I_3 will be connected to the output line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only 1 three-state buffer has access to the output while all other buffers are maintained in a high-impedance state. One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0 and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation reveals that this circuit is another way of constructing a four-to-one-line multiplexer.

4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

The Verilog HDL was introduced in Section 3.10. In the current section, we introduce additional features of Verilog, present more elaborate examples, and compare alternative descriptions of combinational circuits in Verilog. Sequential circuits are presented in Chapter 5. As mentioned previously, the module is the basic building block for modeling hardware with the Verilog HDL. The logic of a module can be described in any one (or a combination) of the following modeling styles:

- Gate-level modeling using instantiations of predefined and user-defined primitive gates.
- Dataflow modeling using continuous assignment statements with the keyword **assign**.
- Behavioral modeling using procedural assignment statements with the keyword **always**.

Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other. Dataflow modeling is used mostly for describing the Boolean equations of combinational logic. We'll also consider here behavioral modeling that is used to describe combinational and sequential circuits at a higher level of abstraction. Combinational logic can be designed with truth tables, Boolean equations, and schematics; Verilog has a construct corresponding to each of these "classical" approaches to design: user-defined primitives, continuous assignments, and primitives, as shown in Fig. 4.31. There is one other modeling style, called switch-level modeling. It is sometimes used in the simulation of MOS transistor circuit models, but not in logic synthesis. We will not consider switch-level modeling.

Gate-Level Modeling

Gate-level modeling was introduced in Section 3.10 with a simple example. In this type of representation, a circuit is specified by its logic gates and their interconnections. Gate-level modeling provides a textual description of a schematic diagram. The Verilog HDL

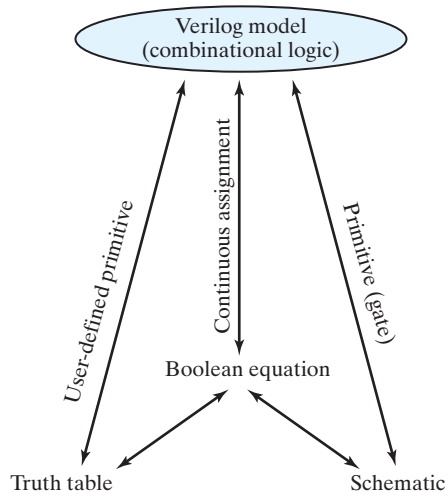

FIGURE 4.31
Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type. The other eight are the same as the ones listed in Section 2.8. They are all declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**. Primitives such as **and** are *n*-input primitives. They can have any number of scalar inputs (e.g., a three-input **and** primitive). The **buf** and **not** primitives are *n*-output primitives. A single input can drive multiple output lines distinguished by their identifiers.

The Verilog language includes a functional description of each type of gate, too. The logic of each gate is based on a four-valued system. When the gates are simulated, the simulator assigns one value to the output of each gate at any instant. In addition to the two logic values of 0 and 1, there are two other values: *unknown* and *high impedance*. An unknown value is denoted by **x** and a high impedance by **z**. An unknown value is assigned during simulation when the logic value of a signal is ambiguous—for instance, if it cannot be determined whether its value is 0 or 1 (e.g., a flip-flop without a reset condition). A high-impedance condition occurs at the output of three-state gates that are not enabled or if a wire is inadvertently left unconnected. The four-valued logic truth tables for the **and**, **or**, **xor**, and **not** primitives are shown in Table 4.9. The truth table for the other four gates is the same, except that the outputs are complemented. Note that for the **and** gate, the output is 1 only when both inputs are 1 and the output is 0 if any input is 0. Otherwise, if one input is **x** or **z**, the output is **x**. The output of the **or** gate is 0 if both inputs are 0, is 1 if any input is 1, and is **x** otherwise.

When a primitive gate is listed in a module, we say that it is *instantiated* in the module. In general, component instantiations are statements that reference lower level components in the design, essentially creating unique copies (or *instances*) of those components in the higher level module. Thus, a module that uses a gate in its description is said to

Table 4.9
Truth Table for Predefined Primitive Gates

and	0	1	**x**	**z**
0	0	0	0	0
1	0	1	**x**	**x**
x	0	**x**	**x**	**x**
z	0	**x**	**x**	**x**

or	0	1	**x**	**z**
0	0	1	**x**	**x**
1	1	1	1	1
x	**x**	1	**x**	**x**
z	**x**	1	**x**	**x**

xor	0	1	**x**	**z**
0	0	1	**x**	**x**
1	1	0	**x**	**x**
x	**x**	**x**	**x**	**x**
z	**x**	**x**	**x**	**x**

not	input	output
	0	1
	1	0
	x	**x**
	z	**x**

instantiate the gate. Think of instantiation as the HDL counterpart of placing and connecting parts on a circuit board.

We now present two examples of gate-level modeling. Both examples use identifiers having multiple bit widths, called *vectors*. The syntax specifying a vector includes within square brackets two numbers separated with a colon. The following Verilog statements specify two vectors:

```
output [0: 3] D;
wire   [7: 0] SUM;
```

The first statement declares an output vector *D* with four bits, 0 through 3. The second declares a wire vector *SUM* with eight bits numbered 7 through 0. (*Note*: The first (left-most) number (array index) listed is always the most significant bit of the vector.) The individual bits are specified within square brackets, so *D[2]* specifies bit 2 of *D*. It is also possible to address parts (contiguous bits) of vectors. For example, *SUM[2: 0]* specifies the three least significant bits of vector *SUM*.

HDL Example 4.1 shows the gate-level description of a two-to-four-line decoder. (See Fig. 4.19.) This decoder has two data inputs *A* and *B* and an enable input *E*. The four outputs are specified with the vector *D*. The **wire** declaration is for internal connections. Three **not** gates produce the complement of the inputs, and four **nand** gates provide the outputs for *D*. Remember that *the output is always listed first in the port list of a primitive*, followed by the inputs. This example describes the decoder of Fig. 4.19 and follows the procedures established in Section 3.10. Note that the keywords **not** and **nand** are written only once and do not have to be repeated for each gate, but commas must be inserted at the end of each of the gates in the series, except for the last statement, which must be terminated with a semicolon.

HDL Example 4.1 (Two-to-Four-Line Decoder)

```
// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol E replaced by enable, for clarity.

module decoder_2x4_gates (D, A, B, enable);
 output      [0: 3]          D;
 input                       A, B;
 input                       enable;
 wire                        A_not,B_not, enable_not;

 not
   G1  (A_not, A),
   G2  (B_not, B),
   G3  (enable_not, enable);
 nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);
endmodule
```

Two or more modules can be combined to build a hierarchical description of a design. There are two basic types of design methodologies: top down and bottom up. In a *top-down* design, the top-level block is defined and then the subblocks necessary to build the top-level block are identified. In a *bottom-up* design, the building blocks are first identified and then combined to build the top-level block. Take, for example, the binary adder of Fig. 4.9. It can be considered as a top-block component built with four full-adder blocks, while each full adder is built with two half-adder blocks. In a top-down design, the four-bit adder is defined first, and then the two adders are described. In a bottom-up design, the half adder is defined, then each full adder is constructed, and then the four-bit adder is built from the full adders.

A bottom-up hierarchical description of a four-bit adder is shown in HDL Example 4.2. The half adder is defined by instantiating primitive gates. The next module describes the full adder by instantiating and connecting two half adders. The third module describes the four-bit adder by instantiating and connecting four full adders. Note that the first character of an identifier cannot be a number, but can be an underscore, so the module name *_4bitadder* is valid. An alternative name that is meaningful, but does not require a leading underscore, is *adder_4_bit*. The instantiation is done by using the name of the module that is instantiated together with a new (or the same) set of port names. For example, the half adder *HA1* inside the full adder module is instantiated with ports *S1*, *C1*, *x,* and *y*. This produces a half adder with outputs *S1* and *C1* and inputs *x* and *y*.

HDL Example 4.2 (Ripple-Carry Adder)

```
// Gate-level description of four-bit ripple carry adder
// Description of half adder (Fig. 4.5b)

// module half_adder (S, C, x, y);                    // Verilog 1995 syntax
// output  S, C;
// input   x, y;

module half_adder (output S, C, input x, y);          // Verilog 2001, 2005 syntax
// Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
endmodule

// Description of full adder (Fig. 4.8)               // Verilog 1995 syntax
// module full_adder (S, C, x, y, z);
// output          S, C;
// input           x, y, z;

module full_adder (output S, C, input x, y, z);       // Verilog 2001, 2005 syntax
  wire S1, C1, C2;

// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule

// Description of four-bit adder (Fig. 4.9)        // Verilog 1995 syntax
// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
// output [3: 0]   Sum;
// output          C4;
// input  [3: 0]   A, B;
// input           C0;
// Alternative Verilog 2001, 2005 syntax:

module ripple_carry_4_bit_adder ( output [3: 0] Sum, output C4,
  input [3: 0] A, B, input C0);
  wire          C1, C2, C3;          // Intermediate carries
// Instantiate chain of full adders
full_adder      FA0 (Sum[0], C1, A[0], B[0], C0),
                FA1 (Sum[1], C2, A[1], B[1], C1),
                FA2 (Sum[2], C3, A[2], B[2], C2),
                FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```

HDL Example 4.2 illustrates Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., **output**), type (**reg**), and declaration of a vector range (e.g., [3: 0]) of a port. The first version of the standard (1995) uses separate statements for these declarations.

Note that modules can be instantiated (nested) within other modules, but module declarations cannot be nested; that is, a module definition (declaration) cannot be placed within another module declaration. In other words, a module definition cannot be inserted into the text between the **module** and **endmodule** keywords of another module. The only way one module definition can be incorporated into another module is by instantiating it. Instantiating modules within other modules creates a hierarchical decomposition of a design. A description of a module is said to be a *structural* description if it is composed of instantiations of other modules. Note also that *instance names* must be specified when defined modules are instantiated (such as *FA0* for the first full adder in the third module), but using a name is optional when instantiating primitive gates. Module *ripple_carry_4_bit_adder* is composed of instantiated and interconnected full adders, each of which is itself composed of half adders and some *glue logic*. The top level, or parent module, of the design hierarchy is the module *ripple_carry_4_bit_adder*. Four copies of *full_adder* are its child modules, etc. *C0* is an input of the cell forming the least significant bit of the chain, and *C4* is the output of the cell forming the most significant bit.

Three-State Gates

As mentioned in Section 4.11, a three-state gate has a control input that can place the gate into a high-impedance state. The high-impedance state is symbolized by **z** in Verilog. There are four types of three-state gates, as shown in Fig. 4.32. The **bufif1** gate behaves like a normal buffer if *control* = 1. The output goes to a high-impedance state **z** when *control* = 0. The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when *control* = 1. The two **notif** gates operate in a similar manner, except that the output is the complement of the input when the gate is not in a high-impedance state. The gates are instantiated with the statement

gate name (*output*, *input*, *control*);

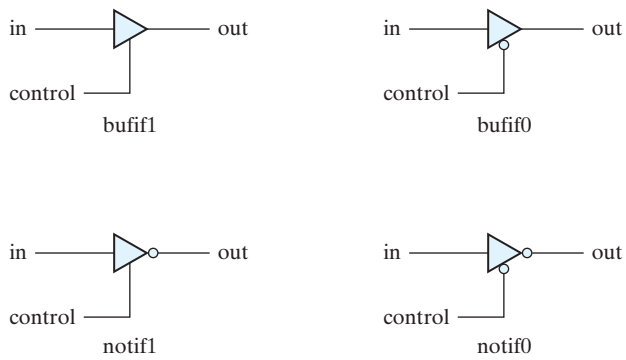

FIGURE 4.32
Three-state gates

The gate name can be that of any 1 of the 4 three-state gates. In simulation, the output can result in 0, 1, **x,** or **z.** Two examples of gate instantiation are

```
bufif1   (OUT, A, control);
notif0   (Y, B, enable);
```

In the first example, input A is transferred to OUT when $control = 1$. OUT goes to **z** when $control = 0$. In the second example, output $Y =$ **z** when $enable = 1$ and output $Y = B'$ when $enable = 0$.

The outputs of three-state gates can be connected together to form a common output line. To identify such a connection, Verilog HDL uses the keyword **tri** (for tristate) to indicate that the output has multiple drivers. As an example, consider the two-to-one-line multiplexer with three-state gates shown in Fig. 4.33.

The HDL description must use a **tri** data type for the output:

```
// Mux with three-state output

module mux_tri (m_out, A, B, select);
  output  m_out;
  input   A, B, select;
  tri     m_out;

  bufif1 (m_out, A, select);
  bufif0 (m_out, B, select);
endmodule
```

The 2 three-state buffers have the same output. In order to show that they have a common connection, it is necessary to declare *m_out* with the keyword **tri.**

Keywords **wire** and **tri** are examples of a set of data types called *nets*, which represent connections between hardware elements. In simulation, their value is determined by a continuous assignment statement or by the device whose output they represent. The word *net* is not a keyword, but represents a class of data types, such as **wire, wor, wand, tri, supply1,** and **supply0.** The **wire** declaration is used most frequently. In fact, if an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a **wire**. The net **wor** models the hardware implementation of the wired-OR configuration (emitter-coupled logic). The **wand** models the wired-AND configuration (open-collector technology; see Fig. 3.26). The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to hardwire an input of a device to either 1 or 0.

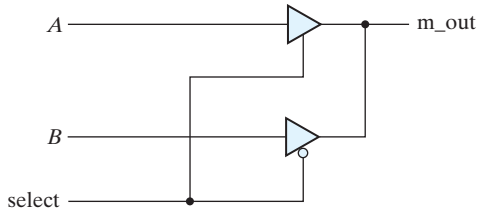

FIGURE 4.33
Two-to-one-line multiplexer with three-state buffers

Dataflow Modeling

Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result. Verilog HDL provides about 30 different operators. Table 4.10 lists some of these operators, their symbols, and the operation that they perform. (A complete list of operators supported by Verilog 2001, 2005 can be found in Table 8.1 in Section 8.2.) It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each. The plus symbol (+) indicates the arithmetic operation of addition; the bitwise logic AND operation (conjunction) uses the symbol &. There are special symbols for bitwise logical OR (disjunction), NOT, and XOR. The equality symbol uses two equals signs (without spaces between them) to distinguish it from the equals sign used with the **assign** statement. The bitwise operators operate bit by bit on a pair of vector operands to produce a vector result. The concatenation operator provides a mechanism for appending multiple operands. For example, two operands with two bits each can be concatenated to form an operand with four bits. The conditional operator acts like a multiplexer and is explained later, in conjunction with HDL Example 4.6.

It should be noted that a bitwise operator (e.g., ~) and its corresponding logical operator (e.g., !) may produce different results, depending on their operand. If the operands are scalar the results will be identical; if the operands are vectors the result will not necessarily match. For example, ~(1010) is (0101), and !(1010) is 0. A binary value is considered to be logically true if it is not 0. In general, use the bitwise operators to describe arithmetic operations and the logical operators to describe logical operations.

Dataflow modeling uses continuous assignments and the keyword **assign.** A continuous assignment is a statement that assigns a value to a net. The data type family *net* is used in Verilog HDL to represent a physical connection between circuit elements. A net

Table 4.10
Some Verilog HDL Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
−	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

is declared explicitly by a net keyword (e.g., **wire**) or by declaring an identifier to be an input port. The logic value associated with a net is determined by what the net is connected to. If the net is connected to an output of a gate, the net is said to be *driven* by the gate, and the logic value of the net is determined by the logic values of the inputs to the gate and the truth table of the gate. If the identifier of a net is the left-hand side of a continuous assignment statement or a procedural assignment statement, the value assigned to the net is specified by a Boolean expression that uses operands and operators. As an example, assuming that the variables were declared, a two-to-one-line multiplexer with scalar data inputs *A* and *B*, select input *S*, and output *Y* is described with the continuous assignment

```
assign Y = (A && S) || (B && S)
```

The relationship between *Y*, *A*, *B*, and *S* is declared by the keyword **assign,** followed by the target output *Y* and an equals sign. Following the equals sign is a Boolean expression. In hardware terms, this assignment would be equivalent to connecting the output of the OR gate to wire *Y*.

The next two examples show the dataflow models of the two previous gate-level examples. The dataflow description of a two-to-four-line decoder with active-low output enable and inverted output is shown in HDL Example 4.3. The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output. The dataflow description of the four-bit adder is shown in HDL Example 4.4. The addition logic is described by a single statement using the operators of addition and concatenation. The plus symbol (+) specifies the binary addition of the four bits of *A* with the four bits of *B* and the one bit of *C_in*. The target output is the *concatenation* of the output carry *C_out* and the four bits of *Sum*. Concatenation of operands is expressed within braces and a comma separating the operands. Thus, *{C_out, Sum}* represents the five-bit result of the addition operation.

HDL Example 4.3 (Dataflow: Two-to-Four Line Decoder)

```
// Dataflow description of two-to-four-line decoder

// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (                    // Verilog 2001, 2005 syntax
  output    [0: 3]      D,
  input                 A, B,
                        enable
);
  assign    D[0] = !((!A) && (!B) && (!enable)),
            D[1] = !(*!A) && B && (!enable)),
            D[2] = !(A && B && (!enable)
            D[3] = !(A && B && (!enable))
endmodule
```

HDL Example 4.4 (Dataflow: Four-Bit Adder)

```
// Dataflow description of four-bit adder
// Verilog 2001, 2005 module port syntax
module binary_adder (
  output [3: 0]          Sum,
  output                 C_out,
  input [3: 0]           A, B,
  input                  C_in
);
  assign {C_out, Sum} = A + B + C_in;
endmodule
```

Dataflow HDL models describe combinational circuits by their *function* rather than by their gate structure. To show how dataflow descriptions facilitate digital design, consider the 4-bit magnitude comparator described in HDL Example 4.5. The module specifies two 4-bit inputs A and B and three outputs. One output (A_lt_B) is logic 1 if A is less than B, a second output (A_gt_B) is logic 1 if A is greater than B, and a third output (A_eq_B) is logic 1 if A is equal to B. Note that equality (identity) is symbolized with two equals signs (= =) to distinguish the operation from that of the assignment operator (=). A Verilog HDL synthesis compiler can accept this module description as input, execute synthesis algorithms, and provide an output netlist and a schematic of a circuit equivalent to the one in Fig. 4.17, all without manual intervention! The designer need not draw the schematic.

HDL Example 4.5 (Dataflow: Four-Bit Comparator)

```
// Dataflow description of a four-bit comparator //V2001, 2005 syntax
module mag_compare
( output                 A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]           A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A = = B);
endmodule
```

The next example uses the conditional operator (? :). This operator takes three operands:

condition ? true-expression : false-expression;

The condition is evaluated. If the result is logic 1, the true expression is evaluated and used to assign a value to the left-hand side of an assignment statement. If the result is

logic 0, the false expression is evaluated. The two conditions together are equivalent to an if–else condition. HDL Example 4.6 describes a two-to-one-line multiplexer using the conditional operator. The continuous assignment

```
assign OUT = select ? A : B;
```

specifies the condition that $OUT = A$ if $select = 1$, else $OUT = B$ if $select = 0$.

HDL Example 4.6 (Dataflow: Two-to-One Multiplexer)

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output    m_out;
  input     A, B;
  input     select;

  assign m_out = (select)? A : B;
endmodule
```

Behavioral Modeling

Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits. Here, we give two simple combinational circuit examples to introduce the subject. Behavioral modeling is presented in more detail in Section 5.6, after the study of sequential circuits.

Behavioral descriptions use the keyword **always**, followed by an optional event control expression and a list of procedural assignment statements. The event control expression specifies when the statements will execute. The target output of a procedural assignment statement must be of the **reg** data type. Contrary to the **wire** data type, whereby the target output of an assignment may be continuously updated, a **reg** data type retains its value until a new value is assigned.

HDL Example 4.7 shows the behavioral description of a two-to-one-line multiplexer. (Compare it with HDL Example 4.6.) Since variable *m_out* is a target output, it must be declared as **reg** data (in addition to the **output** declaration). The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed after the @ symbol. (Note that there is no semicolon (;) at the end of the **always** statement.) In this case, these variables are the input variables *A*, *B*, and *select*. The statements execute if *A*, *B*, or *select* changes value. Note that the keyword **or**, instead of the bitwise logical OR operator "|", is used between variables. The conditional statement **if–else** provides a decision based upon the value of the *select* input. The **if** statement can be written without the equality symbol:

```
if (select) OUT = A;
```

The statement implies that *select* is checked for logic 1.

HDL Example 4.7 (Behavioral: Two-to-One Line Multiplexer)

```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
  output      m_out;
  input       A, B, select;
  reg         m_out;

  always      @(A or B or select)
   if (select == 1) m_out = A;
   else m_out 5 B;
endmodule
```

HDL Example 4.8 describes the function of a four-to-one-line multiplexer. The *select* input is defined as a two-bit vector, and output *y* is declared to have type **reg**. The **always** statement, in this example, has a sequential block enclosed between the keywords **case** and **endcase**. The block is executed whenever any of the inputs listed after the **@** symbol changes in value. The **case** statement is a multiway conditional branch construct. Whenever *in_0, in_1, in_2, in_3* or *select* change, the case expression (*select*) is evaluated and its value compared, from top to bottom, with the values in the list of statements that follow, the so-called **case** items. The statement associated with the first **case** item that matches the **case** expression is executed. In the absence of a match, no statement is executed. Since *select* is a two-bit number, it can be equal to 00, 01, 10, or 11. The **case** items have an implied priority because the list is evaluated from top to bottom.

The list is called a *sensitivity list* (Verilog 2001, 2005) and is equivalent to the *event control expression* (Verilog 1995) formed by "ORing" the signals. Combinational logic is reactive—when an input changes an output may change.

HDL Example 4.8 (Behavioral: Four-to-One Line Multiplexer)

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg m_out,
  input      in_0, in_1, in_2, in_3,
  input [1: 0] select
);
always @ (in_0, in_1, in_2, in_3, select)      // Verilog 2001, 2005 syntax
  case (select)
    2'b00:          m_out = in_0;
    2'b01:          m_out = in_1;
    2'b10:          m_out = in_2;
    2'b11:          m_out = in_3;
  endcase
endmodule
```

Binary numbers in Verilog are specified and interpreted with the letter **b** preceded by a prime. The size of the number is written first and then its value. Thus, 2′*b*01 specifies a two-bit binary number whose value is 01. Numbers are stored as a bit pattern in memory, but they can be referenced in decimal, octal, or hexadecimal formats with the letters ***d′ o′,*** and ***h′,*** respectively. For example, 4′HA = 4′d10 = 4′b1010 and have the same internal representation in a simulator. If the base of the number is not specified, its interpretation defaults to decimal. If the size of the number is not specified, the system assumes that the size of the number is at least 32 bits; if a host simulator has a larger word length—say, 64 bits—the language will use that value to store unsized numbers. The integer data type (keyword **integer**) is stored in a 32-bit representation. The underscore (_) may be inserted in a number to improve readability of the code (e.g., 16′b0101_1110_0101_0011). It has no other effect.

The **case** construct has two important variations: **casex** and **casez**. The first will treat as don't-cares any bits of the **case** expression or the **case** item that have logic value **x** or **z**. The **casez** construct treats as don't-cares only the logic value **z**, for the purpose of detecting a match between the **case** expression and a **case** item.

The list of case items need not be complete. If the list of **case** items does not include all possible bit patterns of the **case** expression, no match can be detected. Unlisted **case** items, i.e., bit patterns that are not explicitly decoded can be treated by using the **default** keyword as the last item in the list of **case** items. The associated statement will execute when no other match is found. This feature is useful, for example, when there are more possible state codes in a sequential machine than are actually used. Having a **default** case item lets the designer map all of the unused states to a desired next state without having to elaborate each individual state, rather than allowing the synthesis tool to arbitrarily assign the next state.

The examples of behavioral descriptions of combinational circuits shown here are simple ones. Behavioral modeling and procedural assignment statements require knowledge of sequential circuits and are covered in more detail in Section 5.6.

Writing a Simple Test Bench

A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit in order to test it and observe its response during simulation. Test benches can be quite complex and lengthy and may take longer to develop than the design that is tested. The results of a test are only as good as the test bench that is used to test a circuit. Care must be taken to write stimuli that will test a circuit thoroughly, exercising all of the operating features that are specified. However, the test benches considered here are relatively simple, since the circuits we want to test implement only combinational logic. The examples are presented to demonstrate some basic features of HDL stimulus modules. Chapter 8 considers test benches in greater depth.

In addition to employing the **always** statement, test benches use the **initial** statement to provide a stimulus to the circuit being tested. We use the term "**always** statement" loosely. Actually, **always** is a Verilog language construct specifying *how* the associated statement is to execute (subject to the event control expression). The **always** statement

executes repeatedly in a loop. The **initial** statement executes only once, starting from simulation time 0, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol #. For example, consider the **initial** block

```
initial
  begin
        A = 0; B = 0;
    #10 A = 1;
    #20 A = 0; B = 1;
  end
```

The block is enclosed between the keywords **begin** and **end.** At time 0, A and B are set to 0. Ten time units later, A is changed to 1. Twenty time units after that (at $t = 30$), A is changed to 0 and B to 1. Inputs specified by a three-bit truth table can be generated with the **initial** block:

```
initial
  begin
    D = 3'b000;
    repeat (7)
    #10 D = D + 3'b001;
  end
```

When the simulator runs, the three-bit vector D is initialized to 000 at time = 0. The keyword **repeat** specifies a looping statement: D is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.

A stimulus module has the following form:

```
module test_module_name;
  // Declare local reg and wire identifiers.
  // Instantiate the design module under test.
  // Specify a stopwatch, using $finish to terminate the simulation.
  // Generate stimulus, using initial and always statements.
  // Display the output response (text or graphics (or both)).
endmodule
```

A test module is written like any other module, but it typically has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type. The outputs of the design module that are displayed for testing are declared in the stimulus module as local **wire** data type. The module under test is then instantiated, using the local identifiers in its port list. Figure 4.34 clarifies this relationship. The stimulus module generates inputs for the design module by declaring local identifiers *t_A* and *t_B* as **reg** type and checks the output of the design unit with the **wire** identifier *t_C*. The local identifiers are then used to instantiate the design module being tested. The simulator associates the (actual) local identifiers within the test bench, *t_A*, *t_B*, and *t_C,* with the formal identifiers of the

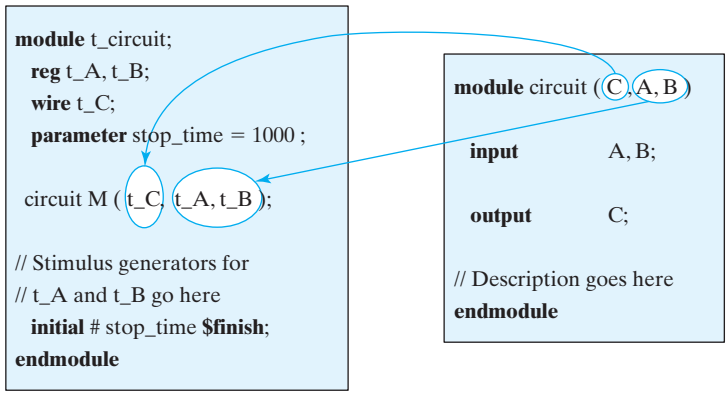

FIGURE 4.34
Interaction between stimulus and design modules

module (*A, B, C*). The association shown here is based on *position* in the port list, which is adequate for the examples that we will consider. The reader should note, however, that Verilog provides a more flexible *name association* mechanism for connecting ports in larger circuits.

The response to the stimulus generated by the **initial** and **always** blocks will appear in text format as standard output and as waveforms (timing diagrams) in simulators having graphical output capability. Numerical outputs are displayed by using Verilog *system tasks*. These are built-in system functions that are recognized by keywords that begin with the symbol **$.** Some of the system tasks that are useful for display are

$display—display a one-time value of variables or strings with an end-of-line return,

$write**—same as **$display, but without going to next line,

$monitor—display variables whenever a value changes during a simulation run,

$time—display the simulation time,

$finish—terminate the simulation.

The syntax for **$display, $write,** and **$monitor** is of the form

Task-name (*format specification, argumentlist*);

The format specification uses the symbol % to specify the radix of the numbers that are displayed and may have a string enclosed in quotes (″). The base may be binary, decimal, hexadecimal, or octal, identified with the symbols %b, %d, %h, and %o, respectively (%B, %D, %H, and %O are valid too). For example, the statement

```
$display ("%d %b %b", C, A, B);
```

specifies the display of *C* in decimal and of *A* and *B* in binary. Note that there are no commas in the format specification, that the format specification and argument list

are separated by a comma, and that the argument list has commas between the variables. An example that specifies a string enclosed in quotes may look like the statement

$display ("time = %0d A = %b", **$time,** A, B);

and will produce the display

time = 3 A = 10 B = 1

where (*time* =), (*A* =), and (*B* =) are part of the string to be displayed. The format specifiers %0d, %b, and %b specify the base for **$time,** *A*, and *B*, respectively. In displaying time values, it is better to use the format %0d instead of %d. This provides a display of the significant digits without the leading spaces that %d will include. (%d will display about 10 leading spaces because time is calculated as a 32-bit number.)

An example of a stimulus module is shown in HDL Example 4.9. The circuit to be tested is the two-to-one-line multiplexer described in Example 4.6. The module *t_mux_2x1_df* has no ports. The inputs for the mux are declared with a **reg** keyword and the outputs with a **wire** keyword. The mux is instantiated with the local variables. The **initial** block specifies a sequence of binary values to be applied during the simulation. The output response is checked with the **$monitor** system task. Every time a variable in its argument changes value, the simulator displays the inputs, output, and time. The result of the simulation is listed under the simulation log in the example. It shows that *m_out* = *A* when *select* = 1 and *m_out* = *B* when *select* = 0 verifying the operation of the multiplexer.

HDL Example 4.9 (Test Bench)

```
// Test bench with stimulus for mux_2x1_df

module t_mux_2x1_df;
  wire        t_mux_out;
  reg         t_A, t_B;
  reg         t_select;
  parameter stop_time = 50;

mux_2x1_df M1 (t_mux_out, t_A, t_B, t_select);      // Instantiation of circuit to be tested

initial # stop_time $finish;

  initial begin                                     // Stimulus generator
          t_select = 1; t_A = 0; t_B = 1;
    #10 t_A = 1; t_B = 0;
    #10 t_select = 0;
    #10 t_A = 0; t_B = 1;
  end

  initial begin                                     // Response monitor
   // $display (″  time  Select  A  B  m_out″);
   // $monitor ($time,, ″ %b  %b  %b  %b ″, t_select, t_A, t_B, t_m_out);
```

```
    $monitor (″time =″, $time,, ″select = %b A = %b B = %b OUT = %b″,
    t_select, t_A, t_B, t_mux_out);
  end
endmodule

// Dataflow description of two-to-one-line multiplexer

// from Example 4.6
module mux_2x1_df (m_out, A, B, select);
  output      m_out;
  input       A, B;
  input       select;

  assign m_out = (select)? A : B;
endmodule
```

```
Simulation log:
select = 1 A = 0 B = 1 OUT = 0 time = 0
select = 1 A = 1 B = 0 OUT = 1 time = 10
select = 0 A = 1 B = 0 OUT = 0 time = 20
select = 0 A = 0 B = 1 OUT = 1 time = 30
```

Logic simulation is a fast and accurate method of verifying that a model of a combinational circuit is correct. There are two types of verification: functional and timing. In *functional* verification, we study the circuit logical operation independently of timing considerations. This can be done by deriving the truth table of the combinational circuit. In *timing* verification, we study the circuit's operation by including the effect of delays through the gates. This can be done by observing the waveforms at the outputs of the gates when they respond to a given input. An example of a circuit with gate delays was presented in Section 3.10 in HDL Example 3.3. We next show an HDL example that produces the truth table of a combinational circuit. A **$monitor** system task displays the output caused by the given stimulus. A commented alternative statement having a **$display** task would create a header that could be used with a **$monitor** statement to eliminate the repetition of names on each line of output.

The analysis of combinational circuits was covered in Section 4.3. A multilevel circuit of a full adder was analyzed, and its truth table was derived by inspection. The gate-level description of this circuit is shown in HDL Example 4.10. The circuit has three inputs, two outputs, and nine gates. The description of the circuit follows the interconnections between the gates according to the schematic diagram of Fig. 4.2. The stimulus for the circuit is listed in the second module. The inputs for simulating the circuit are specified with a three-bit **reg** vector *D*. *D[2]* is equivalent to input *A*, *D[1]* to input *B*, and *D[0]* to input *C*. The outputs of the circuit F_1 and F_2 are declared as **wire.** The complement of *F2* is named *F2_b* to illustrate a common industry practice for designating the complement of a signal (instead of appending *_not*). This procedure

follows the steps outlined in Fig. 4.34. The **repeat** loop provides the seven binary numbers after 000 for the truth table. The result of the simulation generates the output truth table displayed with the example. The truth table listed shows that the circuit is a full adder.

HDL Example 4.10 (Gate-Level Circuit)

```
// Gate-level description of circuit of Fig. 4.2

module Circuit_of_Fig_4_2 (A, B, C, F1, F2);
  input  A, B, C;
  output F1, F2;
  wire   T1, T2, T3, F2_b, E1, E2, E3;
  or   g1 (T1, A, B, C);
  and g2 (T2, A, B, C);
  and g3 (E1, A, B);
  and g4 (E2, A, C);
  and g5 (E3, B, C);
  or   g6 (F2, E1, E2, E3);
  not  g7 (F2_b, F2);
  and g8 (T3, T1, F2_b);
  or   g9 (F1, T2, T3);
endmodule

// Stimulus to analyze the circuit

module test_circuit;
  reg [2: 0] D;
  wire F1, F2;
  Circuit_of_Fig_4_2 (D[2], D[1], D[0], F1, F2);
  initial
   begin
     D = 3'b000;
     repeat (7) #10 D = D 1 1'b1;
   end
  initial
  $monitor ("ABC = %b F1 = %b F2 =%b", D, F1, F2);

endmodule

Simulation log:  ABC = 000 F1 = 0 F2 =0
ABC = 001 F1 = 1 F2 =0 ABC = 010 F1 = 1 F2 =0
ABC = 011 F1 = 0 F2 =1 ABC = 100 F1 = 1 F2 =0
ABC = 101 F1 = 0 F2 =1 ABC = 110 F1 = 0 F2 =1
ABC = 111 F1 = 1 F2 =1
```