2.14?
4. How does a priority encoder solve the problem of multiple inputs being 1 at the same time?
5. What decimal digit is represented by the BCD code 0101?
6. What is the 7-segment code corresponding to the BCD code 0011?
7. What is the purpose of a multiplexer?
8. How many select input bits are needed for a 6-to-1 multiplexer?

9. How can we construct a 2-to-1 multiplexer for 5-bit encoded data inputs?
10. What logic level would you expect on a signal labeled door_closed, connected to a door sensor, when the door is open?
11. If a Verilog net named motor_on_N represents an active-low signal, what Verilog value would you assign it to turn the motor on?

2.4 VERIFICATION OF COMBINATIONAL CIRCUITS

In Section 1.5 we introduced a design methodology to guide us in the design and implementation of a digital system. The first task was to develop and enter a design description based on the application's requirements and constraints. In this chapter, we have seen examples of design descriptions, expressed as schematics and as Verilog models, for simple combinational circuits and components. Most systems are more involved and include sequential components as well as combinational subcircuits, so there is a limit to how much of the methodology we can demonstrate. Nonetheless, there are small-scale applications where combinational circuits are sufficient, so we will show how we can apply our design methodology to them.

The second step in our design methodology is functional verification, that is, ensuring that the design performs the operation required of it. Since, in a combinational circuit, the values of the outputs depend only on the current values of the inputs, we can simply verify that the circuit produces the required output for each combination of input values. For a design description expressed in Verilog, we can develop a *testbench* model that provides input values to the *design under verification* (DUV) and checks that the output values are correct. The DUV is also frequently called a *device under test* (DUT), but that usage may be confused with physical testing of manufactured devices. We will use the term DUV in this book to avoid the confusion. The testbench model is, itself, a Verilog model that we can execute using a simulator. However, it is not intended to describe hardware that will be built. Rather, its purpose is to apply a sequence of values, called *test cases*, to the input connections of the DUV, and to monitor the output connections to ensure that correct values are produced. The DUV is usually an instance of the Verilog module that describes the design. A simulator mimics the passage of time, executing the DUV and testbench models, and assigning values to nets and variables at appropriate simulated times.

The difficult part of developing a testbench model is working out how to express the correctness conditions. If the requirements are expressed as

Boolean equations, the design will probably implement those equations directly, so expressing the correctness conditions as Boolean equations gains nothing. A better approach is to determine some more abstract conditions that are required to hold, and to test that the design satisfies those conditions.

EXAMPLE 2.21 Develop a testbench model for the light_controller_and_enable module for the traffic light control circuit of Example 2.11. Verify the conditions that, when the enable input is 1, the output is the same as the light input, and when the enable input is 0, all light outputs are inactive.

SOLUTION The testbench model includes an instance of the design under verification, as well as code to apply test cases and to check for correct outputs. The organization of these components is shown in Figure 2.23.

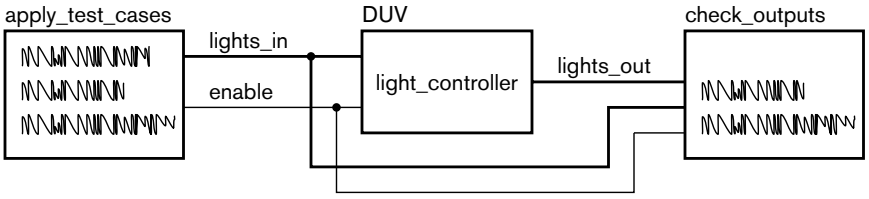

FIGURE 2.23 Organization of the testbench for the light controller.

Since the testbench is a Verilog model, it needs a modue definition. However, since there are no external connections to the testbench, the module has no ports. The module definition is

```
`timescale 1ms/1ms

module light_testbench;

  wire [1:3] lights_out;
  reg  [1:3] lights_in;
  reg        enable;

  light_controller_and_enable duv ( .lights_out(lights_out),
                                    .lights_in(lights_in),
                                    .enable(enable) );

  initial begin
          enable = 0; lights_in = 3'b000;
    #1000 enable = 0; lights_in = 3'b001;
    #1000 enable = 0; lights_in = 3'b010;
```

(*continued*)

```
    #1000 enable = 0; lights_in = 3'b100;
    #1000 enable = 1; lights_in = 3'b001;
    #1000 enable = 1; lights_in = 3'b010;
    #1000 enable = 1; lights_in = 3'b100;
    #1000 enable = 1; lights_in = 3'b000;
    #1000 enable = 1; lights_in = 3'b111;
    #1000 $finish;
  end

  always @(enable or lights_in) begin
    #10
    if (!( ( enable && lights_out == lights_in) ||
           (!enable && lights_out == 3'b000) ))
      $display("Error in light controller output");
  end

endmodule
```

The first line is a *time-scale directive* that indicates to the simulator the time units to be used for delays in the model. In Verilog models, delays are specified as numbers without units. The timescale directive is required to ascribe units to these numbers. In our example, we specify that delays are multiples of 1ms (the first number in the directive), with a precision of 1ms.

Within the module, duv is an instantiation of the light_controller_and_enable module that describes the traffic light control circuit. The input and output ports of the instance are connected to internal variables and nets declared within the testbench module. Note that we have used *named* port connection here, rather than *positional* port connection as we did in Example 1.5. In named association, we write the name of the module port after the "." symbol and the variable or net to which it is connected within parentheses. This allows us to write the connections in any order, rather than following the order of ports in the module. Given the advantages and clarity of named connection, we will use it in models from now on.

Following the instantiation statement is an *initial block* that applies test cases to the DUV. An initial block is the second kind of procedural block, along with the always block that we introduced in Section 2.3. In general, a procedural block is a collection of Verilog statements that are executed one after another, much like statements in a programming language. An initial block starts executing at the beginning of simulation, and when the last statement in the block has been executed, it terminates. We only use initial blocks in testbench models, not in models for circuit designs. In particular, we don't use initial blocks to set the initial conditions for sequential circuits. We will see how to reset sequential circuits in Chapter 4.

The first line of the initial block in this module makes an assignment to the enable input, followed by an assignment to the lights_in input. These two assign-

ments constitute application of one test case to the inputs of the DUV. The block then delays, indicated by the # symbol, for 1000 units of simulated time. Since the timescale directive specified 1ms as the time unit, the delay is for 1 second of simulated time. During this delay, other parts of the model, including the instance of the lights controller, continue executing. After the 1 second delay, the block continues, applying the next test case to the DUV inputs and then waiting a further second of simulated time. The block continues in this way until it reaches the last statement, which is a $finish system task. System tasks, identified by the $ symbol, are built-in operations performed by the simulator. The $finish system task finishes simulation and exits the simulator.

The procedural block at the end of the module is an always block. In Verilog, an always block typically responds to some event, described by the event list after the @ symbol. Whenever that event occurs, the statements in the always block are executed. The block then waits for another occurrence of the event. In this module, the always block has the job of ensuring that the DUV outputs meet the requirements. In developing this block, we need to determine when to check the outputs. If we were to check them at the same time as changing the inputs, the DUV would not yet have responded to the input change, and the outputs would still reflect the previous inputs. In this example, we will wait for an interval of 10ms of simulated time after an input change before checking the outputs. The always block responds to a change in value of either (or both) of the inputs enable or lights_in. When that occurs, the block delays for the 10ms interval. It then tests whether there is an incorrect output from the DUV, and if so, displays an error message using the $display system task. Note that in checking the condition, we use the logical operators && (logical AND), || (logical OR) and ! (logical NOT), rather than the &, |, and ~ operators we used previously. The forms used here deal with truth values and should be used for condition tests in if statements. The forms we used previously deal with bit and vector values and should be used in Boolean equations.

One thing to note about the test cases in this example is that not all possible input combinations are included. While it might be feasible to extend this testbench to be exhaustive, for larger designs, that would be intractable. Even if we wrote Verilog code to generate the input combinations automatically, rather than writing them out explicitly, a simulation would take too long to execute. That is because the number of test cases rises exponentially with the number of inputs. At issue here is the *functional coverage* of our testbench, that is, the proportion of the possible input combinations we have exercised. In the example, we have covered the usual operational cases and two unusual cases. In a larger model, we would have to be selective, and perhaps just cover a "typical" sample of normal cases plus a few unusual cases. We will return to the topic of coverage as part of our more detailed discussion of design methodology in Chapter 10.

Another thing to note about the test cases in the example is that the Verilog code is very repetitious. Each test case involves an assignment to the two inputs, followed by waiting for an interval. In larger models, there are more statements for each test case, and writing them repeatedly can be error prone. Fortunately, Verilog provides a feature that lets us abstract out the common parts of the test cases. We can write a *task* containing the common statements, and invoke the task once for each test case. We provide the particular values to use in each test case as ports to the procedure.

EXAMPLE 2.22 Revise the testbench model of Example 2.21 to use a task for applying the test cases.

SOLUTION The entity declaration is unchanged. The revised module definition is

```
`timescale 1ms/1ms

module light_testbench1;

  wire [1:3] lights_out;
  reg  [1:3] lights_in;
  reg        enable;

  task apply_test ( input       enable_test,
                    input [1:3] lights_in_test );
    begin
      enable = enable_test; lights_in = lights_in_test;
      #1000;
    end
  endtask

  light_controller_and_enable duv ( .lights_out(lights_out),
                                    .lights_in(lights_in),
                                    .enable(enable) );

  initial begin
    apply_test(0, 3'b000);
    apply_test(0, 3'b001);
    apply_test(0, 3'b010);
    apply_test(0, 3'b100);
    apply_test(1, 3'b001);
    apply_test(1, 3'b010);
    apply_test(1, 3'b100);
    apply_test(1, 3'b000);
    apply_test(1, 3'b111);
    $finish;
  end
```

(*continued*)

```
  always @(enable or lights_in) begin
    #10
    if (!( ( enable && lights_out == lights_in) ||
           (!enable && lights_out == 3'b000) ))
      $display("Error in light controller output");
  end

endmodule
```

The difference between this testbench and the one in Example 2.21 is the inclusion of the apply_test task definition within the module. The task definition contains the statements needed to apply each test case. The values to be applied are represented by the ports enable_test and lights_in_test. Each of the port definitions looks similar to the definition of an input port of a module, specifying the direction (into the task in this case), name, and index range for the parameter.

In the initial block, we invoke, or *call*, the task, once per test to be applied. Within parentheses in the task call, we supply the actual values to be used for the task ports for that call. The task then performs the statements in the task body, using those values in place of the port names. When the task statements finish, the task call is complete.

Having verified the functionality of the design, the next task in the design methodology is synthesis. To do that, we need to know what implementation fabric will be used, since synthesis involves refining the design to a structural implementation using primitive elements from the implementation fabric. We will discuss implementation fabrics, including those that can be used for combinational circuits, in more detail in Chapter 6. However, if the circuit is very simple, involving just a few gates, we may be able to use single gates packaged individually. This kind of circuit is sometimes needed as part of a larger system involving off-the-shelf ICs that must be connected together. If one of the ICs has outputs that differ slightly in function from the inputs of another, a small combinational circuit can deal with the differences.

EXAMPLE 2.23 A processor IC has three active-high outputs to control a memory that stores data: mem_en to enable operation of the memory, rd to control reading of data from the memory, and wr to control writing of data to the memory. A memory IC, however, has two active-high inputs: mem_rd to cause it to read data, and mem_wr, to cause it to write data. All other interconnections between the processor and memory are mutually compatible. Implement an interface circuit to compensate for the differences.

SOLUTION The mem_rd input to the memory can be derived using an AND gate applied to the mem_en and rd outputs of the processor. Similarly, the mem_wr input can be derived using an AND gate applied to the mem_en and wr outputs. Thus, we just need two AND gates. These could be implemented using two 1G08 devices, each of which contains a single AND gate in a small 5-pin package that can be used on a printed circuit board. Given the simplicity of this circuit, we would synthesize it manually. That is, we would just instantiate AND-gate components in a structural model of the entire system.

KNOWLEDGE TEST QUIZ

1. What is the purpose of a testbench model?
2. Write a Verilog statement to delay for 1 time unit and then to apply a test-case value of 0101 to a variable named s.
3. What does a Verilog always block do when execution reaches the last statement in the block?
4. Why should a block that checks outputs of a combinational circuit not check them at the same time that the inputs change?
5. When might it be appropriate to implement a combinational circuit using discrete logic gates in individual packages?
6. What is a PLD?