2.13 Verilog Models for Multiplexers

A multiplexer is a combinational circuit and can be modeled using concurrent statements only or using always statements. A conditional operator with **assign statement** can be used to model a multiplexer without always statements. A case statement or if-else statement can also be used to make a model for a multiplexer within an always statement.

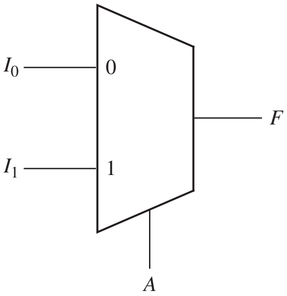
2.13.1 Using Conditional Operator

Figure 2-36 shows a 2-to-1 multiplexer (MUX) with 2 data inputs and one control input. The MUX output is $F = A' \cdot I_0 + A \cdot I_1$. The corresponding Verilog statement is

```
assign F = (~A && I0) || (A && I1);
```

Here, the MUX can be modeled as a single concurrent signal assignment statement. Alternatively, we can represent the MUX by a conditional signal assignment statement as shown in Figure 2-36. This statement executes whenever A, I_0, or I_1 changes. The MUX output is I_0 when $A = 0$; otherwise it is I_1. In the conditional statement, I_0, I_1, and F can be one or more bits.

FIGURE 2-36: 2-to-1 Multiplexer



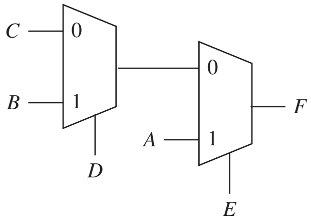
```
// conditional signal assignment
statement
   assign F = (A) ? I1 : I0;
```

The general form of a conditional signal assignment statement is

```
assign signal_name = condition ? expression_T : expression_F;
```

This concurrent statement is executed whenever a change occurs in a signal used in one of the expressions or conditions. If `condition` is true, `signal_name` is set equal to the value of `expression_T`. Otherwise if `condition` is false, `signal_name` is set equal to the value of `expression_F`. Figure 2-37 shows how two cascaded MUXes can be represented by a conditional signal assignment statement. The

FIGURE 2-37: Cascaded 2-to-1 MUXes Using Conditional Assignment



```
assign F = E ? A : ( D ? B : C );
// nested conditional assignment
```

output MUX selects *A* when the condition E is true; otherwise, it selects the output of the first MUX, which is *B* when the condition D is true, or it is *C*.

Figure 2-38 shows a 4-to-1 multiplexer (MUX) with four data inputs and two control inputs, *A* and *B*. The control inputs select which one of the data inputs is transmitted to the output. The logic equation for the 4-to-1 MUX is

$$F = A'B'I_0 + A'B\,I_1 + A\,B'I_2 + A\,B\,I_3$$

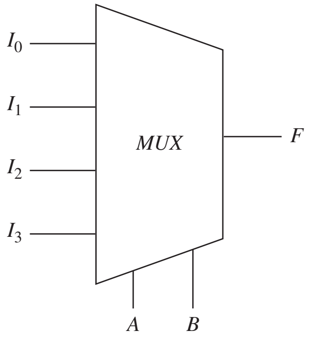
One way to model the MUX is with the Verilog statement

```
assign F = (~A && ~B && I0) || (~A && B && I1) ||
      A && ~B && I2) || (A && B && I3);
```

Another way to model the 4-to-1 MUX is to use a conditional assignment statement:

```
assign F = (A) ? (B ? I3 : I2) : ( B ? I1 : I0 );
```

FIGURE 2-38: 4-to-1 Multiplexer



2.13.2 Using If-else or Case Statement in an Always Block

If a MUX model is used inside an always statement, a concurrent statement cannot be used. The MUX can be modeled using a **case** statement within an always block:

```
always @ (Sel or I0 or I1 or I2 or I3)
  case Sel
    2'b00 : F = I0;
    2'b01 : F = I1;
    2'b10 : F = I2;
    2'b11 : F = I3;
  endcase
```

Since this MUX has four input signals, the selection signal, `Sel` should be a 2-bit signal. The selection signals are represented as `2'b00, 2'b01, 2'b10, and 2'b11` in the form of `<number of bits>'<base><value>`. The `b` represents that the base is binary here. The case statement has the general form:

```
case expression
   choice1 : sequential statements1
   choice2 : sequential statements2
   . . .
   [default : sequential statements]
endcase;
```

The `expression` is evaluated first. If it is equal to `choice1`, then `sequential statements1` are executed; if it is equal to `choice2`, then `sequential statements2` are executed; and so forth. All possible values of the expression must be included in the choices. If all values are not explicitly given, a `default` clause is required in the **`case`** statement. As an alternative, the MUX can also be modeled using an **`if-else`** statement within an always block:

```
always @ (Sel or I0 or I1 or I2 or I3)
begin
  if       (Sel == 2'b00)    F = I0;
  else if  (Sel == 2'b01)    F = I1;
  else if  (Sel == 2'b10)    F = I2;
  else if  (Sel == 2'b11)    F = I3;
end
```

One might notice that combinational circuits can be described using concurrent or sequential statements. Sequential circuits generally require an **always** statement. **Always** statements can be used to make sequential or combinational circuits.

The following are important coding practices while writing synthesizeable Verilog for combinational hardware:

(a) If possible use concurrent assignments (e.g., assign) to design combinational logic.
(b) When procedural assignments (always blocks) are used for combinational logic, use blocking assignments (e.g., "=").
(c) If Verilog 2001 or later is used, instead of specifying contents of sensitivity lists, use always@* to avoid accidental omission of inputs from sensitivity lists. The accidental omission results in incorrect hardware or deviation between simulation and synthesis.

2.14 Modeling Registers and Counters Using Verilog Always Statements

When several flip-flops change state on the same clock edge, statements representing these flip-flops can be placed in the same clocked always statements. Figure 2-39 shows three flip-flops connected as a **cyclic shift register** (or a **rotating shift register**). These flip-flops all change state following the rising edge of the clock. We have assumed a 5 ns propagation delay between the clock edge and the output change. Immediately following the clock edge, the three statements in the always statement execute in sequence with no delay. The new values of the Qs are then scheduled to change after 5 ns. If we omit the delay and replace the sequential statements with

```
Q1 <= Q3;   Q2 <= Q1;   Q3 <= Q2;
```

the operation is basically the same. The three statements execute in sequence in zero time, and then the Qs values change after a delta delay. In both cases, the old values of Q_1, Q_2, and Q_3 are used to compute the new values. This may seem strange

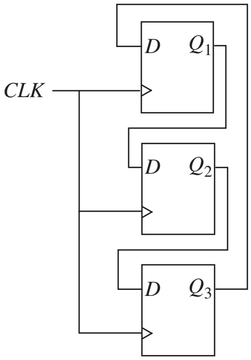


FIGURE 2-39: Cyclic Shift Register

```
always @ (posedge CLK)
begin
  Q1 <= #5 Q3;
  Q2 <= #5 Q1;
  Q3 <= #5 Q2;
end
```

at first, but that is the way the hardware works. At the rising edge of the clock, all of the D inputs are loaded into the flip-flops, but the state change does not occur until after a propagation delay.

The order of the statements is not important when the non-blocking assignment operator "<=" is used. The same result is obtained even if the statements are in reverse order as shown here.

```
always @ (posedge CLK)
begin
  Q3 <= #5 Q2;
  Q2 <= #5 Q1;
  Q1 <= #5 Q3;
end
```

Example 1

What is the hardware obtained if the following code is synthesized?

```
module reg3 (Q1,Q2,Q3,A,CLK);
input   A;
input   CLK;
output  Q1,Q2,Q3;

reg     Q1,Q2,Q3;

always @(posedge CLK)
begin
  Q3 = Q2;  // statement 1
  Q2 = Q1;  // statement 2
  Q1 = A;  // statement 3

end

endmodule
```

Answer: A 3-bit shift register

Explanation: The list of statements executes from top to bottom in order. Note that the blocking operator is used. Therefore, the first statement finishes update before the second statement is executed. Synthesis results in a 3-bit shift register with serial input A, and outputs Q_1, Q_2, and Q_3.

Note: While a register can be modeled using the blocking operator "=" as in this example, it is generally advised to not do so. As mentioned previously, a good coding practice while writing synthesizeable code is to use non-blocking assignments (i.e., "<=") in always blocks intended to create sequential logic, and the blocking operator "=" in always blocks intended to create combinational logic.

Example 2

What is the hardware obtained if the following code is synthesized? Note that this is the same code as in the previous example, but with the statement order inside the always block reversed.

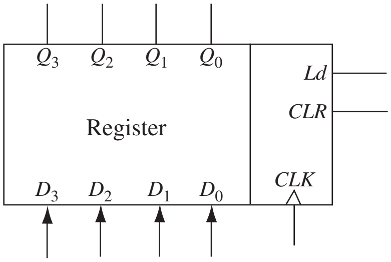
```
module reg31 (Q1,Q2,Q3,A,CLK);
input   A;
input   CLK;
output  Q1,Q2,Q3;

reg     Q1,Q2,Q3;

always @(posedge CLK)
begin

  Q1 = A;  // statement 1
  Q2 = Q1;  // statement 2
  Q3 = Q2;  // statement 3

end

endmodule
```

Answer: A single flip-flop

Explanation: The list of statements executes from top to bottom in order. Note that the blocking operator is used. So the first statement finishes update before the second statement is executed. Q_1 gets the value of the serial input A when statement 1 finishes. In statement 2, the same value propagates to Q_2. In statement 3, the same value propagates to Q_3. In effect, the input A has reached Q_3. Modern synthesis tools will generate a single flip-flop with input A when this code is synthesized. The outputs Q_1, Q_2, and Q_3 can all be connected to the output of the same flip-flop. If the synthesizer does not have good optimization algorithms, it might generate three parallel flip-flops, each with the same input A but with outputs Q_1, Q_2, and Q_3, respectively. As mentioned in the Note to Example 1, it is not a good practice to use the blocking operator "=" in always blocks intended to create sequential logic. If one were to use non-blocking statements, the order of the statements would not have mattered.

Figure 2-40 shows a simple register that can be loaded or cleared on the rising edge of the clock. If *CLR* is set to 1, the register is cleared, and if *Ld* = 1, the *D* inputs are loaded into the register. This register is fully synchronous so that the *Q* outputs change only in response to the clock edge and not in response to a change in *Ld* or *CLR*. In the Verilog code for the register, *Q* and *D* are 4-bit vectors dimensioned [3:>]. Since the register outputs can only change on the rising edge of the clock, *CLR* and *Ld* are not on the sensitivity list. The *CLR* and *Ld* signals are tested after the rising edge of the clock. If *CLR* = *Ld* = 0, no change of *Q* occurs. Since *CLR* is tested before *Ld*, if *CLR* = 1, the **else if** prevents *Ld* from being tested and *CLR* overrides *Ld*.

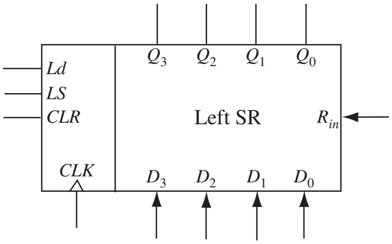
FIGURE 2-40: Register with Synchronous Clear and Load



```
always @ (posedge CLK)
begin
 if (CLR)         Q <= 4'b0000;
 else if (Ld)     Q <= D;
end
```

Next, we will model a left shift register using a Verilog **always** statement. The register in Figure 2-40 is similar to that in Figure 2-41, except that we have added a left shift control input (*LS*). When *LS* is 1, the contents of the register are shifted left and the right-most bit is set equal to *Rin*. The shifting is accomplished by taking the rightmost 3 bits of *Q*, `Q[2:0]`, and concatenating them with *Rin*. For example, if *Q* = 1101 and *Rin* = 0, then `{Q[2:0], Rin}` = 1010, and this value is loaded back into the *Q* register on the rising edge of *CLK*. The code implies that if *CLR* = *Ld* = *LS* = 0, then *Q* remains unchanged.

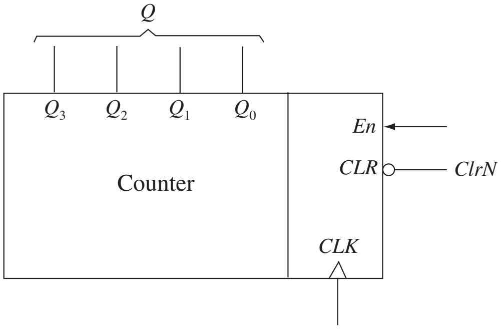
FIGURE 2-41: Left Shift Register with Synchronous Clear and Load



```
always @ (posedge CLK)
begin
 if (CLR)          Q <= 4'b0000;
 else if (Ld)      Q <= D;
 else if (LS)      Q <= {Q[2:0], Rin};
end
```

Figure 2-42 shows a simple synchronous counter. On the rising edge of the clock, the counter is cleared when *ClrN* = , and it is incremented when *ClrN* = *En* = 1. In this example, the signal *Q* represents the 4-bit value stored in the counter. The signal

FIGURE 2-42: Verilog Code for a Simple Synchronous Counter

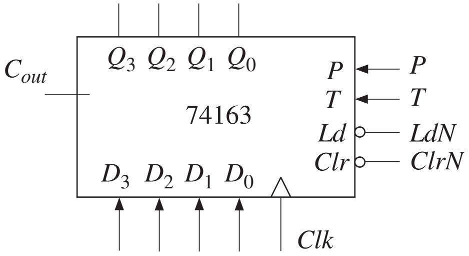


```
reg Q[3:0];

always @ (posedge CLK)
begin
 if (~ClrN)     Q <= 4'b0000;
 else if (En)   Q <= Q + 1;
end
```

Q is declared to be of type **reg** with length of 4 bits. Then the statement `Q <= Q+1;` increments the counter. When the counter is in state 1111, the next increment takes it back to state 0000.

Now, let us create a Verilog model for a standard MSI counter, the 74163. It is a 4-bit fully synchronous binary counter, which is available in both TTL and CMOS logic families. Although rarely used in new designs at present, it represents a general type of counter that is found in many CAD libraries. In addition to performing the counting function, it can be cleared or loaded in parallel. All operations are synchronized by the clock, and all state changes take place following the rising edge of the clock input. A block diagram of the counter is provided in Figure 2-43.

FIGURE 2-43: 74163 Counter Operation



Control Signals			Next State				
ClrN	*LdN*	*PT*	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

This counter has four control inputs—*ClrN*, *LdN*, *P*, and *T*. Both *P* and *T* are used to enable the counting function. While *P* is an actual enable signal to the 4-bit generic counter, *T* is used for a carry connection signal when cascading multiple counters. Operation of the counter is as follows:

1. If *ClrN* = 0, all flip-flops are set to 0 following the rising clock edge.
2. If *ClrN* = 1 and *LdN* = 0, the *D* inputs are transferred (loaded) in parallel to the flip-flops following the rising clock edge.
3. If *ClrN* = *LdN* = 1 and *P* = *T* = 1, the count is enabled and the counter state will be incremented by 1 following the rising clock edge.

If *T* = 1, the counter generates a carry (C_{out}) in state 15; consequently

$$C_{\text{out}} = Q_3\,Q_2\,Q_1\,Q_0\,T$$

The truth table in Figure 2-43 summarizes the operation of the counter. Note that *ClrN* overrides the load and count functions in the sense that when *ClrN* = 0, clearing occurs regardless of the values of *LdN*, *P*, and *T*. Similarly *LdN* overrides the count function. The *ClrN* input on the 74163 is referred to as a *synchronous* clear input because it clears the counter in synchronization with the clock, and no clearing can occur if no clock pulse is present.

The Verilog description of the counter is shown in Figure 2-44. *Q* represents the four flip-flops that comprise the counter. The counter output, *Qout*, changes whenever *Q* changes. The carry output is computed whenever *Q* or *T* changes. The always statement will be executed only at the rising edge of *Clk*. Since clear overrides load and count, the first **if** statement tests *ClrN* first. Since load overrides count, *LdN* is tested next. Finally, the counter is incremented if both *P* and *T* are 1.

FIGURE 2-44: 74163 Counter Model

```
// 74163 FULLY SYNCHRONOUS COUNTER

module c74163(LdN, ClrN, P, T, Clk, D, Cout, Qout);
input        LdN;
input        ClrN;
input        P;
input        T;
input        Clk;
input [3:0]  D;
output       Cout;
output [3:0] Qout;

reg [3:0]    Q;

assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;

always @(posedge Clk)
begin
  if (~ClrN)        Q <= 4'b0000;
  else if (~LdN)    Q <= D;
  else if (P & T)   Q <= Q + 1;
end

endmodule
```

To test the counter, we have cascaded two 74163s to form an 8-bit counter (Figure 2-45). When the counter on the right is in state 1111 and *T1* = 1, *Carry1* = 1. Then for the left counter, *PT* = 1 if *P* = 1. If *PT* = 1, on the next clock the right counter is incremented to 0000 at the same time the left counter

FIGURE 2-45: Two 74163 Counters Cascaded to Form an 8-Bit Counter

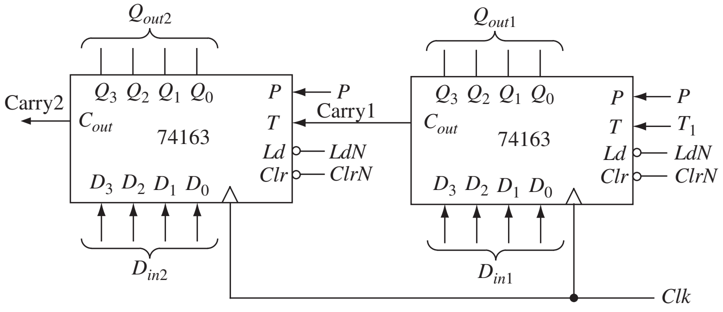


FIGURE 2-46: Verilog for 8-Bit Counter Using 4-Bit Counter Modules

```
// 8-Bit counter using two 74163 counters using the model in Fig 2-44

module eight_bit_counter(ClrN, LdN, P, T1, Clk, Din1, Din2, Count, Carry2);
input         ClrN;
input         LdN;
input         P;
input         T1;
input         Clk;
input [3:0]  Din1;
input [3:0]  Din2;
output [7:0]  Count;
output        Carry2;

wire          Carry1;
wire [3:0]    Qout1;
wire [3:0]    Qout2;

c74163 ct1(LdN, ClrN, P, T1, Clk, Din1, Carry1, Qout1); // instance 1 (right)
c74163 ct2(LdN, ClrN, P, Carry1, Clk, Din2, Carry2, Qout2); //instance 2 (left)

assign Count 5 {Qout2, Qout1};

endmodule
```

is incremented. Figure 2-46 shows the Verilog code for the 8-bit counter. In this code we have used the c74163 model from Figure 2-44 as a component and have instantiated two copies of it. The module c74163 should be available for the module eight_bit_counter by defining it in same file or another file. The instantiation for the lower four bits is done using the statement

```
c74163 ct1(LdN, ClrN, P, T1, Clk, Din1, Carry1, Qout1);
```

where `c74163` is the previously defined component's module name and `ct1` is the instance name. The inputs and outputs are mapped to `Din1` and `Qout1` and other control signals. VHDL uses a `port-map` keyword to accomplish this kind of instantiation whereas in Verilog, no keyword is required. This type of instantiation is required for **structural modeling** interconnecting previously defined modules. These instances should be outside always statements.

Let us now synthesize the Verilog code for a left shift register from Figure 2-41. Before synthesis is started, we must specify a target device (e.g., a particular FPGA or CPLD) so that the synthesizer knows what components are available. Let us assume that the target is a CPLD or FPGA that has D flip-flops with clock enable (D-CE flip-flops). Q and D are of length four bits. Because updates to Q follow on rising edge of the CLK, this suggests that Q must be a register composed of four flip-flops, which we will label Q_3, Q_2, Q_1, and Q_0. Since the flip-flops can change state when Clr, Ld, or Ls is 1, we connect the clock enables to an OR gate whose output is $Clr + Ld + Ls$. Then we connect gates to the D inputs to select the data to be loaded into the flip-flops. If $Clr =$ and $Ld = 1$, D is loaded into the register on the rising clock edge. If $Clr = Ld = 0$ and $Ls = 1$, then Q_2 is loaded into Q_3, Q_1 is loaded into Q_2, and so forth. Figure 2-47 shows the logic circuit for the first two flip-flops. If $Clr = 1$, the D flip-flop inputs are 0 and the register is cleared.

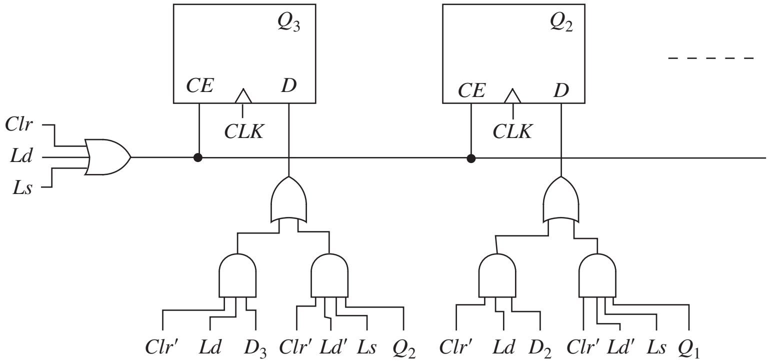


FIGURE 2-47: Synthesis of Verilog Code for Left Shift Register from Figure 2-41

A Verilog synthesizer cannot synthesize delays. Clauses of the form "**#** time" will be ignored by most synthesizers, but some synthesizers may even require that the clauses be removed.

Similarly, initial blocks are usually ignored by synthesis tools. Also, although initial values for signals may be specified in port and signal declarations, these initial values are usually ignored by the synthesizer. A reset signal should be provided if the hardware must be set to a specific initial state. Otherwise, the initial state of the hardware may be unknown and the hardware may malfunction. The only exception to this is that some synthesis tools for FPGAs may utilize the initial blocks and initial values in the initial bit stream that is downloaded into the FPGA. But for synthesis into custom hardware, initial values and initial blocks are typically ignored.

Avoiding Unwanted Latches

Verilog signals retain their current values until they are changed. This can result in the creation of unwanted latches when the Verilog code is synthesized. For example, in a combinational always block created using the statements

```
always @ (Sel or I0 or I1 or I2)
begin
 if      (Sel == 2'b00)  F = I0;
 else if (Sel == 2'b01)  F = I1;
 else if (Sel == 2'b10)  F = I2;
end
```

there would be latches to hold the value of F when Sel changes to `2'b11`. Someone probably wrote this code intending a MUX. Circuits with latches are not combinational hardware any more. The latch creates a variety of timing problems and unexpected behavior. Instead of being 1 bit wide, if `F` is 8 bits wide, eight latches would be created.

One can avoid unwanted latches by assigning a value to combinational signals in every possible execution path in an always block intended to create combinational hardware. Hence one should include an **else** clause in every **if** statement or explicitly include all possible cases of the inputs. For example, the code

```
always @ (Sel or I0 or I1 or I2 or I3)
begin
 if        (Sel == 2'b00)  F = I0;
 else if   (Sel == 2'b01)  F = I1;
 else if   (Sel == 2'b10)  F = I2;
 else if   (Sel == 2'b11)  F = 0;
end
```

would not create any latches but would create a MUX. For **if** then **else** statements and **case** statements, it is important to have all cases specified.

Another method to avoid latches is by initializing at the beginning of the always statement. For instance, the following code is latch free, even though only three of the four possible cases are specified.

```
always @ (Sel or I0 or I1 or I2 or I3)
F = 0;
begin
 if        (Sel == 2'b00)  F = I0;
 else if   (Sel == 2'b01)  F = I1;
 else if   (Sel == 2'b10)  F = I2;
end
```

The following are important coding practices while writing synthesizable Verilog for sequential hardware:

(a) Use an edge-triggered clock in the sensitivity list using the **posedge** or **negedge** keywords.
(b) Use non-blocking assignments—that is, "<=" inside always blocks although it is possible to get sequential hardware by certain uses of the blocking "=" operator.
(c) Do not mix blocking and non-blocking statements in an always block.
(d) Do not make assignments to the same variable from more than one always block. This is not a compile-time error and hence may go unnoticed.
(e) Avoid unwanted latches by assigning a value to combinational output signals in every possible execution path in the always block. This can be done by

i. including else clauses for if statements,
ii. specifying all cases for case statements or have a `default` clause at the end, or
iii. unconditionally assigning default values to all combinational output signals at the beginning of the always block.

2.15 Behavioral and Structural Verilog

Any circuit or device can be represented in multiple forms of abstraction. Consider the different representations for a NAND gate, as illustrated in Figure 2-48. When one hears the term NAND, different designers, depending on their domain of used, and the compiler gives you that. In a similar way, the synthesizer does not actually have to translate any structural descriptions that the designer wrote; it simply gives the hardware that the designer specified in a structural fashion. Some optimizing tools are capable of optimizing imperfect circuits that you might have specified. In general, you have more control over the generated circuitry when you use structural coding. However, it takes a lot more effort to produce a structural model, because one needs to perform state assignments, derive next state equations, and so forth. **Time to market** is an important criterion for success in the IC market; hence, designers often use behavioral design in order to achieve quick time to market. Additionally, CAD tools have matured significantly during the past decade, and most modern synthesis tools are capable of producing efficient hardware for arithmetic and logic circuits.

2.16 Constants

Verilog provides constants in addition to variables and nets. Verilog provides three different ways to define constant values, one of which is using **`define** as in

```
`define   constant_name  constant_value
```

The `define is one of the compiler directives in Verilog and is used to define a number or an expression for a meaningful string. The ` in `define is called *grave accent* (ASCII 0x60). It is different from the character (‘), which is the *apostrophe* character (ASCII 0x27). More on compiler directives can be found in Chapter 8.

The `define compiler directive replaces 'constant_name with constant_value. For example:

```
`define wordsize 16
 reg [1:`wordsize] data;
```

causes the string wordsize to be replaced by 16. It then shows how data is declared to be a reg of width wordsize.

Another method to create constants is to use the parameter keyword as follows:

```
parameter constant_name = constant_value;
```

For example,

```
parameter msb = 15; // defines msb as a constant value 15
parameter [31:0] decim = 1'b1; // value converted to 32 bits
```

Another method to make constants is using localparam.

```
localparam constant_name = constant_value;
```

The localparam is similar to the parameter, but it cannot be directly changed. The localparam can be used to define constants that should not be changed.

Verilog can define constant values in a module using the parameter. The parameter can be used to customize the module instances. Typical uses of parameters are to specify delays and width of variables.

Verilog HDL parameters do not belong to either the variable or the net group. Parameters are not variables; they are constants. Since parameters represent constants, it is illegal to modify their values at run time. However, module parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows customization of module instances. The parameter values can be changed at module instantiation or by using **defparam** statement. These are described in Chapter 8.

2.17 Arrays

A key feature of VLSI circuits is the repeated use of similar structures. Arrays in Verilog can be used while modeling the repetition. Digital systems often use memory arrays. Verilog arrays can be used to create memory arrays and specify the values to be stored in these arrays. In order to use an array in Verilog, we must declare the array upper and lower bound. There are two positions to declare the array bounds:

In one option, the array bounds are declared between the variable type (`reg or net`) and the variable name, and the array bound means the number of bits for the declared variable. If the array bound is defined as [7:0] as shown in the following example,

```
reg   [7:0]   eight_bit_register;
```

the register variable `eight_bit_register` can store one byte (eight bits) of information. The 8-bit register can be initialized to hold the value `00000001` using the following statement:

```
eight_bit_register = 8'b00000001;
```

As a second option, array bounds can be declared after the name of the array. In the declaration that follows, `rega` is an array of n 1-bit registers while `regb` is a single n-bit register.

```
reg  rega [1:n]; // This is an array of n 1-bit registers
reg [1:n] regb; // This is an n-bit register
```

We can define multiple 8-bit registers in one array declaration. In this case, additional upper and lower bound(s) must be declared after the name of the array. In the example that follows, 16 registers are declared; each register can store one-byte (8-bit) vector information.

```
reg  [7:0]  eight_bit_register_array [15:0];
```

The foregoing declaration means that each of the 16 variables in the array can have 8-bit vector information. This array can be initialized as follows:

```
eight_bit_register_array[15] = 8'b00001100;
eight_bit_register_array[14] = 8'b00000000;

              . . . . . . . .
eight_bit_register_array[1]  = 8'b11001100;
eight_bit_register_array[0]  = 8'b00010001;
```

Arrays can be created of various data types. Arrays of wires and integers can be declared as follows:

```
wire wire_array[5:0]; // declares an array of 6 wires
integer inta[1:64]; // declares an array of 64 integer values
```

Matrices

Multidimensional array types may also be defined with two or more dimensions. The following example defines a 2-dimensional array variable in an initial statement, which is a matrix of integers with four rows and three columns with 8-bit elements:

```
reg  [7:0]  matrixA [0:3][0:2] = { { 1,  2,  3},
                                   { 4,  5,  6},
                                   { 7,  8,  9},
                                   {10, 11, 12}};
```

The array element *matrixA[3][1]* references the element in the fourth row and second column, which has a value of 11.

Look-Up Table Method Using Arrays and Parameters

The array construct together with parameter can be used to create look-up tables which can be used to create combinational circuits using the ROM or Look-up Table (LUT) method.

Example

Parity bits are often used in digital communication for error detection and correction. The simplest of these involve transmitting one additional bit with the data, a parity bit. Use Verilog arrays to represent a parity generator that generates a 5-bit-odd-parity generation for a 4-bit input number using the look-up table (LUT) method.

Answer: The input word is a 4-bit binary number. A 5-bit odd-parity representation will contain exactly an odd number of 1s in the output word. This can be accomplished by the ROM or LUT method using a look-up table of size 16 entries × 5 bits. The look-up table is indicated in Figure 2-60.

FIGURE 2-60: LUT Contents for a Parity Code Generator

Input (LUT Address)				Output (LUT Data)				
A	B	C	D	P	Q	R	S	T
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	1	1
0	1	0	0	0	1	0	0	0
0	1	0	1	0	1	0	1	1
0	1	1	0	0	1	1	0	1
0	1	1	1	0	1	1	1	0
1	0	0	0	1	0	0	0	0
1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	0	1	1	0	0	1
1	1	0	1	1	1	0	1	0
1	1	1	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1

The Verilog code for the parity generator is illustrated in Figure 2-61. The first 4 bits of the output are identical to the input. Hence, instead of storing all 5 bits of the output, one might store only the parity bit and then concatenate it to the input bits. The parameter construct is used to define the ParityBit which is 1-bit constant, and the 4-bit input data and 1-bit ParityBit are concatenated to make a parity code as an output.

FIGURE 2-61: Parity Code Generator Using the LUT Method

```
module parity_gen(X, Y);
input  [3:0] X;
output [4:0] Y;

wire         ParityBit;

parameter [0:15] OT = {1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b1,
1'b1, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1};

assign ParityBit = OT[X];
assign Y = {X, ParityBit};

endmodule
```

2.18 Loops in Verilog

Often, one has systems where some activity is happening in a repetitive fashion. Verilog loop statements can be used to express this behavior. A loop statement is a sequential statement. Verilog has several kinds of loop statements including **for** loops and **while** loops. There is also a **repeat** loop.

Forever Loop (Infinite Loop)

Infinite loops are undesirable in common computer languages, but they can be useful in hardware modeling where a device works continuously and continues to work until the power is off. Below is an example for a forever loop:

```
begin
  clk = 1'b0;
  forever #10 clk = ~clk;
end
```

For Loop

One way to augment the basic loop is to use the **for** loop, where the number of invocations of the loop can be specified. Syntax is similar to C language except that begin and end are used instead of { } for more than one statement. Note that, unlike C language, we do not have the operators ++ and − in Verilog. Instead of using those operators as in C language, we need to use its full operational statement, `i = i + 1`. The general form of a **for** loop is as follows:

```
for (initial_statement; expression; incremental_statement)
begin

   sequential statement(s);

end
```

The following example is of initializing an array variable, `eight_bit_register_array`, using a **for** loop.

```
reg  [7:0]  eight_bit_register_array [15:0];

for (i=0; i<16; i=i+1)
begin

  register A[i] = 8'b00000000;

end
```

One could use this type of loop in behavioral models. The following excerpt models a 4-bit adder. The loop index (i) will be initialized to 0 when the **for** loop is entered, and the sequential statements will be executed. Execution will be repeated for $i = 1$, $i = 2$, and $i = 3$; then the loop will terminate. The carry out from one iteration (C_{out}) is copied to the carry in (C_{in}) before the end of the loop. Since variables are used for the sum and carry bits, the update of carry out happens instantaneously. Code like this often appears in Verilog tasks and functions (described in Chapter 8).

```
for (i=0; i<4; i=i+1)
begin

 Cout = (A[i] && B[i]) || (A[i] && Cin) || (B[i] && Cin);
   sum[i] = A[i] ^ B[i] ^ Cin;
   Cin = Cout;

end
```

You could also use the **for** loop to create multiple copies of a basic cell. When the foregoing code is synthesized, the synthesizer typically provides four copies of a 1-bit adder connected in a ripple carry fashion. If you actually desire to create just one copy of the cell and simply use it multiple times as a serial adder, then you have to design a sequential circuit. The loop construct will not synthesize into that behavior.

While Loop

As in **while** loops in most languages, a condition is tested before each iteration. The loop is terminated if the condition is false. The general form of a while loop is

```
while condition
begin

  sequential statements;

end
```

A while loop is used primarily for simulation.

Figure 2-62 illustrates a while loop that models a down counter. We use the **while** statement to continue the decrementing process until the stop is encountered or the counter reaches 0. The counter is decremented on every rising edge of `clk` until either the `count` is 0 or `stop` is 1.

FIGURE 2-62: Use of While Loop

```
`define MAX 100

  module counter_100;

  integer count;

  initial begin
    count = 0;
    while (count < `MAX) begin
      count = count + 1;
    end // while

    $display("number = %d ", count);
  end // initial begin

  endmodule
```

Repeat Loop

The repeat loop repeats the sequential statement(s) for specified times. The number of repetitions is set by a constant value or a logical expression.

```
repeat( 8 )
  begin

     x = x + 1;
     y = y + 2;

  end
```

2.19 Testing a Verilog Model

Once a Verilog model for a system has been made, the next step is to test it. A model has to be tested and validated before it can be successfully used.

A test bench is a piece of Verilog code that can provide input combinations to test a Verilog model for the system under test. It provides stimuli to the system or circuit under test. Test benches are frequently used during simulation to provide sequences of inputs to the circuit or Verilog model under test. Figure 2-63 shows a test bench for testing the 4-bit binary adder that we created earlier in this chapter. The adder we are testing will be treated as a component and will be embedded in the test-bench program. The signals generated within the test bench are interfaced to the adder, as shown in Figure 2-63. The module on the right side is the module under test.

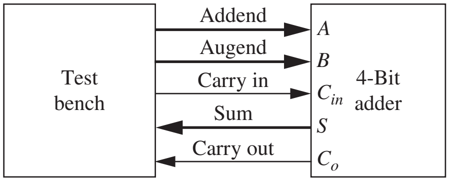


FIGURE 2-63: Interfacing of Signals While Using a Test Bench to Test a 4-Bit Adder

Figure 2-64 shows an example of a test bench for a 4-bit adder without using assertion monitors.

Assertion statements available in languages such as System Verilog provide an easy mechanism to specify properties that the design must adhere to and to verify them. Verilog does not have assertion statements. Hence in this test bench, we simply provide test inputs and check whether the module outputs match the expected outputs. It may be noticed that the test bench module does not have external inputs and outputs; hence, the port list is empty in the module declaration in statement 1. One can see use of Verilog parameter construct in statement 2.

FIGURE 2-64: Test Bench for 4-Bit Adder

```
module TestAdder_v2; //Statement 1

parameter  N = 11;  //Statement 2
reg  [3:0] addend;
reg  [3:0] augend;
reg        cin;
wire [3:0] sum;
wire       cout;

reg [3:0] addend_array[1:N];
reg [1:N] cin_array;
reg [3:0] augend_array[1:N];
reg [3:0] sum_array[1:N];
reg [1:N] cout_array;

initial
begin
  //initialization of addend_array
  addend_array[1] = 4'b0111;
  addend_array[2] = 4'b1101;
  addend_array[3] = 4'b0101;
  addend_array[4] = 4'b1101;
  addend_array[5] = 4'b0111;
  addend_array[6] = 4'b1000;
  addend_array[7] = 4'b0111;
  addend_array[8] = 4'b1000;
  addend_array[9] = 4'b0000;
  addend_array[10] = 4'b1111;
  addend_array[11] = 4'b0000;
  //initialization of cin_array
  cin_array[1] = 1’b0;
  cin_array[2] = 1’b0;
  cin_array[3] = 1’b0;
  cin_array[4] = 1’b0;
  cin_array[5] = 1’b1;
  cin_array[6] = 1’b0;
  cin_array[7] = 1’b0;
  cin_array[8] = 1’b0;
  cin_array[9] = 1’b1;
  cin_array[10] = 1’b1;
  cin_array[11] = 1’b0;

  //initialization of augend_array
  augend_array[1] = 4’b0101;
  augend_array[2] = 4’b0101;
  augend_array[3] = 4’b1101;
  augend_array[4] = 4’b1101;
  augend_array[5] = 4’b0111;
  augend_array[6] = 4’b0111;
  augend_array[7] = 4’b1000;
  augend_array[8] = 4’b1000;
  augend_array[9] = 4’b1101;
  augend_array[10] = 4’b1111;
  augend_array[11] = 4’b0000;

  //initialization of sum_array (expected sum outputs)
  sum_array[1] = 4’b1100;
  sum_array[2] = 4’b0010;
  sum_array[3] = 4’b0010;
  sum_array[4] = 4’b1010;
  sum_array[5] = 4’b1111;
  sum_array[6] = 4’b1111;
  sum_array[7] = 4’b1111;
  sum_array[8] = 4’b0000;
  sum_array[9] = 4’b1110;
  sum_array[10] = 4’b1111;
  sum_array[11] = 4’b0000;

  //initialization of cout_array (expected carry output)
  cout_array[1] = 1'b0;
  cout_array[2] = 1'b1;
  cout_array[3] = 1'b1;
  cout_array[4] = 1'b1;
  cout_array[5] = 1'b0;
  cout_array[6] = 1'b0;
  cout_array[7] = 1'b0;
  cout_array[8] = 1'b1;
  cout_array[9] = 1'b0;
  cout_array[10] = 1'b1;
  cout_array[11] = 1'b0;
end
```

```
integer i;

always
begin
  for(i = 1 ; i <= N ; i = i + 1)
  begin
    $display(i);
    addend <= addend_array[i];//apply an addend test vector
    augend <= augend_array[i];//apply an augend test vector
    cin <= cin_array[i];//apply a carry in

    #(40);//adder expected to take 40 time units

    if(!(sum == sum_array[i] & cout == cout_array[i]))
    begin
      $write("ERROR: ");
      $display("Wrong Answer ");
    end
    else begin
      $display("Correct!!");
    end
  end
  $display(“Test Finished”);
end

Adder4 add1(addend, augend, cin, sum, cout); //module under test instantiated

endmodule
```

Input and output vectors are hard-coded into the test bench. An exhaustive test of the adder module requires 512 tests, since there are 9 input bits (4 addend; 4 augend and 1 carryin). We have a random set of 11 tests.

The **$display** and **$write** statements are used to print test results. The two sets of tasks are identical except that **$display** automatically adds a new line character to the end of its output, whereas the **$write** task does not. In Verilog, a name following the **$** is interpreted as a *system task* or a *system function*. The dollar sign (**$**) essentially introduces a language construct that enables the development of user-defined system tasks and functions. These system tasks are not design semantics, but refer to simulator functionality.

The test module has to be instantiated into the test bench outside the always statement. In this example, Adder4 is the adder module from Figure 2-12. It is instantiated into the test bench and the inputs mapped to test vectors. Use of appropriate delays becomes important in test benches. A **#(40)** delay is specified after applying a set of inputs so that the simulation model has time to evaluate the results. This delay value is computed based on the fact that each bit of the ripple carry adder takes 10 ns in the model in Figure 2-10. Use of incorrect values of delays may give misleading test results. An improperly created test bench may incorrectly indicate that a properly working circuit is faulty.

Similar to **$display**, there are several other Verilog statements to observe outputs—for example, **$strobe**, and **$monitor**. The format string is like that in C/C++ and may contain format characters, as in the following:

```
$display ("Value is %d", para1);
$strobe ("Value is %d", para1);
$monitor ("Value is %d", para1);
```

The **$display** and **$strobe** display once every time they are executed (i.e., every time the statement is encountered), whereas **$monitor** displays every time one of its parameters changes. The difference between **$display** and **$strobe** is that **$strobe** displays the parameters at the very end of the current simulation time unit whereas **$display** outputs them exactly where it is executed.

2.20 A Few Things to Remember

Verilog has different kinds of signal assignments and constructs. Which construct is to be used depends on the purpose of the Verilog model being created. Verilog is typically written for the following three reasons:

i. to design hardware (i.e., to model and synthesize hardware)
ii. to model hardware (i.e., to create simulation models that are not necessarily synthesizeable)
iii. to verify hardware (i.e., to test designs)

The following are illegal in all models:

```
assign a <= b;      // statement 1
assign a <= #10 b; // statement 2
assign a = #10 b;  // statement 3
```

The assign statement is to be used with '=', not with '<='. Hence the first two statements are illegal. Statement 3 uses delayed assignment, which cannot be done with the assign statement. No delay can be specified on the right side of a continuous assign statement. Hence statement 3 is illegal.

The following are legal.

```
assign a = b;      \\ Statement 4
assign #10 a = b;  \\ Statement 5
```

Statement 4 is a concurrent statement. It executes concurrently to all other concurrent assign statements and always blocks. Statement 5 behaves with the inertial type of delay.

How we use assign, blocking, and non-blocking concepts depends on what we are trying to write Verilog for. Some strategies can be followed to result in appropriate models for each purpose.

1. **To design hardware (i.e., to model and synthesize hardware)** This is a major use of Verilog, and the following guidelines are important

(a) Do not use initial blocks. Initial blocks are usually ignored during synthesis, except in some FPGA synthesis tools.

CHAPTER 8

Additional Topics in Verilog

Up to this point, we have described the basic features of Verilog and how they can be used in the digital system design process. In this chapter, we describe additional features of Verilog that illustrate its power and flexibility. Verilog functions and tasks are presented. Several additional features such as user-defined primitives, generate statements, compiler directives, built-in primitives, file I/O, and others are presented. A simple memory model is presented to illustrate the use of tristate signals.

8.1 Verilog Functions

A key feature of VLSI circuits is the repeated use of similar structures. Verilog provides functions and tasks to easily express repeated invocation of the same functionality or the repeated use of structures. These functions are described in this section. A Verilog function is similar to a Verilog task, which will be described later. They have small differences: a function cannot drive more than one output, nor can a function contain delays.

The functions can be defined in the module that they will be used in. The functions can also be defined in separate files, and the compile directive **`include** will be used to include the function in the file (compiler directives are explained in Section 8.12). The functions should be executed in zero time delay, which means that the functions cannot include timing delay information. The functions can have any number of inputs but can return only a single output.

A function returns a value by assigning the value to the function name. The function's return value can be a single bit or multiple bits. The variables declared within the function are local variables, but the functions can also use global variables when no local variables are used. When local variables are used, basically output is assigned only at the end of function execution. The functions can call other functions but cannot call tasks. Functions are frequently used to do type conversions.

A function executes a sequential algorithm and returns a single value to the calling program. When the following function is called, it returns a value equal to the input rotated one position to the right:

```
function [7:0] rotate_right;
  input [7:0] reg_address;

  begin
    rotate_right = reg_address >> 1;
  end
endfunction
```

A function call can be used anywhere that an expression can be used. For example, if $A = 10010101$, the statement

```
B <= rotate_right(A);
```

would set *B* equal to 11001010 and leave *A* unchanged.

The general form of a function declaration is

```
function <range or type> function-name
  input [declarations]
<declarations>    // reg, parameter, integer, etc.

  begin
  sequential statements
  end

endfunction
```

The general form of a function call is

```
function_name(input-argument-list)
```

The number and type of parameters on the `input-argument-list` must match the input [declaration] in the function declaration. The parameters are treated as input values and cannot be changed during the execution of the function.

Example

Write a Verilog function for generating an even parity bit for a 4-bit number. The input is a 4-bit number and the output is a code word that contains the data and the parity bit.

The answer is shown in Figure 8-1. The function name and the final output to be returned from the function must be the same.

FIGURE 8-1: Parity Generation Using a Function

```
// Function example code without a loop
// This function takes a 4-bit vector
// It returns a 5-bit code with even parity

function [4:0] parity;
  input [3:0] A;
  reg temp_parity;

  begin
    temp_parity = A[0] ^ A[1] ^ A[2] ^ A[3];
    parity = {A, temp_parity};
  end
endfunction
```

If parity circuits are used in several parts in a system, one could call the function each time it is desired. The function can be called as follows:

```
module function_test(Z);
  output reg [4:0] Z;
  reg [3:0] INP;
  initial
  begin
    INP = 4'b0101;
    Z = parity(INP);
  end

endmodule
```

Figure 8-2 illustrates a function using a **for** loop. In Figure 8-2, the loop index *i* will be initialized to 0 when the **for** loop is entered, and the sequential statements will be executed. Execution will be repeated for $i = 1$, $i = 2$, and $i = 3$; then the loop will terminate.

FIGURE 8-2: Add Function

```
// This function adds two 4-bit vectors and a carry.
// Illustrates function creation and use of loop.
// It returns a 5-bit sum.

function [4:0] add4;
  input [3:0] A;
  input [3:0] B;
  input       cin;

  reg   [4:0] sum;
  reg         cout;

  begin
    integer i;
    for (i=0; i<=3; i=i+1)
    begin
      cout = (A[i] & B[i]) | (A[i] & cin) | (B[i] & cin);
      sum[i] = A[i] ^ B[i] ^ cin;
      cin = cout;
  end

   sum[4] = cout;
   add4 = sum;
  end
endfunction
```

The function given in Figure 8-2 adds two 4-bit vectors plus a carry and returns a 5-bit vector as the sum. The function name is *add4*; the input arguments are *A*, *B*, and *carry*; and the add4 will be returned. The local variables C_{out} and *sum* are defined to hold intermediate values during the calculation. When the function is called, C_{in} will be initialized to the value of the carry. The **for** loop adds the bits of *A* and *B* serially in the same manner as a serial adder.

The first time through the loop, C_{out} and *sum[0]* are computed using *A[0]*, *B[0]*, and C_{in}. Then the C_{in} value is updated to the new C_{out} value, and execution of the loop is repeated. During the second time through the loop, C_{out} and *sum[1]* are computed using *A[1]*, *B[1]*, and the new C_{in}. After four times through the loop, all values of *sum[i]* have been computed and *sum* is returned. The function call is of the form

```
add4(A, B, carry)
```

A and *B* may be replaced with any expressions that evaluate to a return value with dimensions 3 `downto` 0, and *carry* may be replaced with any expression that evaluates to a bit. For example, the statement

```
Z <= add4(X, ~Y, 1);
```

calls the function *add4*. Parameters *A*, *B*, and *carry* are set equal to the values of *X*, ~*Y*, and 1, respectively. *X* and *Y* must be multiple bits dimensioned 3:0. The function computes

$$Sum = A + B + carry = X + \sim Y + 1$$

and returns this value. Since $\sim Y + 1$ equals the 2's complement of *Y*, the computation is equivalent to subtracting by adding the 2's complement. If we ignore the carry stored in $Z[4]$, the result is $Z[3:0] = X - Y$.

Figure 8-3 illustrates the function to compute square of numbers. The function as well as the function call is illustrated. In the illustrated call to the function, the number is 4 bits wide.

FIGURE 8-3: A Function to Compute Squares and Its Call

```
module test_squares (CLK);

   input CLK;
   reg[3:0] FN;
   reg[7:0] answer;

   function [7:0] squares;
      input[3:0] Number;

      begin
        squares = Number * Number;
      end
   endfunction

   initial
   begin
```

```
      FN = 4'b0011;
   end

   always @(posedge CLK)
   begin
     answer = squares(FN);
   end
endmodule
```

A function executes in zero simulation time. Functions must not contain any timing control statements or delay events. Functions must have at least one input argument. Functions cannot have **output** or **inout** arguments.

Functions can be recursive. Recursive functions must be declared as **automatic**, which causes distinct memory to be allocated each time a function is called. Otherwise, if a function is called recursively, each call will overwrite the memory allocated in the previous call.

8.2 Verilog Tasks

Tasks facilitate decomposition of Verilog code into modules. Unlike functions, which return only a single value implicitly with the function name, tasks can return any number of values. The form of a task declaration is

```
task task_name
  input [declarations]
  output [declarations]

 <declarations>     // reg, parameter, integer, etc.

begin
  sequential statements
end task_name;
```

The `formal-parameter-list` specifies the inputs and outputs to the task and their types. A task call is a sequential or concurrent statement of the form

```
task_name(actual-parameter-list);
```

Unlike functions, a task can be executed in non-zero simulation time. Tasks may contain delay, event, or timing control statements. Tasks can have zero or more arguments of type input, output, or inout. Tasks do not return with a value, but they can pass multiple values through **output** and **inout** arguments.

As an example, we will write a task *Addvec*, which will add two *4*-bit data and a carry and return a *4*-bit sum and a carry. We will use a task call of the form

```
Addvec(A, B, Cin, Sum, Cout);
```

where *A*, *B*, and *Sum* are *4*-bit data and C_{in} and C_{out} are 1-bit data.

Figure 8-4 gives the task definition. *Add1*, *Add2*, and C_{in} are input parameters, and *sum* and C_{out} are output parameters. The addition algorithm is essentially the

FIGURE 8-4: Task for Adding Multiple Bits

```
// This task adds two 4-bit data and a carry and
// returns an n-bit sum and a carry. Add1 and Add2 are assumed
// to be of the same length and dimensioned 3 downto 0.

    task Addvec;
      input [3:0] Add1;
      input [3:0] Add2;
      input Cin;
      output [3:0] sum;
      output cout;

      reg C;

      begin
         C = Cin;
         integer i;
         for(i = 0; i <= 4; i = i + 1)
            begin
               sum[i] = Add1[i] ^ Add2[i] ^ C ;
               C = (Add1[i] & Add2[i]) | (Add1[i] & C) | (Add2[i] & C);
            end
         cout = C ;
      end
    endtask
```

same as the one used in the *add4* function. *C* must be a variable, since the new value of *C* is needed each time through the loop; hence it is declared as **reg**. After *4* times through the loop, all 4 bits of *sum* have been computed. It is desirable not to mix blocking and non-blocking statements in tasks.

The tasks can be defined in the module that the functions will be used in. The tasks can also be defined in separate files, and the compile directive ‘**include should be used (‘)** (described in Section 8.12) will be used to include the task in the file. The variables declared within the task are local to the task. When no local variables are used, global variables will be used for input and output. When only local variables are used within the task, the variables from the last execution within the task will be passed to the caller.

Functions:
At least one input arguments, but no output or inout arguments
Returns a single value by assigning the value to the function name
Can call other functions, but cannot call tasks
Cannot embed delays, wait statements or any time-controlled statement
Executes in zero time
Can be recursive
Cannot contain non-blocking assignment or procedural continuous assignments

Tasks:
Any number of input, output or inout arguments
Outputs need not use task name
Can call other functions or tasks
May contain time-controlled statements
Task can recursively call itself

8.3 Multivalued Logic and Signal Resolution

In previous chapters, we have mostly used two-valued bit logic in our Verilog code. In order to represent tristate buffers and buses, it is necessary to be able to represent a third value, *Z*, which represents the high-impedance state. It is also at times necessary to have a fourth value, *X*, to represent an unknown state. This unknown state may occur if the initial value of a signal is unknown, or if a signal is simultaneously driven to two conflicting values, such as 0 and 1. If the input to a gate is *Z*, the gate output may assume an unknown value, *X*.

8.3.1 A 4-Valued Logic System

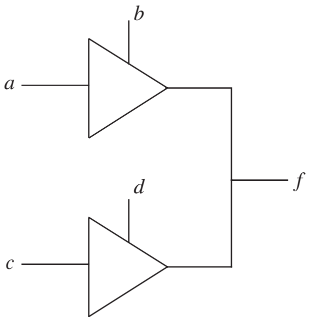
Signals in a 4-valued logic can assume the four values: *X*, 0, 1, and *Z*, where each of the symbols represent the following:

• *X*	Unknown
• 0	0
• 1	1
• *Z*	High impedance

The high impedance state is used for modeling tristate buffers and buses. This unknown state can be used if the initial value of a signal is unknown, or if a signal is simultaneously driven to two conflicting values, such as 0 and 1. Verilog uses the 4-valued logic system by default.

Let us model tristate buffers using the 4-valued logic. Figure 8-5 shows two tristate buffers with their outputs tied together, and Figure 8-6 shows the

FIGURE 8-5: Tristate Buffers with Active-High Output Enable



corresponding Verilog representation in two different ways. Data type X01Z, which can have the four values X, 0, 1, and Z, is assumed. The tristate buffers have an active-high output enable, so that when $b = 1$ and $d = 0$, $f = a$; when $b = 0$ and $d = 1$, $f = c$; and when $b = d = 0$, the f output assumes the high-Z state. If $b = d = 1$, an output conflict can occur. Tristate buffers are used for bus management whereby only one active-high output should be enabled to avoid the output conflict.

FIGURE 8-6: Verilog Code for Tristated Buffers

```
module t_buff_exmpl (a, b, c, d, f);

   input a;
   input b;
   input c;
   input d;
   output f;
   reg f;

   always @(a or b)
   begin : buff1
      if (b == 1'b1)
         f = a ;
      else
         f = 1'bz ; //"drive" the output high Z when not enabled
   end

   always @(c or d)
   begin : buff2
      if (d == 1'b1)
         f = c ;
      else
         f = 1'bz ; //"drive" the output high Z when not enabled
   end

endmodule
```

(a) Tristate module with always statements

```
module t_buff_exmpl2 (a, b, c, d, f);
  input a;
   input b;
   input c;
   input d;
   output f;

   assign f = b ? a: 1'bz ;
   assign f = d ? c: 1'bz ;

endmodule
```

(b) Tristate module with assign statements

The operation of a tristate bus with the 4-valued logic, is specified by the following table:

	X	0	1	*Z*
X	*X*	*X*	*X*	*X*
0	*X*	0	*X*	0
1	*X*	*X*	1	1
Z	*X*	0	1	*Z*

This table gives the resolved value of a signal for each pair of input values: *Z* resolved with any value returns that value, *X* resolved with any value returns *X*, and 0 resolved with 1 returns *X*. If individual wires `s(0), s(1), s(2)` were to change as indicated in the following table at the times indicated, signal `R` in the last column shows the result of the resolution

Time	s(0)	s(1)	s(2)	R
0	*Z*	*Z*	*Z*	*Z*
2	0	*Z*	*Z*	0
4	0	1	*Z*	*X*
6	*Z*	1	*Z*	1
8	*Z*	1	1	1
10	*Z*	1	0	*X*

AND and OR functions for the 4-valued logic may be defined using the following tables:

AND	*X*	0	1	*Z*
X	*X*	0	*X*	*X*
0	0	0	0	0
1	*X*	0	1	*X*
Z	*X*	0	*X*	*X*

OR	*X*	0	1	*Z*
X	*X*	*X*	1	*X*
0	*X*	0	1	*X*
1	1	1	1	1
Z	*X*	*X*	1	*X*

The first table corresponds to the way an AND gate with 4-valued inputs would work. If one of the AND gate inputs is 0, the output is always 0. If both inputs are 1, the output is 1. In all other cases, the output is unknown (*X*), since a high-Z gate input may act like either a 0 or a 1. For an OR gate, if one of the inputs is 1, the output is always 1. If both inputs are 0, the output is 0. In all other cases, the output is *X*.

8.4 Built-in Primitives

The focus of this book is on behavioral-level modeling, and hence we focused on Verilog constructs that allow that. However, Verilog allows modeling at switch-level details and has several predefined primitives for that. It also has predefined gate-level primitives with drive strength among other things. There are 14 predefined logic gate primitives and 12 predefined switch primitives to provide the gate- and switch-level modeling facility. Modeling at the gate and switch level has several advantages:

(i) Synthesis tools can easily map it to the desired circuitry since gates provide a very close one-to-one mapping between the intended circuit and the model.

(ii) There are primitives such as the bidirectional transfer gate that cannot be otherwise modeled using continuous assignments.

The Verilog module in Figure 2-7 is defined in Figure 8-7 using built-in primitives. The **and** and **or** are built-in primitives with one output and multiple inputs. The output terminal must be listed first followed by inputs as in

```
and (out, in_1, in_2, ..., in_n);
```

FIGURE 8-7: Verilog Gate Module Using Built-In Primitives

```
module two_gates (A, B, D, E);      // Figure 2-7 shows the same module using
   output E;                        // concurrent statements instead of
   input A, B, D;                   // built-in primitives

   wire C;

   or    (E,C,D);                   // output port first followed by inputs
   and   (C,A,B);                   // output port first

endmodule
```

Verilog also provides built-in primitives for tristate gates. The **bufif0** is a non-inverting buffer primitive with active-low control input while **bufif1** has active-high control input. The **notif0** and **notif1** are inverting buffers with active-low and active-high controls, respectively. These primitives can support multiple outputs. The outputs must be listed first followed by input and finally the tristate control input. An array of four inverting tristate buffers can be created as shown in Figure 8-8.

FIGURE 8-8: Verilog Array of Tristate Buffers Using Built-In Primitives

```
module tri_driver (in, out, tri_en);
   input [3:0] in;
   output [3:0] out;
   input tri_en;

   bufif0 buf_array[3:0] (out, in, tri_en); // array of three-state buffers

endmodule
```

The statement

```
bufif0 buf_array[3:0] (out, in, tri_en);   // instance name is
                                           // indexed here
```

in Figure 8-8 uses `buf_array[3:0]` as instance name. Notice that the instance name is indexed here. This statement could be replaced using

```
bufif0 buf_array3 (out[3], in[3], tri_en); // instance name not
                                           // indexed
bufif0 buf_array2 (out[2], in[2], tri_en);
bufif0 buf_array1 (out[1], in[1], tri_en);
bufif0 buf_array0 (out[0], in[0], tri_en);
```

where the instance names are not indexed. The instance names are not signal names.

The circuit in Figure 8-5, presented using the behavioral model in Figure 8-6, can be written using Verilog built-in primitives as in Figure 8-9.

FIGURE 8-9: Verilog Tristate Buffer Module Using Built-In Primitives

```
module tri_buffer (a,b,c,d,f);
   input a,b,c,d;
   output f;

   bufif1 buf_one (f,a,b); //output first followed by inputs, control last.
   bufif1 buf_two (f,c,d);

endmodule
```

Table 8.1 lists the built-in primitives in Verilog. The first 12 are gate-level primitives, and the rest are switch-level primitives. Switch-level models are not discusseded in this book, but they are useful if low-level models of designs are to be built.

TABLE 8-1: Built-In Gate and Switch Primitives in Verilog

Built-in Primitive Type	Primitives
n-input gates	**and, nand, nor, or, xnor, xor**
n-output gates	**buf, not**
Three-state gates	**bufif0, bufif1, notif0, notif1**
Pull gates	**pulldown, pullup**
MOS switches	**cmos, nmos, pmos, rcmos, rnmos, rpmos**
Bidirectional Switches	**rtran, rtranif0, rtranif1, tran, transif0, tranif1**

The built-in primitives can have an optional delay specification. A delay specification can contain up to three delay values, depending on the gate type. For a three-delay specification,

(i) the first delay refers to the transition to the rise delay (i.e., transition to 1),

(ii) the second delay refers to the transition to the fall delay (i.e., transition to 0), and

(iii) the third delay refers to the transition to the high-impedance value (i.e., turn-off).

If only one delay is specified, it is rise delay. If there are only two delays specified, they are rise delay and fall delay. If turn-off delay must be specified, all three delays must be specified. The **pullup** and **pulldown** instance declarations must not include delay specifications. The following are examples of built-in primitives with one, two, and three delays:

```
and #(10) a1 (out, in1, in2);            // only one delay
                                         // (so rise delay=10)
and #(10,12) a2 (out, in1, in2);         // rise delay=10 and
                                         // fall delay=12
bufif0 #(10,12,11) b3 (out, in, ctrl);   // rise, fall, and
                                         // turn-off delays
```

8.5 User-Defined Primitives

The predefined gate primitives in Verilog can be augmented with new primitive elements called user-defined primitives (UDPs). UDPs define the functionality of the primitive in truth table or state table form. Once these primitives are specified by the user, instances of these new UDPs can be created in exactly the same manner as built-in gate primitives are instantiated. While the built-in primitives are only combinational, UDPs can be combinational or sequential.

A *combinational UDP* uses the value of its inputs to determine the next value of its output. A *sequential UDP* uses the value of its inputs and the current value of its output to determine the value of its output. Sequential UDPs provide a way to model sequential circuits such as flip-flops and latches. A sequential UDP can model both level-sensitive and edge-sensitive behavior.

A UDP can have multiple input ports but has exactly one output port. Bidirectional inout ports are not permitted on UDPs. All ports of a UDP must be scalar—that is, vector ports are not permitted. Each UDP port can be in one of three states: 0, 1, or X. The tristate or high-impedance state value Z is not supported. If Z values are passed to UDP inputs, they shall be treated the same as X values. In sequential UDPs, the output always has the same value as the internal state.

A UDP begins with the keyword **primitive**, followed by the name of the UDP. The functionality of the primitive is defined with a truth table or state table, starting with the keyword **table** and ending with the keyword **endtable**. The UDP definition then ends with the keyword **endprimitive**. The truth table for a UDP consists of a section of columns, one for each input followed by a colon and finally the output column. The multiplexer we defined in Figure 2-36 using continuous assign statements is redefined as a UDP in Figure 8-10.

FIGURE 8-10: User-Defined Primitive (UDP) for a 2-to-1 Multiplexer

```
primitive mux1 (F, A, I0, I1);
  output F;
  input A, I0, I1; //A is the select input

table
  // A   I0  I1       F
     0    1   0   :   1   ;
```

```
        0   1   1   :   1   ;
        0   1   x   :   1   ;
        0   0   0   :   0   ;
        0   0   1   :   0   ;
        0   0   x   :   0   ;
        1   0   1   :   1   ;
        1   1   1   :   1   ;
        1   x   1   :   1   ;
        1   0   0   :   0   ;
        1   1   0   :   0   ;
        1   x   0   :   0   ;
        x   0   0   :   0   ;
        x   1   1   :   1   ;
endtable

endprimitive
```

The first entry in the truth table in Figure 8-10 can be explained as follows: when A equals 0, I0 equals 1, and I1 equals 0, then output F equals 1. The input combination 0xx (A=0, I0=x, I1=x) is not specified. If this combination occurs during simulation, the value of output port F will become x. Each row of the table in the UDP is terminated by a semicolon.

The multiplexer model can also be specified more concisely in a UDP by using ?. The ? means that the signal listed with ? can take the values of 0,1, or x. Using ? the multiplexer can be defined as in Figure 8-11.

FIGURE 8-11: User-Defined Primitive (UDP) for a 2-to-1 Multiplexer Using?

```
primitive mux2 (F, A, I0, I1);
  output F;
  input A, I0, I1;

table
  // A   I0  I1      F
        0   1   ?   :   1   ;    // ? can equal 0, 1, or x
        0   0   ?   :   0   ;
        1   ?   1   :   1   ;
        1   ?   0   :   0   ;
        x   0   0   :   0   ;
        x   1   1   :   1   ;
endtable

endprimitive
```

UDPs are instantiated just as are built-in primitives. For instance, the preceding multiplexer can be instantiated by using the statement

```
mux1 (outF, Sel, in1, in2)
```

where outF, Sel, in1, and in2 are the names of the output, select, data input1, and data input2 signals.

As an example of a sequential UDP, we present the description of a D flip-flop in Figure 8-12. In a sequential UDP, the output must be defined as a **reg**. Edge-sensitive behavior can be represented in tabular form by listing the value before and after the edge. For instance, (01) means a positive (rising) edge and (10) means a negative (falling) edge. In the sequential UDP, there is an additional colon separating the inputs to the present state in the state table.

FIGURE 8-12: User-Defined Primitive (UDP) for a D Flip-Flop

```
primitive DFF (Q, CLK, D);
  output Q;
  input CLK, D;

  reg Q;

table
  // CLK,  D,    Q,  Q+
    (01)  0   : ? : 0  ; //rising edge with input 0
    (01)  1   : ? : 1  ; //rising edge with input 1
    (0?)  1   : 1 : 1  ; //Present state 1, either rising edge or steady clock
    (?0)  ?   : ? : -  ; //Falling edge or steady clock, no change in output
     ?  (??) : ? : -  ; //Steady clock, ignore inputs, no change in output
endtable

endprimitive
```

The (01) in the first line of the table indicates a rising edge clock. The '-' in the output column means that the output should not change for any of the circumstances covered by that line. For instance, the line

```
(?0) ? ? : - ;
```

means that if there is a falling edge or steady clock, whether the input and present state are 0,1, or x, the output must not change. This line actually represents 27 possibilities, because each ? represents three possibilities. If this line was omitted in the code, the simulator would yield an x output in these situations. It is important to make the truth table as unambiguous as possible by specifying all possible cases. The last line of code clarifies further that under steady clock (i.e., no clock edges), the flip-flop must ignore all inputs.

```
? (??) : ? : - ;
```

The rows in the primitive truth table do not need to be in order. Hence the row order

```
0 1 0 : 1 ;
0 0 0 : 0 ;
```

is acceptable. A simulator scans the truth table from top to bottom. If level-sensitive behavior such as asynchronous set and reset are in a table along with edge-sensitive behavior for data, the level-sensitive behavior should be listed before the edge-sensitive behavior.

```
   input a;
   input b;
   output c;

   reg c;

   wire nand_value;

   assign nand_value = ~(a & b) ;

   always @ (nand_value)
   begin
     if (nand_value == 1'b1)
       #(Trise + 3 * load) c = 1'b1;
     else
       #(Tfall + 3 * load) c = 1'b0;
   end

endmodule

module NAND2_test (in1, in2, in3, in4, out1, out2);
   input in1;
   input in2;
   input in3;
   input in4;
   output out1;
   output out2;

   NAND2 #(2, 1, 2) U1 (in1, in2, out1);
   NAND2 U2 (in3, in4, out2);
endmodule
```

The module *NAND2_test* tests the NAND2 component. The parameter declaration in the module specifies default values for *Trise*, *Tfall*, and *load*. When *U1* is instantiated, the parameter map specifies different values for *Trise*, *Tfall*, and *load*. When *U2* is instantiated, no parameter map is included, so the default values are used. Another way to instantiate *U1* is by using **defparam** as follows:

```
defparam U1.Trise = 2;
defparam U1.Tfall = 1;
defparam U1.load = 2;
NAND2 U1 (in1,in2, out1);
NAND2 U2 (in3, in4, out2);
```

8.9 Named Association

Up to this point, we have used *positional association* in the port maps and parameter maps that are part of an instantiation statement. For example, assume that the module declaration for a full adder is

```
module FullAdder (Cout, Sum, X, Y, Cin);

output Cout;
output Sum;
input X;
input Y;
input Cin;

.........

endmodule
```

The statement

```
FullAdder FA0 (Co[0], S[0], A[0], B[0], Ci[0]);
```

creates a full adder and connects *A[0]* to the *X* input of the adder, *B[0]* to the *Y* input, Ci[0] to the C_{in} input, Co[0] to the C_{out} output, and *S[0]* to the *Sum* output of the adder. The first signal in the port map is associated with the first signal in the module declaration, the second signal with the second signal, and so on.

As an alternative, we can use *named association*, in which each signal in the port map is explicitly associated with a signal in the port of the module declaration. For example, the statement

```
FullAdder FA0 (.Sum(S[0]), .Cout(Co[0]), .X(A[0]), .Y(B[0]),
.Cin(Ci[0]));
```

makes the same connections as the previous instantiation statement—that is, *Sum* connects to *S[0]*, C_{out} connects to *Co[0]*, *X* connects to *A[0]*, and so on. When named association is used, the order in which the connections are listed is not important, and any port signals not listed are left unconnected. Use of named association makes code easier to read, and it offers more flexibility in the order in which signals are listed.

When named association is used with a parameter map, any unassociated parameter assumes its default value. For example, if we replace the statement in Figure 8-18 labeled U1 with the following:

```
NAND2 #(.load(3), .Trise(4)) U1 (in1, in2, out1);
```

Tfall would assume its default value of 2 ns.

8.10 Generate Statements

In Chapter 2, we instantiated four full adders and interconnected them to form a 4-bit adder. Specifying the port maps for each instance of the full adder would become very tedious if the adder had 8 or more bits. When an iterative array of identical operations or module instance is required, the **generate** statement provides an easy way of instantiating these components. The example of Figure 8-19 shows how a **generate** statement can be used to instantiate four 1-bit full adders to create a 4-bit adder. A 5-bit vector is used to represent the carries, with C_{in} the same

FIGURE 8-19: Adder4 Using Generate Statement

```
module Adder4 (A, B, Ci, S, Co);
   input[3:0] A; //inputs
   input[3:0] B;
   input Ci;
   output[3:0] S; //outputs
   output Co;

   wire[4:0] C;

   assign C[0] = Ci ;

   genvar i;
   generate
   for (i=0; i<4; i=i+1)
     begin: gen_loop
       FullAdder FA (A[i], B[i], C[i], C[i+1], S[i]);
     end
   endgenerate

   assign Co = C[4] ;

endmodule

module FullAdder (X, Y, Cin, Cout, Sum);
   input X; //inputs
   input Y;
   input Cin;
   output Cout; //outputs
   output Sum;

   assign #10 Sum = X ^ Y ^ Cin ;
   assign #10 Cout = (X & Y) | (X & Cin) | (Y & Cin) ;
endmodule
```

as $C(0)$ and C_{out} the same as $C(4)$. The **for** loop generates four copies of the full adder, each with the appropriate **port map** to specify the interconnections between the adders.

Another example where the **generate** statement would have been very useful is the array multiplier. The Verilog code for the array multiplier (Chapter 4) made repeated use of port map statements in order to instantiate each module instance. They could have been replaced with **generate** statements.

In the preceding example, we used a **generate** statement of the form

```
genvar gen_variable1,…;
generate
for (  for_loop_condition with gen_variables )
  concurrent statement(s)
endgenerate
```

At compile time, a set of `concurrent statement(s)` is generated for each value of the identifier in the given range. In Figure 8-19, one concurrent statement—a module instance instantiation statement—is used. The statement

```
FullAdder FA (A[i], B[i], C[i], C[i+1], S[i]);
```

inside the **generate** clause creates the effect of the following four statements:

```
FullAdder FA (A[0], B[0], C[0], C[1], S[0]);
FullAdder FA (A[1], B[1], C[1], C[2], S[1]);
FullAdder FA (A[2], B[2], C[2], C[3], S[2]);
FullAdder FA (A[3], B[3], C[3], C[4], S[3]);
```

A **generate** statement itself is defined to be a concurrent statement, so nested generate statements are allowed.

Conditional Generate

A **generate** statement with an **if** clause may be used to conditionally generate a set of concurrent statement(s). This type of **generate** statement has the form

```
generate
if condition
  concurrent statement(s)
endgenerate
```

In this case, the concurrent statements(s) are generated at compile time only if the condition is true. Conditional compilation can also be accomplished using the **`ifdef** compiler directive, which is explained in Section 8-12. Conditional compilation is useful for

(i) Selectively including behavioral, structural, or switch-level models as desired.

(ii) Selectively including different timing or structural information.

(iii) Selectively including different stimuli for different runs under different scenarios.

(iv) Selectively adapting the module functionality to similar but different needs from different customers.

Figure 8-20 illustrates the use of conditional compilation using a **generate** statement with an **if** clause. An N-bit left-shift register is created if *Lshift* is true, using the statement

```
generate
  if(Lshift)
    assign shifter = {Q[N - 1:1], Shiftin} ;
  else
    assign shifter = {Shiftin, Q[N:2]} ;
endgenerate
```

If *Lshift* is false, a right-shift register is generated using another conditional **generate** statement. The example also shows how parameters and **generate** statements can be used together. It illustrates the use of parameters to write a Verilog model with parameters so that the size and function can be changed when it is instantiated. It also shows another example of named association.

FIGURE 8-20: Shift Register Using Conditional Compilation

```
module shift_reg (D, Qout, CLK, Ld, Sh, Shiftin);
   parameter N = 4;
   parameter Lshift = 1;

   output[N:1] Qout;
   input[N:1] D;
   input CLK;
   input Ld;
   input Sh;
   input Shiftin;

   reg[N:1] Q;
   wire[N:1] shifter;

   assign Qout = Q ;

   generate
     if(Lshift)
       assign shifter = {Q[N - 1:1], Shiftin} ; //left shift register
     else
       assign shifter = {Shiftin, Q[N:2]} ; //right shift register
   endgenerate

   always @(posedge CLK)
   begin
     if (Ld == 1'b1)
       begin
         Q <= D ;
       end
     else if (Sh == 1'b1)
       begin
         Q <= shifter ;
       end
   end
endmodule
```

8.11 System Functions

In addition to tasks and functions that the user can create, Verilog has tasks and functions at the system level. System tasks and functions in Verilog start with the **$** sign. System tasks that we have used so far include **$display, $monitor,** and **$write**. The file I/O functions tasks discussed in Section 8.13 are also system tasks. These system functions are not for synthesis, however; they are intended for convenience during simulation.

Systems tasks mainly include display tasks for outputting text or data during simulation, file I/O tasks, and simulation control tasks such as **$finish** and **$stop.** We describe a few of the system tasks/functions in this section.

8.11.1 Display Tasks

Display tasks are very useful during simulation to check outputs. There are several of them, and there are variations for displaying data in binary, hex, or octal formats (e.g., **$displayb, $displayh, $displayo).** The major tasks are the following:

$display	Immediately outputs text or data with new line.
$write	Immediately outputs text/data without new line.
$strobe	Outputs text or data at the end of the current simulation step.
$monitor	Displays text or data for every event on signal.

8.11.2 File I/O Tasks

File I/O tasks are described in detail in Section 8.13.

8.11.3 Simulation Control Tasks

There are two system tasks to control the simulation: **$stop** and **$finish**. The **$stop** task is a system function to temporarily suspend simulation for interaction. **$finish** is a system task to come out of simulation. The **$finish** system task simply makes the simulator exit and pass control back to the host operating system. Both these tasks take an optional expression argument (0, 1, or 2) that determines what type of diagnostic message is printed.

8.11.4 Simulation Time Functions

It can be beneficial to access simulation time. The following system functions provide access to current simulation time:

$time	Returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it.
$stime	Returns an unsigned integer that is a 32-bit time, scaled to the time-scale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low-order 32 bits of the current simulation time are returned.
$realtime	Returns a real number time that, like **$time**, is scaled to the time unit of the module that invoked it.

The following statement can be used to print simulation time:

```
$monitor($time);
```

We can assign it to other variables as follows:

```
time simtime;     // time is one of the variable data types
simtime = $time;  // Assign current simulation time to variable
                  // simtime
```

8.11.5 Conversion Functions

There are also system functions to perform conversions between data types.

$signed() and $unsigned()

Two system functions are used to handle type casting on expressions: **$signed()** and **$unsigned()**. These functions evaluate the input expression and return a value with the same size and value of the input expression and the type defined by the function.

```
reg [3:0] regA;
reg signed [7:0] regB;

regA = $unsigned(-4);          // regA = 4'b1100
regB = $signed (8′b11111100);  // regB = -4
```

$realtobits and $bitstoreal

The system functions **$realtobits** and **$bitstoreal** are useful for type conversions. Although Verilog is not a strongly typed language, there are type restrictions on many operators. For instance, the concatenate, replicate, modulus, case equality, reduction, shift, and bit-wise operators cannot work with real-number operands. Real numbers can be converted to bits or vice versa using the afore-mentioned conversion functions. The system functions **$realtobits** and **$bitstoreal** can also be used for passing bit patterns across module ports, when they are represented as real numbers on the other side. For example, consider the following code:

```
module bus_driver (net_bus);
  output net_bus;

  real sig;
  wire [64:1] net_bus = $realtobits(sig);

endmodule
```

The variable `sig`, which is in real type is converted to bits using the statement

```
wire [64:1] net_bus = $realtobits(sig);
```

and passed to `net_bus`.

8.11.6 Probablistic Functions

Verilog has system functions to generate probabilistic distributions or random numbers**.** The **$random** function returns a random signed integer that is 32-bits.

8.12 Compiler Directives

Verilog has several compiler directives that add programming convenience to the development and maintenance of Verilog modules. All Verilog compiler directives are preceded by the (‘) character. This character is called *grave accent* (ASCII 0x60). It is different from the character (′), which is the *apostrophe* character