I) INTRODUCTION TO VERILOG

Verilog was originally intended for simulation and verification of digital circuits. Subsequently, with the addition of synthesis capability, Verilog has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the Verilog code into a hardware implementation of the described circuit. In this book our main use of Verilog will be for synthesis. Verilog is a complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important.

A) REPRESENTATION OF DIGITAL CIRCUITS IN VERILOG

When using CAD tools to synthesize a logic circuit, the designer can provide the initial description of the circuit in several different ways One efficient way is to write this description in the form of Verilog source code. The Verilog compiler translates this code into a logic circuit.

Verilog allows the designer to describe a desired circuit in a number of ways. One possibility is to use Verilog constructs that describe the structure of the circuit in terms of circuit elements, such as logic gates. A larger circuit is defined by writing code that connects such elements together. This approach is referred to as the structural representation of logic circuits. Another possibility is to describe a circuit more abstractly, by using logic expressions and Verilog programming constructs that define the desired behavior of the circuit, but not its actual structure in terms of gates. This is called the behavioral representation

B) STRUCTURAL SPECIFICATION OF LOGIC CIRCUITS

Verilog includes a set of gate-level primitives that correspond to commonly-used logic gates. A gate is represented by indicating its functional name, output, and inputs. For example, a two-input **AND** gate, with output y and inputs x_1 and x_2, is denoted as

AND (y, x_1, x_2);

A four-input **OR** gate is specified as

OR (y, x_1,x_2, x_3, x_4);

The keywords nand and nor are used to define the NAND and NOR gates in the same way. The NOT gate given by

NOT (y, x); implements y = x’

The gate-level primitives can be used to specify larger circuits. A logic circuit is specified in the form of a module that contains the statements that define the circuit. A module has inputs and outputs, which are referred to as its ports. The word port is a commonly-used term that refers to an input or output connection to an electronic circuit.

Consider the multiplexer circuit in Figure 2.36. This circuit can be represented by the Verilog code in Figure 2.37. The first statement gives the module a name, example1, and indicates that there are four port signals.

The next two statements declare that x_1, x_2, and s are to be treated as input signals, while f is the output. The actual structure of the circuit is specified in the four statements that follow. The NOT gate gives $k = \bar{s}$. . The AND gates produce $g = \bar{s}x_1$ and $h = sx_2$. The outputs of AND gates are combined in the OR gate to form

$$\begin{aligned} f &= g + h \\ &= \bar{s}x_1 + sx_2 \end{aligned}$$

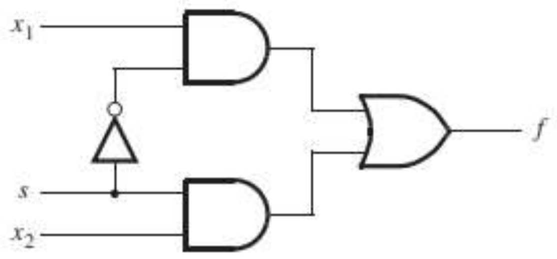


Figure 2.36 The logic circuit for a multiplexer.

```
module example1 (x1, x2, s, f);
   input x1, x2, s;
   output f;

   not (k, s);
   and (g, k, x1);
   and (h, s, x2);
   or (f, g, h);

endmodule
```

Figure 2.37 Verilog code for the circuit in Figure 2.36.

The module ends with the end module statement. We have written the Verilog keywords in bold type to make the text easier to read.

A second example of Verilog code is given in Figure 2.38. It defines a circuit that has four input signals, x_1, x_2, x_3, and x_4, and three output signals, f, g, and h. It implements the logic functions

$$
\begin{aligned}
g &= x_1x_3 + x_2x_4 \\
h &= (x_1 + \overline{x}_3)(\overline{x}_2 + x_4) \\
f &= g + h
\end{aligned}
$$

```
module  example2 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   and (z1, x1, x3);
   and (z2, x2, x4);
   or (g, z1, z2);
   or (z3, x1, ~x3);
   or (z4, ~x2, x4);
   and (h, z3, z4);
   or (f, g, h);

endmodule
```

Figure 2.38 Verilog code for a four-input circuit.

Instead of using explicit NOT gates to define $\overline{x}_2$ and $\overline{x}_3$, we have used the Verilog operator "~" (tilde character on the keyboard) to denote complementation. Thus, $\overline{x}_2$ is indicated as ~x2 in the code.

C) BEHAVIORAL SPECIFICATION OF LOGIC CIRCUITS:

Using gate-level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a logic circuit. One possibility is to define the circuit using logic expressions. Figure 2.40 shows how the circuit in Figure 2.36 can be defined with the expression

$$f = \overline{s}x_1 + sx_2$$

The AND and OR operations are indicated by the “&” and “|” Verilog operators, respectively. The assign keyword provides a continuous assignment for the signal f. The word continuous stems from the use of Verilog for simulation; whenever any signal on the right-hand side changes its state, the value of f will be re-evaluated.

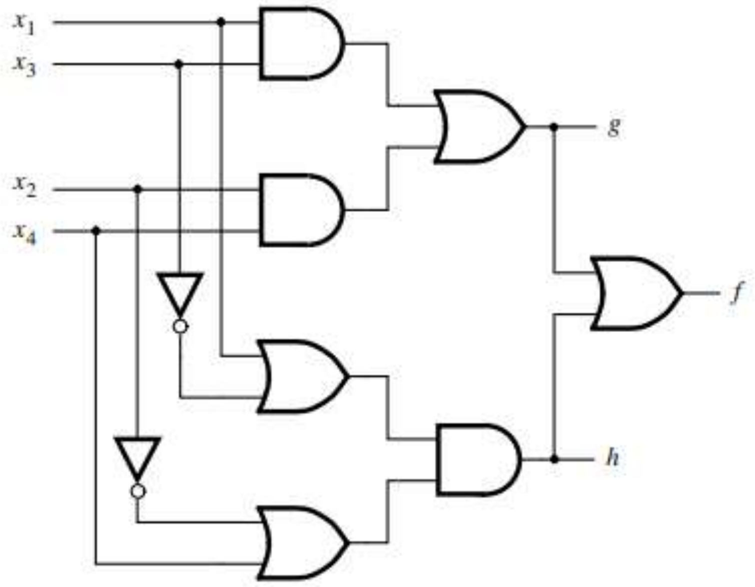


Figure 2.39 Logic circuit for the code in Figure 2.38.

```
module example3 (x1, x2, s, f);
   input x1, x2, s;
   output f;

   assign f = (~s & x1) | (s & x2);

endmodule
```

Figure 2.40 Using the continuous assignment to specify the circuit in Figure 2.36.

Using logic expressions makes it easier to write Verilog code. But even higher levels of abstraction can often be used to advantage. Consider again the multiplexer circuit of Figure 2.36. The circuit can be described in words by saying that f = x_1 if s = 0 and f = x_2 if s = 1. In Verilog, this behavior can be defined with the if-else statement

```
if (s == 0)
   f = x1;
else
   f = x2;
```

```
module example4 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   assign g = (x1 & x3) | (x2 & x4);
   assign h = (x1 | ~x3) & (~x2 | x4);
   assign f = g | h;

endmodule
```

Figure 2.41 Using the continuous assignment to specify the circuit in Figure 2.39.

```
// Behavioral specification
module example5 (x1, x2, s, f);
   input x1, x2, s;
   output f;
   reg f;

   always @(x1 or x2 or s)
      if (s == 0)
         f = x1;
      else
         f = x2;

endmodule
```

Figure 2.42 Behavioral specification of the circuit in Figure 2.36.

The complete code is given in Figure 2.42. The first line illustrates how a comment can be inserted. The if-else statement is an example of a Verilog procedural statement.

Verilog syntax requires that procedural statements be contained inside a construct called an always block, as shown in Figure 2.42. An always block can contain a single statement, as in this example, or it can contain multiple statements. A typical Verilog design module may include several always blocks, each representing a part of the circuit being modeled. An important property of the always block is that the statements it contains are evaluated in the order given in the code. This is in contrast to the continuous assignment statements, which are evaluated concurrently and hence have no meaningful order.

The part of the always block after the @ symbol, in parentheses, is called the sensitivity list. This list has its roots in the use of Verilog for simulation. The statements inside an always block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. In this way, the complexity of a simulation process is simplified, because it is not necessary to execute every statement in the code at all times. When Verilog is being employed for synthesis of circuits, as in this book, the sensitivity list simply tells the Verilog compiler which signals can directly affect the outputs produced by the always block.

D) HIERARCHICAL VERILOG CODE:

The examples of Verilog code given so far include just a single module. For larger designs, it is often convenient to create a hierarchical structure in the Verilog code, in which there is a top-level module that includes multiple instances of lower-level modules. To see how hierarchical Verilog code can be written consider the circuit in Figure 2.44. This circuit comprises two lower-level modules: the adder module, and the module that drives a 7-segment display.

The purpose of the circuit is to generate the arithmetic sum of the two inputs x and y, using the adder module, and then to show the resulting decimal value on the 7-segment display. For the adder module continuous assignment statements are used to specify the two-bit sum s_1s_0.

The assignment statement for s_0 uses the Verilog XOR operator, which is specified s_0 = a ^ b.

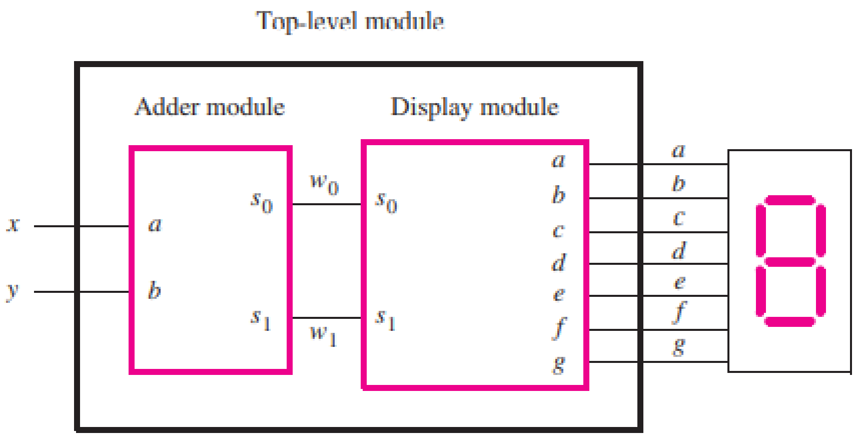


Figure 2.44 A logic circuit with two modules.

```
// An adder module
module  adder (a, b, s1, s0);
   input a, b;
   output s1, s0;

   assign s1 = a & b;
   assign s0 = a ^ b;

endmodule
```

Figure 2.45 Verilog specification of the circuit in Figure 2.12.

```
// A module for driving a 7-segment display
module display (s1, s0, a, b, c, d, e, f, g);
   input s1, s0;
   output a, b, c, d, e, f, g;

   assign a = ~s0;
   assign b = 1;
   assign c = ~s1;
   assign d = ~s0;
   assign e = ~s0;
   assign f = ~s1 & ~s0;
   assign g = s1 & ~s0;

endmodule
```

Figure 2.46 Verilog specification of the circuit in Figure 2.34.

```
module adder_display (x, y, a, b, c, d, e, f, g);
   input x, y;
   output a, b, c, d, e, f, g;
   wire w1, w0;

   adder U1 (x, y, w1, w0);
   display U2 (w1, w0, a, b, c, d, e, f, g);

endmodule
```

Figure 2.47 Hierarchical Verilog code for the circuit in Figure 2.44.

The code for the display module includes continuous assignment statements that correspond to the logic expressions for each of the seven outputs of the display circuit..

- The statement

assign b=1;

assigns the output b of the display module to have the constant value 1. The top-level Verilog module, named adder_display, is given in Figure 2.47. This module has the inputs x and y, and the outputs a,..., g.

- The statement

wire w_1, w_0;

is needed because the signals w1 and w0 are neither inputs nor outputs of the circuit in Figure 2.44. Since these signals cannot be declared as input or output ports in the Verilog code, they have to be declared as (internal) wires.

- The statement

adder U1 (x, y, w1, w0);

instantiates the adder module from Figure 2.45 as a submodule. The submodule is given a name, U1, which can be any valid Verilog name. In this instantiation statement the signals attached to the ports of the adder submodule are listed in the same order as those in Figure 2.45. Thus, the input ports x and y of the top-level module in Figure 2.47 are connected to the first two ports of adder, which are named a and b. The order in which signals are listed in the instantiation statement determines which signal is connected to each port in the submodule. The instantiation statement also attaches the last two ports of the adder submodule, which are its outputs, to the wires w_1 and w_0 in the top-level module.

- The statement

display U2 (w1, w0, a, b, c, d, e, f, g);

instantiates the other submodule in our circuit. Here, the wires w1 and w0, which have already been connected to the outputs of the adder submodule, are attached to the corresponding input ports of the display submodule. The display submodule’s output ports are attached to the a,..., g output ports of the top-level module.

II) VERILOG FOR COMBINATIONAL CIRCUITS

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

A) THE CONDITIONAL OPERATOR:

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

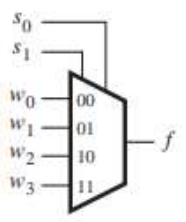
conditional_expression? true_expression : false_expression

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

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

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

A 4-to-1 multiplexer can be defined using the conditional operator in an assign statement as shown in Figure 4.25.

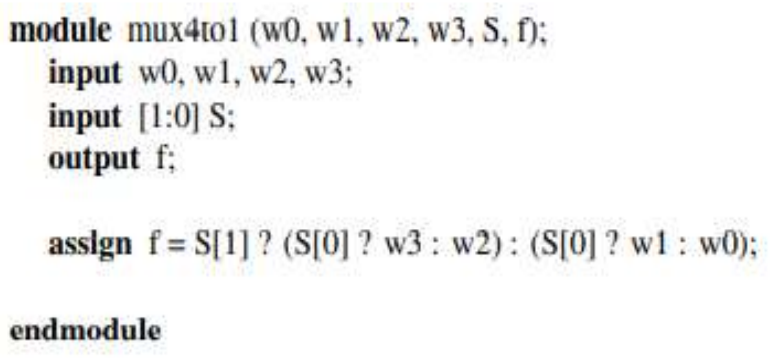


(a) Graphical symbol

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

(b) Truth table

As seen in the truth table in Figure above, if the select input $s_1 = 1$, then f is set to either w_2 or w_3 based on the value of s_0. Similarly, if $s_1 = 0$, then f is set to either w_0 or w_1. Figure 4.25 shows how nested conditional operators can be used to define this function. The module is called mux4to1. Its select inputs are represented by the two-bit vector S. The first conditional expression tests the value of bit s_1. If $s_1 = 1$, then s_0 is tested and f is set to w_3 if $s_0 = 1$ and f is set to w_2 if s0 = 0. This corresponds to the third and fourth rows of the truth table in Figure. Similarly, if $s_1 = 0$ the conditional operator on the right chooses $f = w_1$ if $s_0 = 1$ and $f = w_0$ if $s_0 = 0$, thus realizing the first two rows of the truth table



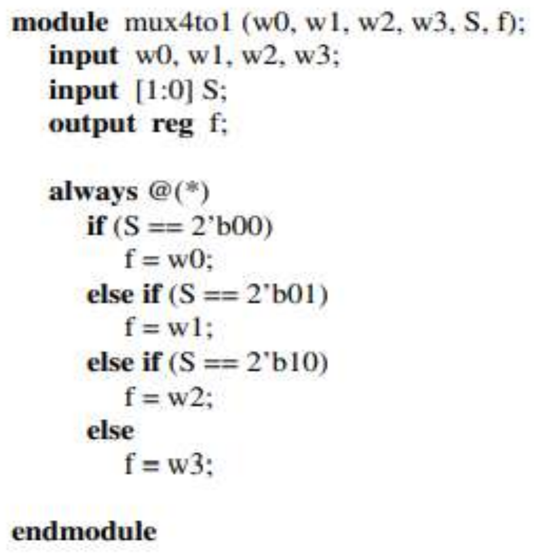
```
module mux4to1 (w0, w1, w2, w3, S, f);
   input w0, w1, w2, w3;
   input [1:0] S;
   output f;

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

endmodule
```

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

B) THE IF-ELSE STATEMENT:

It has the syntax if (conditional_expression) statement; else statement; If the expression is evaluated to true then the first statement (or a block of statements delineated by begin and end keywords) is executed, or else the second statement (or a block of statements) is executed.

The if-else statement can be used to implement larger multiplexers. A4-to-1 multiplexer is shown in Figure 4.27. The if-else clauses set f to the value of one of the inputs $w_0,...,w_3$, depending on the valuation of S. Another way of defining the same circuit is presented in Figure 4.28. In this case, a four-bit vector W is defined instead of single-bit signals w_0, w_1, w_2, and w_3. Also, the four different values of S are specified as decimal rather than binary numbers.

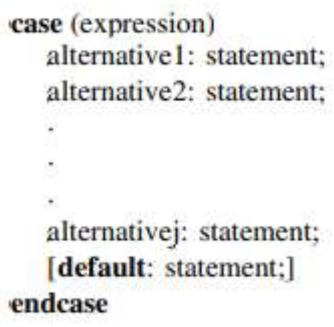


```
module mux4to1 (w0, w1, w2, w3, S, f);
    input w0, w1, w2, w3;
    input [1:0] S;
    output reg f;

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

endmodule
```

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

C) THE CASE STATEMENT:

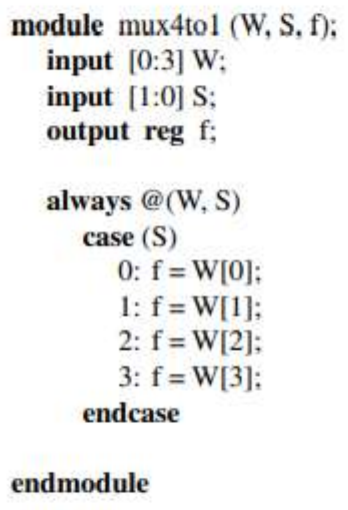
The if-else statement provides the means for choosing an alternative based on the value of an expression. When there are many possible alternatives, the code based on this statement may become awkward to read. Instead, it is often possible to use the Verilog case statement which is defined as



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

The value of the controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional default clause should be included.

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



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

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

endmodule
```

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

D) THE FOR LOOP:

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

The for loop has the syntax

for (initial_index; terminal_index; increment) statement;

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

Unlike for loops in high-level programming languages, the Verilog for loop does not specify changes that take place in time through successive loop iterations. Instead, during each iteration it specifies a different subcircuit.

EXAMPLE:

Figure 4.37 shows how the for loop can be used to specify a 2-to-4 decoder circuit. The effect of the loop is to repeat the if-else statement four times, for k = 0,..., 3. The first loop iteration sets y0 = 1 if W = 0 and En = 1. Similarly, the other three iterations set the values of y1, y2, and y3 according to the values of W and En. This arrangement can be used to specify a large n-to-2n decoder simply by increasing the sizes of vectors W and Y accordingly, and making n − 1 be the terminal index value of k.

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

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

endmodule
```

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

III) USING STORAGE ELEMENTS WITH CAD TOOLS

A) USING VERILOG CONSTRUCTS FOR STORAGE ELEMENTS:

One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates. Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules. Figure 5.30 shows a schematic created with a schematic capture tool, which includes three types of flip-flops that are imported from a library provided as part of the CAD system. The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive edge-triggered T flip-flop. The D and T flip-flops have asynchronous, active-low clear and preset inputs. If these inputs are not connected in a schematic, then the CAD tool makes them inactive by assigning the default value of 1 to them. When the gated D latch is synthesized for implementation in a chip, the CAD tool may not generate the cross-coupled NOR or NAND gates. In some chips the AND-OR circuit depicted in Figure 5.31 may be preferable. This circuit is functionally equivalent to the cross-coupled. One aspect of this circuit should be mentioned. From the functional point of view, it appears that the circuit can be simplified by removing the AND gate with the inputs Data and Latch.

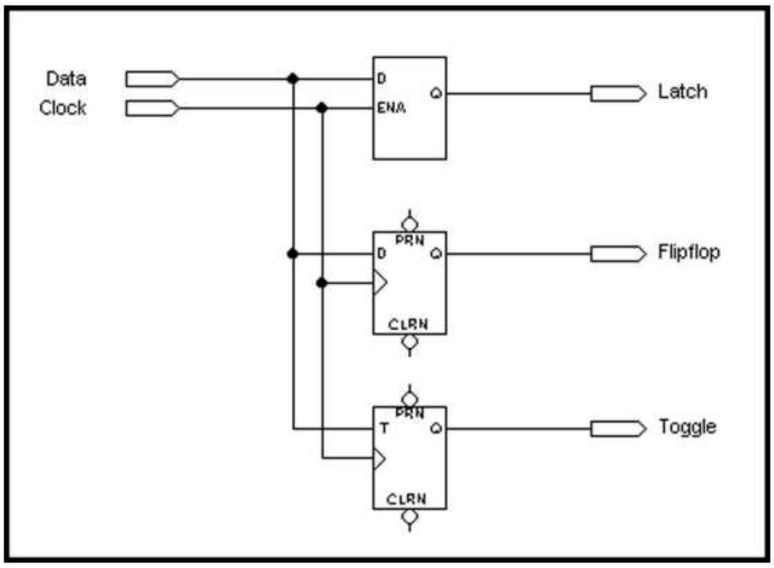


Figure 5.30 Three types of storage elements in a schematic.

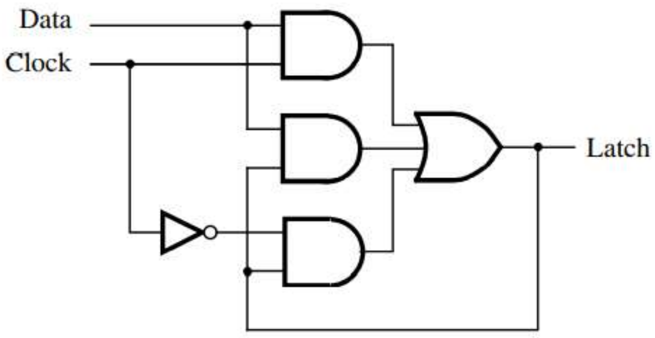


Figure 5.31 Gated D latch generated by CAD tools.

Without this gate, the top AND gate sets the value stored in the latch when the clock is 1, and the bottom AND gate maintains the stored value when the clock is 0. But without this gate, the circuit has a timing problem known as a static hazard.

The results of a timing simulation for the implementation in Figure 5.32 are given in Figure 5.33. The Latch signal, which is the output of the gated D latch implemented as indicated in Figure 5.31, follows the Data input whenever the Clock signal is 1. Because of propagation delays in the chip, the Latch signal is delayed in time with respect to the Data signal. Since the Flipflop signal is the output of the D flip-flop, it changes only after a positive clock edge. Similarly, the output of the T flip-flop, called Toggle in the figure, toggles when Data = 1 and a positive clock edge occurs.

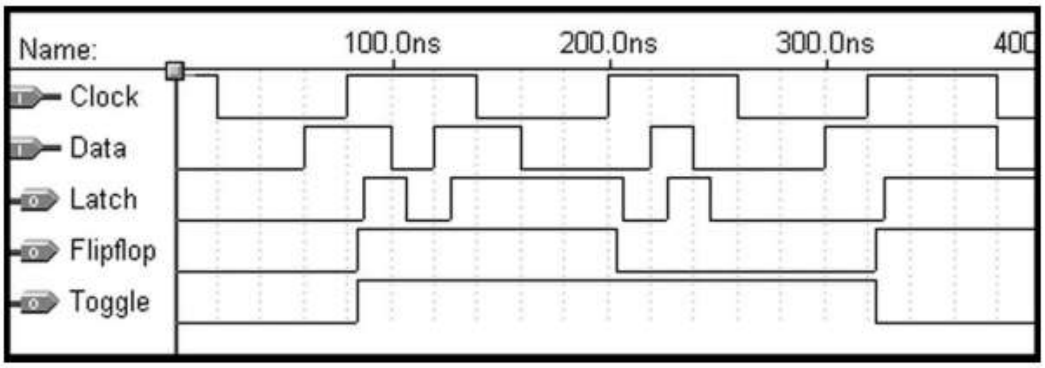


Figure 5.33 Timing simulation for the storage elements in Figure 5.30.

```
module  D_latch (D, Clk, Q);
   input  D, Clk;
   output  reg  Q;

   always @(D, Clk)
      if (Clk)
         Q = D;

endmodule
```

Figure 5.34 Code for a gated D latch.

The timing diagram illustrates the delay from when the positive clock edge occurs at the input pin of the chip until a change in the flip-flop output appears at the output pin of the chip. This time is called the clock-to-output time, t_{co}.

The code in Figure 5.34 defines a module named D_latch, Example 5.1 which has the inputs D and Clk and the output Q. The if clause defines that the Q output must take the value of D when Clk = 1. Since no else clause is given, a latch will be synthesized to maintain the value of Q when Clk = 0. Therefore, the code describes a gated D latch. The sensitivity list includes Clk and D because both of these signals can cause a change in the value of the Q output.

A simple way of specifying a storage element is by using the if-else statement to describe the desired behavior responding to changes in the levels of data and clock inputs. Consider the always block

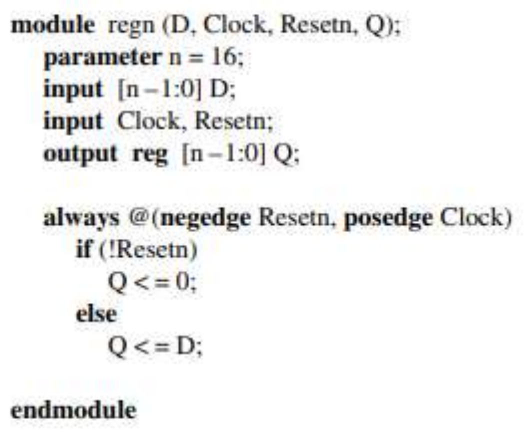
```
always @(Control, B)
   if (Control)
      A = B;
```

where A is a variable of reg type. This code specifies that the value of A should be made equal to the value of B when Control = 1. But the statement does not indicate an action that should occur when Control = 0. In the absence of an assigned value, the Verilog compiler assumes that the value of A caused by the if statement must be maintained when Control is not equal to 1. This notion of implied memory is realized by instantiating a latch in the circuit.

An **always** construct is used to define a circuit that responds to changes in the signals that appear in the sensitivity list. While in the examples presented so far the always blocks are sensitive to the levels of signals, it is also possible to specify that a response should take place only at a particular edge of a signal. The desired edge is specified by using the Verilog keywords posedge and negedge, which are used to implement edge-triggered circuits.

B) USING VERILOG CONSTRUCTS FOR REGISTERS AND COUNTERS:

i) **AN N-BIT REGISTER**: Since registers of different sizes are often needed in logic circuits, it Example 5.9 is advantageous to define a register module for which the number of flip-flops can be easily changed. The code for an n-bit register is given in Figure 5.46. The parameter n specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.



```
module regn (D, Clock, Resetn, Q);
   parameter n = 16;
   input [n–1:0] D;
   input Clock, Resetn;
   output reg [n–1:0] Q;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q < = 0;
      else
         Q < = D;

endmodule
```

Figure 5.46 Code for an *n*-bit register with asynchronous clear.

ii) **A FOUR-BIT SHIFT REGISTER**:

Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the D input.

Figure 5.47 defines the module named muxdff, which represents this subcircuit. The two data inputs are named D0 and D1, and they are selected using the Sel input. The if-else statement specifies that on the positive clock edge if Sel = 0, then Q is assigned the value of D0; otherwise, Q is assigned the value of D1.

```
module  muxdff (D0, D1, Sel, Clock, Q);
   input  D0, D1, Sel, Clock;
   output  reg  Q;

   always @(posedge Clock)
      if (!Sel)
         Q <= D0;
      else
         Q <= D1;

endmodule
```

Figure 5.47 Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

An alternative way of defining the same circuit is given in Figure 5.48. In this code, the conditional assignment statement specifies a 2-to-1 multiplexer with the output D, which is then connected to the flip-flop in the always block.

```
module  muxdff (D0, D1, Sel, Clock, Q);
   input  D0, D1, Sel, Clock;
   output  reg  Q;

   wire  D;
   assign D = Sel ? D1 : D0;

   always @(posedge Clock)
      Q <= D;

endmodule
```

Figure 5.48 Alternative code for a D flip-flop with a 2-to-1 multiplexer on the D input.

Figure 5.49 defines the four-bit shift register. The module Stage3 instantiates the leftmost flip-flop, which has the output Q3, and the module Stage0 instantiates the right-most flip-flop, Q0. When L = 1, the register is loaded in parallel from the R input; and when L = 0, shifting takes place in the left to right direction. Serial data is shifted into the mostsignificant bit, Q3, from the w input. ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER.

```
module shift4 (R, L, w, Clock, Q);
   input [3:0] R;
   input L, w, Clock;
   output wire [3:0] Q;

   muxdff Stage3 (w, R[3], L, Clock, Q[3]);
   muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
   muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
   muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule
```

Figure 5.49 Hierarchical code for a four-bit shift register.

A different style of code for the four-bit shift register is given in Figure 5.50. Instead of using subcircuits, the shift register is defined. All actions take place at the positive edge of the clock. If L = 1, the register is loaded in parallel with the four bits of input R. If L = 0, the contents of the register are shifted to the right and the value of the input w is loaded into the most-significant bit.

```
module shift4 (R, L, w, Clock, Q);
   input [3:0] R;
   input L, w, Clock;
   output reg [3:0] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] <= Q[1];
         Q[1] <= Q[2];
         Q[2] <= Q[3];
         Q[3] <= w;
      end

endmodule
```

Figure 5.50 Alternative code for a four-bit shift register.

iii) UP-COUNTER

Figure 5.52 represents a four-bit up-counter with a reset input, Resetn, and an enable input, E. The outputs of the flip-flops in the counter are represented by the vector named Q. The if statement specifies an asynchronous reset of the counter if Resetn = 0. The else if clause specifies that if E = 1 the count is incremented on the positive clock edge.

```
module upcount (Resetn, Clock, E, Q);
   input Resetn, Clock, E;
   output reg [3:0] Q;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q < = 0;
      else if (E)
         Q < = Q + 1;

endmodule
```

Figure 5.52 Code for a four-bit up-counter.

iv) UP-COUNTER WITH PARALLEL LOAD:

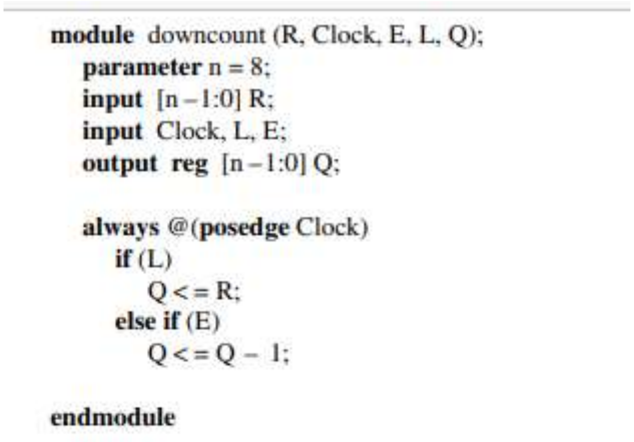
The code in Figure 5.53 defines an up-counter has a parallel-load input in addition to a reset input. The parallel data is provided as the input vector R. The first if statement provides the same asynchronous reset as in Figure 5.52. The else if clause specifies that if L = 1 the flip-flops in the counter are loaded in parallel from the R inputs on the positive clock edge. If L = 0, the count is incremented, under control of the enable input E.

```
module upcount (R, Resetn, Clock, E, L, Q);
   input [3:0] R;
   input Resetn, Clock, E, L;
   output reg [3:0] Q;

   always @(negedge Resetn, posedge Clock)
      if (!Resetn)
         Q < = 0;
      else if (L)
         Q < = R;
      else if (E)
         Q < = Q + 1;

endmodule
```

Figure 5.53 A four-bit up-counter with a parallel load.

v) DOWN-COUNTER WITH PARALLEL LOAD

Figure 5.54 shows the code for a down-counter named down count. A down-counter is normally used by loading it with some starting count and then decrementing its contents. The starting count is represented in the code by the vector R. On the positive clock edge, if L = 1 the counter is loaded with the input R, and if L = 0 the count is decremented, under control of the enable input E.

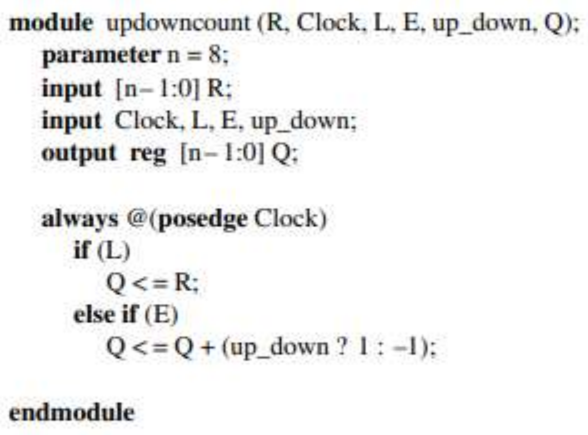


```
module  downcount (R, Clock, E, L, Q);
   parameter n = 8;
   input  [n – 1:0] R;
   input  Clock, L, E;
   output  reg  [n – 1:0] Q;

   always @(posedge Clock)
      if (L)
         Q < = R;
      else if (E)
         Q < = Q – 1;

endmodule
```

Figure 5.54 A down-counter with a parallel load.

vi) UP/DOWN COUNTER

Verilog code for an up/down counter is given in Figure 5.55. This module combines the capabilities of the counters defined in Figures 5.53 and 5.54. It includes a control signal up_down that governs the direction of counting.



```
module  updowncount (R, Clock, L, E, up_down, Q);
   parameter n = 8;
   input  [n– 1:0] R;
   input  Clock, L, E, up_down;
   output  reg  [n– 1:0] Q;

   always @(posedge Clock)
      if (L)
         Q < = R;
      else if (E)
         Q < = Q + (up_down ? 1 : –1);

endmodule
```

Figure 5.55 Code for an up/down counter.