Verilog HDL: Digital Design and Modeling

Chapter 8

Behavioral Modeling
//module showing use of the initial keyword
module initial_ex (x1, x2, x3, x4, x5);

output x1, x2, x3, x4, x5;
reg x1, x2, x3, x4, x5;

//display variables
initial
$monitor ($time, " x1x2x3x4x5 = %b", {x1, x2, x3, x4, x5});

//initialize variables to 0
//multiple statements require begin . . . end
initial
begin
  #0  x1 = 1'b0
  x2 = 1'b0;
  x3 = 1'b0;
  x4 = 1'b0;
  x5 = 1'b0;
end

//set x1
//single statement requires no begin . . . end
initial
  #10  x1 = 1'b1;

//set x2 and x3
initial
begin
  #10  x2 = 1'b1;
  #10  x3 = 1'b1;
end

//set x4 and x5
initial
begin
  #10  x4 = 1'b1;
  #10  x5 = 1'b1;
end

//continued on next page

Figure 8.1  Module to illustrate the use of the initial statement.
//reset variables
initial
begin
  #20 x1 = 1'b0;
  #10 x2 = 1'b0;
  #10 x3 = 1'b0;
  #10 x4 = 1'b0;
  #10 x5 = 1'b0;
end

//determine length of simulation
initial
  #70 $finish;
endmodule

Figure 8.1  (Continued)

Page 368

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>x1x2x3x4x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
</tr>
<tr>
<td>10</td>
<td>11010</td>
</tr>
<tr>
<td>20</td>
<td>01111</td>
</tr>
<tr>
<td>30</td>
<td>00111</td>
</tr>
<tr>
<td>40</td>
<td>00011</td>
</tr>
<tr>
<td>50</td>
<td>00001</td>
</tr>
<tr>
<td>60</td>
<td>00000</td>
</tr>
</tbody>
</table>

Figure 8.2  Outputs for the module of Figure 8.1.

Figure 8.3   Waveforms for the module of Figure 8.1.
//module showing the use of initial keyword
module initial_ex2 (z1);

output [7:0] z1;

reg [7:0] z1;

//display variables
initial
$monitor ("z1 = %b", z1);

//apply stimulus
initial
begin
    #0 z1 = 8'h00;
    #10 z1 = 8'hc0;
    #10 z1 = 8'he0;
    #10 z1 = 8'hfc;
    #10 z1 = 8'hff;
    #10 z1 = 8'h00;
end
endmodule

Figure 8.4  Vector waveforms generated by an initial statement.

z1 = 00000000
z1 = 11000000
z1 = 11100000
z1 = 11111100
z1 = 11111111
z1 = 00000000

Figure 8.5  Outputs for the module of Figure 8.4.
Figure 8.6 Waveforms for the module of Figure 8.4.

```verilog
module clk_gen2 (clk);

output clk;
reg clk;

initial //initialize clock to 0
    clk = 1'b0;

always //toggle clock every 10 time units
    #10 clk = ~clk;

initial //determine length of simulation
    #100 $finish;
endmodule
```

Figure 8.7 Clock waveform generation.

Figure 8.8 Waveform for the clock generation module of Figure 8.7.
// clock generation using initial and forever
module clk_gen3 (clk);

output clk;
reg clk;

// define clock
initial begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

// define length of simulation
initial
    #100 $finish;
endmodule

Figure 8.9 Alternative method to generate clock pulses.

Figure 8.10 Waveforms for the clock generation module of Figure 8.9.
//behavioral 3-input AND gate
module and3_bh (x1, x2, x3, z1);

input x1, x2, x3;
output z1;

wire x1, x2, x3; //alternatively do not declare wires
reg z1; //because inputs are wire by default

always @ (x1 or x2 or x3)
  z1 = #5 (x1 & x2 & x3);
endmodule

Figure 8.11 Module to illustrate the use of an always statement with an event control list.

//test bench for behavioral 3-input AND gate
module and3_bh_tb;

reg x1, x2, x3;
wire z1;

//generate stimulus and display variables
initial begin
  apply_stimulus
  reg [3:0] invect;
  for (invect = 0; invect < 8; invect = invect + 1)
    begin
      {x1, x2, x3} = invect [2:0];
      $display ($time, "x1x2x3 = %b%b%b, z1 = %b", x1, x2, x3, z1);
    end
end

//instantiate the module into the test bench
and3_bh inst1 ( .x1(x1), .x2(x2), .x3(x3), .z1(z1) );
endmodule

Figure 8.12 Test bench for the 3-input AND gate of Figure 8.11.
6 \times 1 \times 2 \times 3 = 000, z_1 = 0
12 \times 1 \times 2 \times 3 = 001, z_1 = 0
18 \times 1 \times 2 \times 3 = 010, z_1 = 0
24 \times 1 \times 2 \times 3 = 011, z_1 = 0
30 \times 1 \times 2 \times 3 = 100, z_1 = 0
36 \times 1 \times 2 \times 3 = 101, z_1 = 0
42 \times 1 \times 2 \times 3 = 110, z_1 = 0
48 \times 1 \times 2 \times 3 = 111, z_1 = 1

**Figure 8.13**  Outputs for the behavioral model of a 3-input AND gate of Figure 8.11.

**Figure 8.14**  Waveforms for the behavioral model of a 3-input AND gate of Figure 8.11.
//behavioral model for a 4-bit adder
module adder4 (a, b, cin, sum, cout);

input [3:0] a, b;
input cin;
output [3:0] sum;
output cout;

wire [3:0] a, b;
wire cin;

reg [3:0] sum;
reg cout;

always @ (a or b or cin)
begin
  sum = a + b + cin;
  cout = (a[3] & b[3]) |
        & (a[0] & b[0])) |
        & (a[0] | b[0]) & cin);
end
endmodule

Figure 8.15 Module for the 4-bit adder of Example 8.4.
module adder4_tb;

reg [3:0] a, b;
reg cin;
wire [3:0] sum;
wire cout;

//display variables
initial
$monitor("a=%b, b=%b, cin=%b, cout=%b, sum=%b",
    a, b, cin, cout, sum);

//apply input vectors
initial
begin
    #0 a=4'b0000; b=4'b0000; cin=1'b0;
    #10 a=4'b0001; b=4'b0001; cin=1'b0;
    #10 a=4'b0011; b=4'b0011; cin=1'b0;
    #10 a=4'b0101; b=4'b0101; cin=1'b0;
    #10 a=4'b1001; b=4'b1001; cin=1'b0;
    #10 a=4'b0101; b=4'b0101; cin=1'b1;
    #10 a=4'b1001; b=4'b1001; cin=1'b1;
    #10 a=4'b1100; b=4'b1100; cin=1'b1;
    #10 a=4'b1111; b=4'b1111; cin=1'b1;
end

//instantiate the module into the test bench
adder4 inst1 (  
    .a(a),
    .b(b),
    .cin(cin),
    .sum(sum),
    .cout(cout))
;

endmodule

Figure 8.16  Test bench for the 4-bit adder of Figure 8.15.
a=0000, b=0000, cin=0, cout=0, sum=0000  
a=0001, b=0001, cin=0, cout=0, sum=0010  
a=0001, b=0011, cin=0, cout=0, sum=0100  
a=0101, b=0001, cin=0, cout=0, sum=0110  
a=0111, b=0001, cin=0, cout=0, sum=1000  
a=0101, b=0101, cin=0, cout=0, sum=1010  
a=1001, b=0101, cin=0, cout=1, sum=1111  
a=1000, b=1000, cin=1, cout=0, sum=0001  
a=1011, b=1110, cin=0, cout=1, sum=1111  
a=1111, b=1111, cin=1, cout=1, sum=1111

**Figure 8.17**  Outputs for the 4-bit adder of Figure 8.15.

**Figure 8.18**  Waveforms for the 4-bit adder of Figure 8.15.
//add shift operations
module add_shift (a, b, sum, left_shift_rslt, right_shift_rslt);

input [7:0] a, b;
output [8:0] sum;
output [15:0] left_shift_rslt, right_shift_rslt;

wire [7:0] a, b;
reg [8:0] sum;
reg [15:0] left_shift_rslt, right_shift_rslt;

always @ (a or b)
begin
    sum = a + b;
    left_shift_rslt = sum << 4;
    right_shift_rslt = sum >> 4;
end
endmodule

Figure 8.19 Behavioral design module for an 8-bit add-shift unit.

//add shift test bench
module add_shift_tb;

reg [7:0] a, b;
wire [8:0] sum;
wire [15:0] left_shift_rslt, right_shift_rslt;

initial //display variables
$monitor("a=%b, b=%b, sum=%b, left_shift_rslt=%b,
          right_shift_rslt=%b", a, b, sum, left_shift_rslt, right_shift_rslt);

initial //apply input vectors
begin
    #0 a = 8'b0101_0101;
        b = 8'b0101_0101;

    #10 a = 8'b0000_1100;
        b = 8'b0000_0100;

    #10 a = 8'b1111_0000;
        b = 8'b0000_1111;

    #10 a = 8'b1010_0000;
        b = 8'b0000_1111; //continued on next page

Figure 8.20 Test bench for the add-shift unit of Figure 8.19.
#10  a = 8'b1111_1111;
      b = 8'b1111_1111;
#10  $stop;
end

add_shift inst1 (   //instantiate the module
    .a(a),
    .b(b),
    .sum(sum),
    .left_shift_rslt(left_shift_rslt),
    .right_shift_rslt(right_shift_rslt)
);
endmodule

Figure 8.20  (Continued)

Page 379

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>sum</th>
<th>left_shift_rslt</th>
<th>right_shift_rslt</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010101</td>
<td>01010101</td>
<td>010101010</td>
<td>0000101010100000</td>
<td>0000000000001010</td>
</tr>
<tr>
<td>00001100</td>
<td>00000100</td>
<td>000010000</td>
<td>0000000000000000</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>11110000</td>
<td>00001111</td>
<td>011111111</td>
<td>0001111111100000</td>
<td>0000000000001111</td>
</tr>
<tr>
<td>10100000</td>
<td>00001111</td>
<td>010101111</td>
<td>0000101011110000</td>
<td>0000000000001010</td>
</tr>
<tr>
<td>11111111</td>
<td>11111111</td>
<td>111111110</td>
<td>0001111111100000</td>
<td>0000000000001111</td>
</tr>
</tbody>
</table>

Figure 8.21  Outputs for the add-shift unit of Figure 8.19.
Figure 8.22  Waveforms for the add-shift unit of Figure 8.19.

Page 381

```verilog
//behavioral 4:1 multiplexer
module mux_4to1_behav (d, s, enbl, z1);

input [3:0] d;
input [1:0] s;
input enbl;
output z1;

//define internal nets
wire net0, net1, net2, net3;
reg z1;  //z1 is used in the always statement
          //and must be declared as type reg

//define AND gates
assign net0 = (d[0] & ~s[1] & ~s[0] & enbl),
net1 = (d[1] & ~s[1] & s[0] & enbl),
net3 = (d[3] & s[1] & s[0] & enbl);

always @ (net0 or net1 or net2 or net3)
z1 = (net0 || net1 || net2 || net3);

endmodule
```

Figure 8.24  Behavioral module for the 4:1 multiplexer of Example 8.6.
// test bench for behavioral 4:1 multiplexer
module mux_4to1_behav_tb;

reg [3:0] d;
reg [1:0] s;
reg enbl;
wire z1;

initial // display variables
$monitor ("s=%b, d=%b, z1=%b", s, d, z1);

initial // apply input vectors
begin
  #0 s=2'b00; d=4'b0000; enbl=1'b1; // d[0]=0; z1=0
  #10 s=2'b00; d=4'b1001; enbl=1'b1; // d[0]=1; z1=1
  #10 s=2'b01; d=4'b1010; enbl=1'b1; // d[1]=1; z1=1
  #10 s=2'b10; d=4'b0011; enbl=1'b1; // d[2]=0; z1=0
  #10 s=2'b01; d=4'b1011; enbl=1'b1; // d[1]=1; z1=1
  #10 s=2'b11; d=4'b1000; enbl=1'b1; // d[3]=1; z1=1
  #10 s=2'b11; d=4'b0110; enbl=1'b1; // d[3]=0; z1=0
  #10 $stop;
end

// instantiate the module into the test bench
mux_4to1_behav inst1 ( .d(d),
  .s(s),
  .enbl(enbl),
  .z1(z1)
);
endmodule

Figure 8.25  Test bench for the 4:1 multiplexer of Figure 8.24.

s=00, d=0000, z1=0
s=00, d=1001, z1=1
s=01, d=1010, z1=1
s=10, d=0011, z1=0
s=01, d=1011, z1=1
s=11, d=1000, z1=1
s=11, d=0110, z1=0

Figure 8.26  Outputs for the 4:1 multiplexer of Figure 8.24.
Waveforms for the 4:1 multiplexer of Figure 8.24.

Behavioral module for an 8-bit odd parity generator.

```
//behavioral 8-bit odd parity generator
module par_gen8_behav (x, z1);

input [7:0] x;
output z1;

//inputs are wire by default
reg z1;

always @ (x)

endmodule
```
//test bench for behavioral 4:1 multiplexer
module par_gen8_behav_tb;

reg [7:0] x;
wire z1;

//display variables
initial
$monitor ("x=%b, z1=%b", x, z1);

//apply input vectors
initial
begin
  #0 x = 8'b0000_0000;
  #10 x = 8'b0001_1010;
  #10 x = 8'b1001_1010;
  #10 x = 8'b1001_1111;
  #10 x = 8'b1101_0011;
  #10 x = 8'b1001_1010;
  #10 x = 8'b1101_1010;
  #10 x = 8'b1011_1111;
  #10 x = 8'b1111_1111;
  #10 $stop;
end

//instantiate the module into the test bench
par_gen8_behav inst1 (  
  .x(x),
  .z1(z1)
);
endmodule

Figure 8.29  Test bench for the 8-bit odd parity generator of Figure 8.28.

<table>
<thead>
<tr>
<th>x</th>
<th>z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>1</td>
</tr>
<tr>
<td>00011010</td>
<td>0</td>
</tr>
<tr>
<td>10011010</td>
<td>1</td>
</tr>
<tr>
<td>10011111</td>
<td>1</td>
</tr>
<tr>
<td>11010011</td>
<td>0</td>
</tr>
<tr>
<td>10011010</td>
<td>1</td>
</tr>
<tr>
<td>11011010</td>
<td>0</td>
</tr>
<tr>
<td>10111111</td>
<td>0</td>
</tr>
<tr>
<td>11111111</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 8.30  Outputs for the 8-bit odd parity generator of Figure 8.28.
Figure 8.31  Waveforms for the 8-bit odd parity generator of Figure 8.28.

//behavioral intrastatement delay example
module intra_stmt_dly (x1, x2);

`timescale 10ns / 1ns  //assign timescale

output x1, x2;
reg x1, x2;

initial  //apply input signals
begin
    x1 = #0 1'b0;
    x2 = #0 1'b0;

    x1 = #1 1'b1;
    x2 = #0.5 1'b1;

    x1 = #1 1'b0;
    x2 = #2 1'b0;

    x1 = #1 1'b1;
    x2 = #2 1'b1;

    x1 = #2 1'b0;
    x2 = #1 1'b0;
end
endmodule

Figure 8.32  Module to generate waveforms using intrastatement delays.
Figure 8.33 Waveforms for the intrastatement delay module of Figure 8.32.

Figure 8.35 Module to illustrate intrastatement delay.
//test bench for intrastatement delay
module intra_stmt_dly2_tb;

reg x1, x2, x3, x4;
wire z1, z2, z3;

//apply input vectors and display variables
initial begin: apply_stimulus
  reg [4:0] invect;
  for (invect=0; invect<16; invect=invect+1)
    begin
      {x1, x2, x3, x4} = invect [4:0];
      #10 $display("x1 x2 x3 x4 = %b, z1=%b, z2=%b, z3=%b",
                  {x1, x2, x3, x4}, z1, z2, z3);
    end
end

//instantiate the module into the test bench
intra_stmt_dly2 inst1 (.x1(x1),
                        .x2(x2),
                        .x3(x3),
                        .x4(x4),
                        .z1(z1),
                        .z2(z2),
                        .z3(z3));
endmodule

Figure 8.36  Test bench for the intrastatement delay module of Figure 8.35.
\[ z_1 = \#2 \left( x_1 x_2' x_3 + x_1 x_2' x_4 + x_1' x_3' x_4 + x_2 x_3' x_4 \right) \]

\[ z_2 = \#3 \left( x_1 \geq x_2 \right) \oplus x_3 \oplus x_4 \]

\[ z_3 = \#4 \left( x_1 \oplus x_2 \oplus x_3 \oplus x_4 \right) \]

\[
\begin{array}{cccccc}
& x_1 & x_2 & x_3 & x_4 & z_1 & z_2 & z_3 \\
0000 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0001 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0010 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0011 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
0100 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0101 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
0110 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0111 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1000 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
1001 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1010 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
1011 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
1100 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
1101 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
1110 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1111 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

**Figure 8.37**  Outputs for the intrastatement delay module of Figure 8.35.

**Figure 8.38**  Waveforms for the intrastatement delay module of Figure 8.35.
//behavioral module to illustrate interstatement delay
module inter_stmt_dly (x1, x2, x3, z1, z2);

input x1, x2, x3;
output z1, z2;

reg z1, z2;

always @(x1 or x2 or x3)
begin
    z1 = (x1 & x2 & ~x3);
    #5 z2 = ~(x1 ^ x2 ^ x3);
end
endmodule

Figure 8.39 Module to demonstrate interstatement delay.

//test bench for interstatement delay
module inter_stmt_dly_tb;

reg x1, x2, x3;
wire z1, z2;

//apply input vectors and display variables
initial begin:
    apply_stimulus
    reg [3:0] invect;
    for (invect=0; invect<8; invect=invect+1)
    begin
        {x1, x2, x3} = invect [3:0];
        #10 $display("x1 x2 x3 = %b, z1 = %b, z2 = %b",
        {x1, x2, x3}, z1, z2);
    end
end

//instantiate the module into the test bench
inter_stmt_dly inst1 (.
    .x1(x1),
    .x2(x2),
    .x3(x3),
    .z1(z1),
    .z2(z2)
);
endmodule

Figure 8.40 Test bench for the interstatement delay module of Figure 8.39.
\[ z_1 = (x_1 \& x_2 \& \neg x_3); \]
\[ z_2 = \neg(x_1 \oplus x_2 \oplus x_3); \]

\begin{tabular}{|c|c|c|}
\hline
x_1 & x_2 & x_3 \tabularnewline
\hline
0 & 0 & 0 \tabularnewline
0 & 0 & 1 \tabularnewline
0 & 1 & 0 \tabularnewline
0 & 1 & 1 \tabularnewline
1 & 0 & 0 \tabularnewline
1 & 0 & 1 \tabularnewline
1 & 1 & 0 \tabularnewline
1 & 1 & 1 \tabularnewline
\hline
\end{tabular}

\[ z_1 = 0, \quad z_2 = 1 \]
\[ z_1 = 0, \quad z_2 = 0 \]
\[ z_1 = 0, \quad z_2 = 0 \]
\[ z_1 = 0, \quad z_2 = 1 \]

\textbf{Figure 8.41}  Outputs for the interstatement delay module of Figure 8.39.

\textbf{Figure 8.42}  Waveforms for the interstatement delay module of Figure 8.39.
// example of blocking procedural assignment
module blocking1 (x1, x2, z1, z2, z3);

input x1, x2;
output z1, z2, z3;
reg z1, z2, z3;

always @(x1 or x2)
begin
  #1 z1 = x1 & x2;
  #1 z2 = x1 | x2;
  #1 z3 = x1 ^ x2;
end
endmodule

Figure 8.43  Module showing delayed blocking assignments.

// test bench for blocking assignment
module blocking1_tb;

reg x1, x2;
wire z1, z2, z3;

// display variables
initial
$monitor("x1 x2 = %b, z1 = %b, z2 = %b, z3 = %b",
        {x1, x2}, z1, z2, z3);

// apply input vectors
initial
  #0 x1 = 1'b0; x2 = 1'b0;
  #10 x1 = 1'b0; x2 = 1'b1;
  #10 x1 = 1'b1; x2 = 1'b0;
  #10 x1 = 1'b1; x2 = 1'b1;
  #10 $stop;
end

// instantiate the module into the test bench
blocking1 inst1 (.
    .x1(x1),
    .x2(x2),
    .z1(z1),
    .z2(z2),
    .z3(z3)
);
endmodule

Figure 8.44  Test bench for the blocking assignment module of Figure 8.43.
Figure 8.45  Waveforms for the blocking assignment module of Figure 8.43.

Figure 8.46  Module showing blocking assignments using intrastatement delays.
Figure 8.47  Test bench for the module using blocking assignments for intrastatement delays.
//blocking assignment to generate waveform
module blocking3 (clk);
output clk;
reg clk;

initial begin
    clk = #4 1'b0;
    clk = #4 1'b1;
    clk = #8 1'b0;
    clk = #10 1'b1;
    clk = #10 1'b0;
    clk = #20 1'b1;
    clk = #30 1'b0;
end
endmodule

Figure 8.49 Clock generation using blocking procedural assignment with intra-statement delays.

Figure 8.50 Waveform for Figure 8.49.
//example of nonblocking statements
module nonblock (clk);

output clk;
reg clk;

initial begin
clk <= #5 1'b0;
clk <= #10 1'b1;
clk <= #15 1'b0;
clk <= #20 1'b1;
clk <= #30 1'b0;
end
endmodule

Figure 8.51   Module to illustrate the use of nonblocking assignments with intra-statement delays.

Figure 8.52   Waveform for Figure 8.51.
//example of nonblocking assignment
module nonblock3 (data_reg_a, data_reg_b, index);

output [7:0] data_reg_a, data_reg_b;
output [3:0] index;

reg [7:0] data_reg_a, data_reg_b;
reg [3:0] index;

initial
begin
    data_reg_a = 8'h84;
data_reg_b = 8'h0f;
index = 4'b0;

data_reg_a [1:0] <= #10 2'b11;
data_reg_b [7:0] <= #5 {data_reg_a[3:0], 4'b0011};
index <= index +1;
end
endmodule

Figure 8.53 Module to model blocking and nonblocking assignments.

Figure 8.54 Waveforms for Figure 8.53.
//parallel-in serial-out shift register
module shift_reg_piso (clk, load, x, y, z1);

input clk, load;
input [1:4] x;
output [1:4] y;
output z1;

reg [1:4] y; //must be reg if used in always
assign z1 = y[4];

always @ (posedge clk)
begin
    y[1] <= ((load && x[1]) || (~load && 1'b0));
    y[2] <= ((load && x[2]) || (~load && y[1]));
    y[3] <= ((load && x[3]) || (~load && y[2]));
    y[4] <= ((load && x[4]) || (~load && y[3]));
end
endmodule

Figure 8.55  Behavioral module for a 4-bit parallel-in, serial-out shift register using nonblocking assignments.

//test bench for piso shift register
module shift_reg_piso_tb;
reg clk, load;
reg [1:4] x;
wire [1:4] y;
wire z1;
initial //define clock
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
    end
initial
$monitor ("load=%b, x=%b, y=%b, z1=%b", load, x, y, z1);

//continued on next page

Figure 8.56  Test bench for the 4-bit shift register using behavioral modeling.
// apply input vectors

initial
begin
    #0 load = 1'b0; x = 4'b0000;
    #5 load = 1'b1; x = 4'b0101;
    #10 load = 1'b0;
    #30 load = 1'b1; x = 4'b1100;
    #20 load = 1'b0;
    #10 load = 1'b1; x = 4'b1111;
    #10 load = 1'b0;
    #30 $stop;
end

// instantiate the module
shift_reg_piso inst1 (  
    .clk(clk),  
    .load(load),  
    .x(x),  
    .y(y),  
    .z1(z1)  
);  
endmodule

Figure 8.56 (Continued)

load=0, x=UUU0, y=xxxx, z1=x
load=1, x=0010, y=xxxx, z1=x
load=1, x=0101, y=0101, z1=1
load=0, x=0101, y=0101, z1=1
load=0, x=0010, y=0010, z1=0
load=1, x=1100, y=0010, z1=0
load=1, x=1100, y=1100, z1=0
load=0, x=1100, y=1100, z1=0
load=0, x=1100, y=0110, z1=0
load=1, x=1111, y=0110, z1=0
load=0, x=1111, y=0111, z1=1
load=1, x=1111, y=0111, z1=1
load=1, x=1111, y=1111, z1=1
load=0, x=1111, y=1111, z1=1
load=0, x=1111, y=0111, z1=1

Figure 8.57 Outputs for the shift register of Figure 8.55.
Figure 8.58 Waveforms for the shift register of Figure 8.55.

//sum-of-products equation using if - else
module sop_eqn_if_else (x1, x2, x3, x4, z1);
  input x1, x2, x3, x4;
  output z1;
  reg z1;
  always @ (x1 or x2 or x3 or x4)
    begin
      if ((x1 && x2) || (x3 && x4))
        z1 = #2 1;
      else
        z1 = #2 0;
    end
endmodule

Figure 8.59 Module to illustrate the use of if...else conditional statements.
module sop_eqn_if_else_tb;

reg x1, x2, x3, x4;
wire z1;

//apply input vectors and display variables
initial
begin: apply_stimulus and display variables
  reg [4:0] invect;
  for (invect = 0; invect < 16; invect = invect + 1)
  begin
    {x1, x2, x3, x4} = invect [4:0];
    #10 $display "%x1x2x3x4 = %b, z1 = %b",
        (x1, x2, x3, x4), z1);
  end
end

//instantiate the module into the test bench
sop_eqn_if_else inst1 (  .x1(x1),  .x2(x2),  .x3(x3),  .x4(x4),  .z1(z1)
);
endmodule

Figure 8.60  Test bench for the behavioral module of Figure 8.59.
Figure 8.61  Outputs for the sum-of-products module of Figure 8.59.

Figure 8.62  Waveforms for the sum-of-products module of Figure 8.59.
//behavioral modulo-16 counter
module ctr_mod_16 (clk, rst_n, count);

input clk, rst_n;
output [3:0] count;

wire clk, rst_n;
reg [3:0] count;

//define counting sequence
always @ (posedge clk or negedge rst_n)
begin
  if (rst_n == 0)
    count <= 4'b0000;
  else
    count <= (count + 1) % 16;
end
endmodule

Figure 8.63 Behavioral module for a modulo-16 counter using conditional statements.

//modulo-16 counter test bench
module ctr_mod_16_tb;

reg clk, rst_n;
wire [3:0] count;

//display count
initial
$monitor ("count = %b", count);

//define reset
initial
begin
  #0 rst_n = 1'b0; //assert reset
  #5 rst_n = 1'b1; //deassert reset
end

//continued on next page

Figure 8.64 Test bench for the modulo-16 counter of Figure 8.63.
begin
  #0  clk = 1'b0;

  forever
    #10 clk = ~clk;
  end

//define length of simulation
initial
begin
  #330 $stop;
end

//instantiate the module into the test bench
ctr_mod_16 inst1 (  
  .clk(clk),
  .rst_n(rst_n),
  .count(count)  
);
endmodule

Figure 8.64  (Continued)

Page 409

count = 0000
count = 0001
count = 0010
count = 0011
count = 0100
count = 0101
count = 0110
count = 0111
count = 1000
count = 1001
count = 1010
count = 1011
count = 1100
count = 1101
count = 1110
count = 1111
count = 0000

Figure 8.65  Outputs for the modulo-16 counter of Figure 8.63.
Figure 8.66  Waveforms for the modulo-16 counter of Figure 8.63.

Figure 8.67  Behavioral model of a $D$ flip-flop.
/behavioral d_ff test bench

module d_ff_tb;

reg d, clk, set_n, rst_n;
wire q, q_n;

initial
$monitor ($time, " set_n=%b, rst_n=%b, d=%b, clk=%b, q = %b", set_n, rst_n, d, clk, q);

//define clock
initial
begin
  #5 clk = 1'b0;
  forever
    #5 clk = ~clk;
end

//define set and reset
initial
begin
  #0 rst_n = 1'b0;
  d     = 1'b0;
  clk   = 1'b0;
  set_n = 1'b1;
  #5 rst_n = 1'b1;

  #5 set_n = 1'b0;
  #5 set_n = 1'b1;

  #5 rst_n = 1'b0;
  #5 rst_n = 1'b1;
end

//define data input
initial
begin
  #32 d = 1'b0;
  #10 d = 1'b1;
  #10 d = 1'b0;

//continued on next page

Figure 8.68  Test bench for the D flip-flop module of Figure 8.67.
`d_ff inst1 (  
  .d(d),  
  .clk(clk),  
  .set_n(set_n),  
  .rst_n(rst_n),  
  .q(q),  
  .q_n(q_n)  
);`

endmodule

Figure 8.68  (Continued)

Page 413

Figure 8.70  Waveforms for the D flip-flop module of Figure 8.67.
//d flip-flop behavioral
module d_ff_bh (rst_n, clk, d, q, q_n);

input rst_n, clk, d;
output q, q_n;

wire rst_n, clk, d;
reg q;

assign q_n = ~q;

always @ (rst_n or posedge clk)
begin
  if (rst_n == 0)
    q = 1'b0;
  else q = d;
end
endmodule

Figure 8.71 Alternative method to design a D flip-flop.

//d_ff_bh test bench
module d_ff_bh_tb;

reg rst_n, clk, d;
wire q;

initial
$monitor ("rst_n=%b, clk=%b, d=%b, q=%b", rst_n, clk, d, q);

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

//continued on next page

Figure 8.72 Test bench for the module of Figure 8.71.
initial
begin
    #0 rst_n = 1'b0;
    #0 d    = 1'b0;
    #5 rst_n = 1'b1;

    #10 d = 1'b1;
    #10 d = 1'b1;
    #10 d = 1'b0;
    #10 d = 1'b0;
    #10 d = 1'b0;

    #10 $stop;
end

d_ff_bh inst1 (    
    .rst_n(rst_n),
    .clk(clk),
    .d(d),
    .q(q)
);
endmodule

Figure 8.72  (Continued)

Page 415

Figure 8.74  Waveforms for the $D$ flip-flop module of Figure 8.71.
//behavioral jkff
module jkff (clk, j, k, set_n, rst_n, q, q_n);

input clk, j, k, set_n, rst_n;
output q, q_n;
wire clk, j, k, set_n, rst_n;
reg q, q_n;

initial
  q = 1'b0;
always @ (posedge clk or negedge set_n or negedge rst_n)
begin
  if(~rst_n)
    begin
      q <= 1'b0;
      q_n <= 1'b1;
    end
  else if (~set_n)
    begin
      q <= 1'b1;
      q_n <= 1'b0;
    end
  else if (j==1'b0 && k==1'b0)
    begin
      q <= q;
      q_n <= q_n;
    end
  else if (j==1'b0 && k==1'b1)
    begin
      q <= 1'b0;
      q_n <= 1'b1;
    end
  else if (j==1'b1 && k==1'b0)
    begin
      q <= 1'b1;
      q_n <= 1'b0;
    end
  else if (j==1'b1 && k==1'b1)
    begin
      q <= q_n;
      q_n <= q;
    end
end
endmodule

Figure 8.75  Behavioral module for a JK flip-flop.
//jk flip-flop test bench
module jkff_tb;

reg clk, j, k, rst_n, set_n;
wire q, q_n;

initial //display outputs at simulation time
$monitor ($time, " q = %b", q);

initial
begin
  set_n = 1'b1;
  rst_n = 1'b0;
  j   = 1'b0;
  k   = 1'b0;
  clk = 1'b0;
  #3 rst_n = 1'b1;
  forever
    #5 clk = ~clk;
  end

initial
begin
  #10 j = 1'b1;
      k = 1'b0;  //set
  
  #10 j = 1'b1;
      k = 1'b1;  //toggle set
  
  #10 j = 1'b0;
      k = 1'b0;  //no change (set)
  
  #10 j = 1'b0;
      k = 1'b1;  //reset
  
  #10 $stop;
end

jkff inst1 (  
  .clk(clk),  
  .j(j),  
  .k(k),  
  .q(q),  
  .q_n(q_n)  
);
endmodule

Figure 8.76  Test bench for the JK flip-flop behavioral module of Figure 8.75.
Chapter 8  Behavioral Modeling  44

Page 419

<table>
<thead>
<tr>
<th></th>
<th>0 q = 0</th>
<th>18 q = 1</th>
<th>28 q = 0</th>
<th>38 q = 1</th>
<th>58 q = 0</th>
</tr>
</thead>
</table>

**Figure 8.77** Outputs for the JK flip-flop behavioral module of Figure 8.75.

**Figure 8.78** Waveforms for the JK flip-flop behavioral module of Figure 8.75.

Page 421

```verilog
//Behavioral even-odd counter
module ctr_evn_odd3 (rst_n, clk, y);

input rst_n, clk;
output [3:0] y;
wire rst_n, clk; //inputs are wire

reg [3:0] y; //outputs are reg
reg [3:0] d; //an internal next state

//reset counter and set y
always @ (posedge clk or negedge rst_n)
begin
  if (rst_n == 0)
    y <= 4'b0000;
  else
    y <= d;
end

//continued on next page
```

**Figure 8.80** Behavioral module for the even-odd counter of Example 8.21 using conditional statements only.
always @(y)
begin
  if ((y[3]==1 && y[1]==0) ||
    d[3] = 1'b1;
  else
    d[3] = 1'b0;
end

always @(y)
begin
  if ((y[2]==1 && y[1]==0) ||
    d[2] = 1'b1;
  else
    d[2] = 1'b0;
end

//Dy[1] = y[1]'
always @(y)
begin
  if (y[1]==0)
    d[1] = 1'b1;
  else
    d[1] = 1'b0;
end

always @(y)
begin
  if ((y[1]==0 && y[0]==1) ||
      (y[3]==0 && y[0]==1) ||
      (y[2]==0 && y[0]==1) ||
    d[0] = 1'b1;
  else
    d[0] = 1'b0;
end
endmodule

Figure 8.80 (Continued)
// test bench for even-odd counter
module ctr_even_odd3_tb;

reg rst_n, clk;
wire [3:0] y;

// display count
initial
$monitor ("count = %b", y);

// define clock
initial
begin
  clk = 1'b0;
  forever
    #5 clk = ~clk;
  end

// define length of simulation
initial
begin
  #3 rst_n = 1'b0;
  #1 rst_n = 1'b1;

  repeat (17) @ (posedge clk);
  #2;

  $stop;
end

// instantiate the module into the test bench
ctr_even_odd3 inst1 (  
  .rst_n(rst_n),
  .clk(clk),
  .y(y)
);
endmodule

Figure 8.81 Test bench for the even-odd counter of Figure 8.80.
Figure 8.82 Outputs for the even-odd counter of Figure 8.80.

Figure 8.83 Waveforms for the even-odd counter of Figure 8.80.
//behavioral 4-bit Gray code counter
module ctr_gray4_case (clk, rst_n, count);

input clk, rst_n;
output [3:0] count;

wire clk, rst_n;//inputs are wire
reg [3:0] count;//outputs are reg
reg [3:0] next_count;//define internal reg

//set next count
always @(posedge clk or negedge rst_n)
begin
  if (~rst_n)
    count <= 4'b0000;
  else
    count <= next_count;
end

//determine next count
always @ (count)
begin
  case (count)
    4'b0000 : next_count = 4'b0001;
    4'b0001 : next_count = 4'b0011;
    4'b0011 : next_count = 4'b0010;
    4'b0010 : next_count = 4'b0110;
    4'b0110 : next_count = 4'b0111;
    4'b0111 : next_count = 4'b0010;
    4'b0101 : next_count = 4'b0110;
    4'b0100 : next_count = 4'b1100;
    4'b1100 : next_count = 4'b1101;
    4'b1101 : next_count = 4'b1111;
    4'b1111 : next_count = 4'b1010;
    4'b1010 : next_count = 4'b1011;
    4'b1011 : next_count = 4'b1001;
    4'b1001 : next_count = 4'b1000;
    default : next_count = 4'b0000;
  endcase
end
endmodule

Figure 8.84  Behavioral module for a Gray code counter using the case statement.
module ctr_gray4_case_tb;

reg clk, rst_n;
wire[3:0] count;

initial
$monitor("count = %b", count);

//define reset
initial begin
#0 rst_n = 1'b0;
#5 rst_n = 1'b1;
end

//define clock
initial begin
#0 clk = 1'b0;
forever
#10 clk = ~clk;
end

//define length of simulation
initial begin
#330 $stop;
end

//instantiate the module into the test bench
ctr_gray4_case inst1(
.clk(clk),
.rst_n(rst_n),
.count(count)
);
endmodule

Figure 8.85 Test bench for the Gray code counter of Figure 8.84.
count = 0000  count = 0011  count = 0110  count = 0101  count = 1010
count = 0001  count = 0100  count = 1100  count = 1001  count = 1000
count = 0010  count = 1101  count = 0011  count = 0100  count = 0000
count = 0110  count = 1111  count = 1010  count = 1011  count = 1110

Figure 8.86 Outputs for the Gray code counter of Figure 8.84.

Figure 8.87 Waveforms for the Gray code counter of Figure 8.84.
//behavioral even-odd counter
module ctr_evn_odd (clk, rst_n, count);

input clk, rst_n;
output [3:0] count;

wire clk, rst_n; //inputs are wire
reg [3:0] count; //outputs are reg
reg [3:0] next_count; //define internal reg

always @ (posedge clk or negedge rst_n) //set next count
begin
  if (~rst_n) //rst_n is active-low
    count = 4\'b0000;
  else
    count = next_count;
end

always @ (count) //determine next count
begin
  case (count)
    4\'b0000 : next_count = 4\'b0010;
    4\'b0010 : next_count = 4\'b0100;
    4\'b0100 : next_count = 4\'b0110;
    4\'b0110 : next_count = 4\'b1000;
    4\'b1000 : next_count = 4\'b1010;
    4\'b1010 : next_count = 4\'b1100;
    4\'b1100 : next_count = 4\'b1110;
    4\'b1110 : next_count = 4\'b0001;
    4\'b0001 : next_count = 4\'b0011;
    4\'b0011 : next_count = 4\'b0101;
    4\'b0101 : next_count = 4\'b0111;
    4\'b0111 : next_count = 4\'b1001;
    4\'b1001 : next_count = 4\'b1011;
    4\'b1011 : next_count = 4\'b1101;
    4\'b1101 : next_count = 4\'b1111;
    default : next_count = 4\'b0000;
  endcase
end

endmodule

Figure 8.88  Behavioral module for an even-odd counter.
//ctr_evn_odd test bench
module ctr_evn_odd_tb;

reg clk, rst_n; //inputs are reg for tb
wire [3:0] count; //outputs are wire for tb

initial
$monitor("count = %b", count);

initial //define clk input
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
    end

initial //define reset input
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
    #320 $stop;
end

//instantiate the module into the test bench
ctr_evn_odd inst1 (  
    .clk(clk),  
    .rst_n(rst_n),  
    .count(count)  
);
endmodule

Figure 8.89 Test bench for the even-odd counter of Figure 8.88.

count = 0000     count = 0011  
count = 0010     count = 0101  
count = 0100     count = 0111  
count = 0110     count = 1001  
count = 1000     count = 1011  
count = 1010     count = 1101  
count = 1100     count = 1111  
count = 1110     count = 0000  
count = 0001

Figure 8.90 Outputs for the even-odd counter of Figure 8.88.
Figure 8.91  Waveforms for the even-odd counter of Figure 8.88.

Figure 8.93  Behavioral module for the 4-function ALU of Example 8.24.
module alu4_tb;

reg [3:0] a, b;
reg [1:0] opcode;
wire [7:0] z;

//display variables
initial
$monitor ("a=%b, b=%b, opcode=%b, result=%b", a, b, opcode, z);

//apply input vectors
initial
begin
//add operation
#0 a=4'b0001; b=4'b0001; opcode=2'b00; //sum=2
#10 a=4'b0010; b=4'b1101; opcode=2'b00; //sum=15(f)
#10 a=4'b1111; b=4'b1111; opcode=2'b00; //sum=30(1e)

//subtract operation
#10 a=4'b1000; b=4'b0100; opcode=2'b01; //difference=4
#10 a=4'b1111; b=4'b0101; opcode=2'b01; //difference=10(a)
#10 a=4'b1110; b=4'b0011; opcode=2'b01; //difference=11(b)
#10 a=4'b0100; b=4'b1000; opcode=2'b01; //difference=-4(fc)
#10 a=4'b0110; b=4'b1111; opcode=2'b01; //difference=-9(f7)

//multiply operation
#10 a=4'b0100; b=4'b0111; opcode=2'b10; //product=28(1c)
#10 a=4'b0101; b=4'b0011; opcode=2'b10; //product=15(f)
#10 a=4'b1111; b=4'b1111; opcode= 'b10; //product=225(e1)
//continued on next page

Figure 8.94 Test bench for the 4-function ALU of Figure 8.93.
//divide operation
#10 a=4'b1111; b=4'b0101; opcode=2'b11; //quotient=3

#10 a=4'b1100; b=4'b0011; opcode=2'b11; //quotient=4

#10 a=4'b1110; b=4'b0010; opcode=2'b11; //quotient=7

#10 a=4'b0011; b=4'b1100; opcode=2'b11; //quotient=0

#10 $stop;
end

//instantiate the module into the test bench
alu4 inst1 (.a(a),
    .b(b),
    .opcode(opcode),
    .z(z)
);
endmodule

Figure 8.94 (Continued)

Page 434

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>opcode</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0001</td>
<td>00</td>
<td>00000010</td>
</tr>
<tr>
<td>0010</td>
<td>1101</td>
<td>00</td>
<td>00001111</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>00</td>
<td>00011110</td>
</tr>
<tr>
<td>1000</td>
<td>0100</td>
<td>01</td>
<td>00000100</td>
</tr>
<tr>
<td>1111</td>
<td>0101</td>
<td>01</td>
<td>00000101</td>
</tr>
<tr>
<td>1110</td>
<td>0011</td>
<td>01</td>
<td>00000111</td>
</tr>
<tr>
<td>0100</td>
<td>1000</td>
<td>10</td>
<td>00011100</td>
</tr>
<tr>
<td>0110</td>
<td>1111</td>
<td>01</td>
<td>00010111</td>
</tr>
<tr>
<td>0100</td>
<td>0111</td>
<td>10</td>
<td>00001100</td>
</tr>
<tr>
<td>0101</td>
<td>0011</td>
<td>10</td>
<td>00001111</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>10</td>
<td>00010011</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>11</td>
<td>00000011</td>
</tr>
<tr>
<td>1100</td>
<td>0011</td>
<td>11</td>
<td>00000100</td>
</tr>
<tr>
<td>1110</td>
<td>0010</td>
<td>11</td>
<td>00000111</td>
</tr>
<tr>
<td>0011</td>
<td>1100</td>
<td>11</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Figure 8.95 Outputs for the 4-function ALU of Figure 8.93.
Figure 8.96 Waveforms for the 4-function ALU of Figure 8.93.

Figure 8.98 Behavioral module for the Moore synchronous sequential machine of Figure 8.97.
//determine outputs
always @(y)
begin
  if (y == state_d) //== specifies logical
    z1 = 1'b1; //equality or compare
  else
    z1 = 1'b0;
  if (y == state_e)
    z2 = 1'b1;
  else
    z2 = 1'b0;
end

//determine next state
always @(y or x1)
begin
  case (y) //case is a multiple-way
    state_a: //conditional branch
      if (x1) //if y = state_a, then
        next_state = state_c; //do if . . . else
      else
        next_state = state_b;
    state_b:
      if (x1)
        next_state = state_d;
      else
        next_state = state_e;
    state_c:
      if (x1)
        next_state = state_e;
      else
        next_state = state_d;
    state_d:next_state = state_a;
    state_e:next_state = state_a;
    default: next_state = state_a;
  endcase
end
endmodule

Figure 8.98 (Continued)
module moore_ssm_tb;

reg clk, rst_n, x1;
wire [2:0] y;
wire z1, z2;

initial
$monitor ("x1 = %b, state = %b, z1 = %b, z2 = %b",
   x1, y, z1, z2);

//define clock
initial
begin
  clk = 1'b0;
  forever
    #10 clk = ~clk;
end

//define input sequence
initial
begin
  #0  rst_n = 1'b0; //rst to state_a (001)
  #15 rst_n = 1'b1;

  x1 = 1'b0;
  @ (posedge clk) //go to state_b (101)

  x1 = 1'b1;
  @ (posedge clk) //go to state_d (000); assert z1

  x1 = 1'b0;
  @ (posedge clk) //go to state_a (001)

  x1 = 1'b1;
  @ (posedge clk) //go to state_c (011)

  x1 = 1'b1;
  @ (posedge clk) //go to state_e (111); assert z2

  x1 = 1'b0;
  @ (posedge clk) //go to state_a (001)
  //continued on next page

Figure 8.99  Test bench for the Moore synchronous sequential machine of Figure 8.98.
x1 = 1'b0;
@ (posedge clk) //go to state_b (101)

x1 = 1'b0;
@ (posedge clk) //go to state_e (111); assert z2

x1 = 1'b0;
@ (posedge clk) //go to state_a (001)

x1 = 1'b1;
@ (posedge clk) //go to state_c (011)

x1 = 1'b0;
@ (posedge clk) //go to state_d (000); assert z1

x1 = 1'b0;
@ (posedge clk) //go to state_a (001)

#10 $stop;
end

moore_ssm inst1 ( //instantiate the module
  .clk(clk),
  .rst_n(rst_n),
  .x1(x1),
  .y(y),
  .z1(z1),
  .z2(z2)
);
endmodule

Figure 8.99  (Continued)

Page 439

Figure 8.100  Waveforms for the Moore synchronous sequential machine of Figure 8.98.
//behavioral mealy ssm
module mealy_ssm (clk, rst_n, x1, x2, y1, z1);
input clk, rst_n, x1, x2;
output y1;
output z1;
reg y1, next_state;
reg z1;

parameter state_a = 1'b0, //assign state codes
        state_b = 1'b1;

always @ (posedge clk) //set next state
begin
        if (~rst_n)
                y1 <= state_a;
        else
                y1 <= next_state;
end

always @(x1) //determine outputs
begin
        if ((y1 == state_a) && (x1 == 1'b1))
                z1 = 1'b1;
        else
                z1 = 1'b0;
end

always @ (y1 or x1 or x2) //determine next state
begin
        case (y1)
                state_a: 
                        if (x1)
                                next_state = state_b;
                        else
                                next_state = state_a;
                state_b: 
                        if (x2)
                                next_state = state_a;
                        else
                                next_state = state_b;
                default: next_state = state_a;
        endcase
end
endmodule

Figure 8.102  Behavioral module for the Mealy synchronous sequential machine of Figure 8.101.
// test bench for mealy ssm
module mealy_ssm_tb;

reg clk, rst_n, x1, x2;
wire y1, z1;

// display variables
initial
$monitor ("x1=%b, x2=%b, state=%b, z1=%b",
  x1, x2, y1, z1);

// define clock
initial
begin
  clk = 1'b0;
  forever
    #10 clk = ~clk;
  end

initial               // define input sequence
begin
  #0  rst_n = 1'b0;   // rst to state_a
  #15 rst_n = 1'b1;

  x1 = 1'b0;  x2 = 1'b0;
  @ (posedge clk)  // go to state_a

  x1 = 1'b1;  x2 = 1'b0;
  @ (posedge clk)  // assert z1; go to state_b

  x1 = 1'b0;  x2 = 1'b0;
  @ (posedge clk)  // go to state_b

  x1 = 1'b0;  x2 = 1'b1;
  @ (posedge clk)  // go to state_a

  x1 = 1'b0;  x2 = 1'b0;
  @ (posedge clk)  // go to state_a

  x1 = 1'b0;  x2 = 1'b0;
  @ (posedge clk)  // assert z1; go to state_b

  #10 $stop;
end

// continued on next page

Figure 8.103 Test bench for the Mealy synchronous sequential machine of Figure 8.101.
// instantiate the module into the test bench
mealy_ssm inst1 (
  .clk(clk),
  .rst_n(rst_n),
  .x1(x1),
  .x2(x2),
  .y1(y1),
  .z1(z1)
);
endmodule

Figure 8.103  (Continued )

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>state</th>
<th>z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 8.104  Outputs for the Mealy synchronous sequential machine of Figure 8.101.

Figure 8.105  Waveforms for the Mealy synchronous sequential machine of Figure 8.101.
//behavioral moore ssm2
module moore_ssm2 (clk, rst_n, x1, x2, x3, y, 
                   z1, z2, z3, z4, z5);

//specify inputs and outputs
input clk, rst_n, x1, x2, x3;

output [2:0] y; //y is an array of 3 bits
output z1, z2, z3, z4, z5;

reg [2:0] y, next_state; //must be reg for always
wire z1, z2, z3, z4, z5;

//assign state codes
parameter state_a = 3'b000, //param defines a constant
                 state_b = 3'b001, //state names must have
                 state_c = 3'b010, //at least 2 characters
                 state_d = 3'b011,
                 state_e = 3'b100,
                 state_f = 3'b101;

//set next state
always @ (posedge clk) //reset synched to posedge clk
begin
    if (~rst_n) //if (~rst_n) is true,
        y <= state_a; //y <= state_a
    else
        y <= next_state;
end

//determine outputs
assign z1 = (~y[2] & ~y[1] & y[0] & ~clk);
assign z2 = (~y[2] & y[1] & ~y[0] & ~clk);
assign z3 = (~y[2] & y[1] & y[0] & ~clk);
assign z4 = ( y[2] & ~y[1] & ~y[0] & ~clk);
assign z5 = ( y[2] & ~y[1] & y[0] & ~clk);

//continued on next page

Figure 8.108 Behavioral module for the Moore sequential machine of Figure 8.107.
//determine next state
always @ (y or x1 or x2 or x3)
begin
    case (y) //case is a multiple-way state
        state_a: //conditional branch
            if (x1==0 & x2==0 & x3==1) //if y = state_a, then
                next_state = state_b; //do if-else if-else
            else if (x1==0 & x2==1 & x3==0)
                next_state = state_c;
            else if (x1==0 & x2==1 & x3==1)
                next_state = state_d;
            else if (x1==1 & x2==0 & x3==0)
                next_state = state_e;
            else if (x1==1 & x2==0 & x3==1)
                next_state = state_f;
            else
                next_state = state_a;
        state_b: next_state = state_a;
        state_c: next_state = state_a;
        state_d: next_state = state_a;
        state_e: next_state = state_a;
        state_f: next_state = state_a;
        default next_state = state_a;
    endcase
end
endmodule

Figure 8.108 (Continued)

Page 448

//test bench for moore ssm
module moore_ssm_tb;

reg clk, rst_n, x1;
wire [2:0] y;
wire z1, z2;

initial
$monitor ("x1 = %b, state = %b, z1 = %b, z2 = %b",
    x1, y, z1, z2);

//continued on next page

Figure 8.109 Test bench for the Moore synchronous sequential machine of Figure 8.108.
//define clock
initial
begin
  clk = 1'b0;
  forever
    #10 clk = ~clk;
end

//define input sequence
initial
begin
  #0 rst_n = 1'b0; //rst to state_a (001)
  #15 rst_n = 1'b1;
  x1=1'b0; x2=1'b0; x3=1'b1;
  @ (posedge clk) //go to state_b (001); assert z1
  x1=1'b0; x2=1'b1; x3=1'b0;
  @ (posedge clk) //go to state_a (000)
  x1=1'b0; x2=1'b1; x3=1'b1;
  @ (posedge clk) //go to state_c (010); assert z2
  x1=1'b0; x2=1'b0; x3=1'b0;
  @ (posedge clk) //go to state_a (000)
  x1=1'b1; x2=1'b0; x3=1'b1;
  @ (posedge clk) //go to state_d (011); assert z3
  x1=1'b0; x2=1'b0; x3=1'b0;
  @ (posedge clk) //go to state_a (000)
  x1=1'b1; x2=1'b0; x3=1'b1;
  @ (posedge clk) //go to state_e (100); assert z4
  x1=1'b0; x2=1'b0; x3=1'b1;
  @ (posedge clk) //go to state_a (000)
  #10 $stop;
end

//instantiate the behav module into the test bench
moore_ssm2 inst1 ( 
  .clk(clk),
  .rst_n(rst_n),
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .y(y),
  .z1(z1),
  .z2(z2),
  .z3(z3),
  .z4(z4),
  .z5(z5)
);
endmodule

Figure 8.109  (Continued)
Figure 8.110  Waveforms for the Moore synchronous sequential machine of Figure 8.108
/behavioral module for a mealy ssm
module mealy_ssm2 (clk, rst_n, x1, y, z1);

input clk, rst_n, x1;
output [1:0] y;
output z1;

reg [1:0] y, next_state;
wire z1;

parameter state_a = 2'b00, //assign state codes
     state_b = 2'b01,
     state_c = 2'b11,
     state_d = 2'b10;

always @(posedge clk) //set next state
begin
  if (~rst_n)
    y <= state_a;
  else
    y <= next_state;
end

assign z1 = ((y[1]) && (~y[0]) && (x1) && (clk));

always @(y or x1) //determine next state
begin
  case (y)
    state_a:
      if (x1)
        next_state = state_b;
      else
        next_state = state_a;
    state_b:
      if (x1)
        next_state = state_b;
      else
        next_state = state_c;
    state_c:
      if (x1)
        next_state = state_b;
      else
        next_state = state_d;
  //continued on next page
endcase

Figure 8.112 Behavioral module for the Mealy machine of Example 8.28.
```verilog
define the state transitions:

\[
\text{state\_d:}
\begin{align*}
\text{if (x1)} & : \text{next\_state} = \text{state\_b}; \\
\text{else} & : \text{next\_state} = \text{state\_a}; \\
\text{default: next\_state} & = \text{state\_a};
\end{align*}
\]

endcase
end
endmodule

Page 452

// test bench for mealy ssm2
module mealy_ssm2_tb;

reg clk, rst_n, x1;
wire [1:0] y;
wire z1;

// display variables
initial
$monitor ("x1=%b, state=%b, z1=%b", x1, y, z1);

// define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

// define input sequence
initial
begin
    #0 rst_n = 1'b0; // rst to state_a
    #15 rst_n = 1'b1;

    x1 = 1'b0; @ (posedge clk) // go to state a
    x1 = 1'b1; @ (posedge clk) // go to state b
    x1 = 1'b1; @ (posedge clk) // go to state b

// continued on next page
```

Figure 8.112  (Continued)

Figure 8.113  Test bench for the Mealy machine of Figure 8.112.
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_d} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_d} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_d} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_d} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_d} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_d} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_d} \]
\[ x_1 = 1'b1; @ (posedge \text{clk}) \quad \text{//go to state_b} \]
\[ x_1 = 1'b0; @ (posedge \text{clk}) \quad \text{//go to state_c} \]
\#10 $\text{stop;}$

eady\_ssm2\ inst\_1 \quad \text{//instantiate the module}
  \. \text{clk}(\text{clk}),
  \. \text{rst}_n(\text{rst}_n),
  \. x_1(x_1),
  \. y(y),
  \. z_1(z_1)
  \);
endmodule

Figure 8.113  (Continued)

Figure 8.114  Waveforms for the Mealy synchronous sequential machine of Figure 8.112.
// behavioral asynchronous sequential machine
module asm8 (x1, rst_n, ye, z1, z2);

input x1, rst_n;
output [1:0] ye;
output z1, z2;

wire x1, rst_n;
reg [1:0] ye, next_state; // must be reg for always
reg z1, z2;

// assign state codes
parameter state_a = 2'b00, // parameter defines a constant
state_b = 2'b01,
state_c = 2'b11,
state_d = 2'b10;

always @ (x1 or rst_n) // latch next state
begin
  if (~rst_n)
    ye <= state_a;
  else
    ye <= next_state;
end

always @ (ye) // define outputs for each state
begin
  // sensitivity list must contain
  if (ye == state_a) // the state variables
    begin
      z1 = 1'b0;
      z2 = 1'b0;
    end

  if (ye == state_b) // == is logical equality
    begin
      z1 = 1'b1;
      z2 = 1'b0;
    end

  if (ye == state_c)
    begin
      z1 = 1'b1;
      z2 = 1'b1;
    end

  // continued on next page

Figure 8.117 Behavioral module for the asynchronous sequential machine of Example 8.29.
if (ye == state_d)
begin
    z1 = 1'b0;
    z2 = 1'b1;
end
end

//determine next state
always @(x1)
begin
    case (ye)
        state_a:
            if (x1)
                next_state = state_b;
            else
                next_state = state_a;

        state_b:
            if (~x1)
                next_state = state_c;
            else
                next_state = state_b;

        state_c:
            if (x1)
                next_state = state_d;
            else
                next_state = state_c;

        state_d:
            if (~x1)
                next_state = state_a;
            else
                next_state = state_d;

        default:
            next_state = state_a;
endcase
end
endmodule
In Figure 8.117, a test bench for the asynchronous sequential machine is shown.

```verilog
module asm8_tb;

reg x1, rst_n;       //inputs are reg in test bench
wire [1:0] ye;      //outputs are wire in test bench
wire z1, z2;

initial begin
  #0 rst_n = 1'b0;
  x1 = 1'b0;
  #10 rst_n = 1'b1;
  #10 x1 = 1'b1; //state_a to state_b; assert z1
  #10 x1 = 1'b0; //state_b to state_c; assert z1, z2
  #10 x1 = 1'b1; //state_c to state_d; deassert z1; assert z2
  #10 x1 = 1'b0; //state_d to state_a; no outputs
  #10 x1 = 1'b1; //state_a to state_b; assert z1
  #10 x1 = 1'b0; //state_b to state_c; assert z1, z2
  #10 x1 = 1'b1; //state_c to state_d; deassert z1; assert z2
  #10 x1 = 1'b0; //state_d to state_a; no outputs
  #40 $stop;
end

//instantiate the module into the test bench
asm8 inst1 (
  .x1(x1),
  .rst_n(rst_n),
  .ye(ye),
  .z1(z1),
  .z2(z2)
);
endmodule
```

Figure 8.118 Test bench for the asynchronous sequential machine of Figure 8.117.
Figure 8.119  Waveforms for the asynchronous sequential machine of Figure 8.117.

Figure 8.121  Behavioral module for the 16-function ALU of Example 8.30.
sra_op = 4'd1000,
srl_op = 4'd1001,
sla_op = 4'd1010,
sll_op = 4'd1011,
ror_op = 4'd1100,
rol_op = 4'd1101,
inc_op = 4'd1110,
dec_op = 4'd1111;

//execute operations
always @(a or b or opcode)
begin
    case (opcode)
    add_op : result = a + b;
    sub_op : result = a - b;
    mul_op : result_mul = a * b;
    div_op : result = a / b;
    and_op : result = a & b;
    or_op  : result = a | b;
    xor_op : result = a ^ b;
    not_op : result = ~a;
    sra_op : result = {a[7], a[7], a[6], a[5], a[4], a[3], a[2], a[1]};
    srl_op : result = a >> 1;
    sla_op : result = {a[7], a[5], a[4], a[3], a[2], a[1], a[0], 1'b0};
    sll_op : result = a << 1;
    ror_op : result = {a[0], a[7], a[6], a[5], a[4], a[3], a[2], a[1]};
    rol_op : result = {a[6], a[5], a[4], a[3], a[2], a[1], a[0], a[7]};
    inc_op : result = a + 1;
    dec_op : result = a - 1;
    default : result = 0;
    endcase
end
endmodule

Figure 8.121 (Continued)
module alu_16fctn_tb;

reg [7:0] a, b; //inputs are reg for test bench
reg [3:0] opcode;
wire [7:0] result; //outputs are wire for test bench
wire [15:0] result_mul;

initial $monitor("a=%h, b=%h, opcode=%h, rslt=%h, rslt_mul=%h",
            a, b, opcode, result, result_mul);

initial begin
    //add op ------------------------------------------
    #0 a = 8'b00000000;
b = 8'b00000000;
opcode = 4'b0000;

    #10 a = 8'b01100010; //a = 98d = 62h
    b = 8'b00011100; //b = 28d = 1Ch
    opcode = 4'b0000; //result = 126d = 007eh

    #10 a = 8'b11111111; //a = 255d = ffh
    b = 8'b11111111; //b = 255d = ffh
    opcode = 4'b0000; //result = 510d = 01feh

    //sub op ------------------------------------------
    #10 a = 8'b10000000; //a = 128d = 80h
    b = 8'b01100011; //b = 99d = 63h
    opcode = 4'b0001; //result = 29d = 001dh

    #10 a = 8'b11111111; //a = 255d = ffh
    b = 8'b00001111; //b = 15d = fh
    opcode = 4'b0001; //result = 240d = 00f0h

    //mul op ------------------------------------------
    #10 a = 8'b00011001; //a = 25d = 19h
    b = 8'b00011001; //b = 25d = 19h
    opcode = 4'b0010; //result_mul = 625d = 0271h

    #10 a = 8'b10000100; //a = 132d = 84h
    b = 8'b11111010; //b = 250d = fah
    opcode = 4'b0010; //result_mul = 33000d = 80e8h

    //continued on next page

Figure 8.122 Test bench for the 16-function ALU of Figure 8.121.
#10 a = 8'b11110000; //a = 240d = f0h
b = 8'b00010100; //b = 20d = 14h
opcode = 4'b0010; //result_mul = 4800d = 12c0h

//div op ------------------------------------------
#10 a = 8'b11110000; //a = 240d = f0h
b = 8'b00001111; //b = 15d = 0fh
opcode = 4'b0011; //result = 16d = 10h
#10 a = 8'b00010000; //a = 16d = 10h
b = 8'b00010010; //b = 18d = 12h
opcode = 4'b011; //result = 0d = 0h

//and op ------------------------------------------
#10 a = 8'b11111111;
b = 8'b11110000;
opcode = 4'b0100; //result = 11110000

//or op -------------------------------------------
#10 a = 8'b10101010;
b = 8'b11110101;
opcode = 4'b0101; //result = 11111111

//xor op ------------------------------------------
#10 a = 8'b00001111;
b = 8'b10101010;
opcode = 4'b0110; //result = 10100101

//not op ------------------------------------------
#10 a = 8'b00001111;
opcode = 4'b0111; //result = 11110000

//sra op ------------------------------------------
#10 a = 8'b10001110;
opcode = 4'b1000; //result = 11000111

//srl op ------------------------------------------
#10 a = 8'b11110011;
opcode = 4'b1001; //result = 01111001

//sla op ------------------------------------------
#10 a = 8'b10001111;
opcode = 4'b1010; //result = 10011110

//continued on next page

Figure 8.122 (Continued)
Figure 8.122  (Continued)
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>opcode</th>
<th>rslt</th>
<th>rslt_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>00000000</td>
<td>xxxx</td>
</tr>
<tr>
<td>62</td>
<td>1c</td>
<td>0</td>
<td>01111110</td>
<td>7e</td>
</tr>
<tr>
<td>ff</td>
<td>ff</td>
<td>0</td>
<td>11111110</td>
<td>fe</td>
</tr>
<tr>
<td>80</td>
<td>63</td>
<td>1</td>
<td>00011101</td>
<td>1d</td>
</tr>
<tr>
<td>ff</td>
<td>0f</td>
<td>1</td>
<td>11110000</td>
<td>7e</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>2</td>
<td>11110000</td>
<td>f0</td>
</tr>
<tr>
<td>84</td>
<td>fa</td>
<td>2</td>
<td>11110000</td>
<td>f0</td>
</tr>
<tr>
<td>f0</td>
<td>0f</td>
<td>3</td>
<td>00010000</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>3</td>
<td>00000000</td>
<td>00</td>
</tr>
<tr>
<td>ff</td>
<td>f0</td>
<td>4</td>
<td>11110000</td>
<td>f0</td>
</tr>
<tr>
<td>aa</td>
<td>f5</td>
<td>5</td>
<td>11111111</td>
<td>f0</td>
</tr>
<tr>
<td>0f</td>
<td>aa</td>
<td>6</td>
<td>10100101</td>
<td>a5</td>
</tr>
<tr>
<td>0f</td>
<td>aa</td>
<td>7</td>
<td>11110000</td>
<td>f0</td>
</tr>
<tr>
<td>8e</td>
<td>aa</td>
<td>8</td>
<td>11000111</td>
<td>c7</td>
</tr>
<tr>
<td>f3</td>
<td>aa</td>
<td>9</td>
<td>01111001</td>
<td>79</td>
</tr>
<tr>
<td>8f</td>
<td>aa</td>
<td>a</td>
<td>10011110</td>
<td>9e</td>
</tr>
<tr>
<td>77</td>
<td>aa</td>
<td>b</td>
<td>11101110</td>
<td>ee</td>
</tr>
<tr>
<td>55</td>
<td>aa</td>
<td>c</td>
<td>10101010</td>
<td>aa</td>
</tr>
<tr>
<td>55</td>
<td>aa</td>
<td>d</td>
<td>10101010</td>
<td>aa</td>
</tr>
<tr>
<td>fd</td>
<td>aa</td>
<td>e</td>
<td>11111110</td>
<td>fe</td>
</tr>
<tr>
<td>fe</td>
<td>aa</td>
<td>f</td>
<td>11111110</td>
<td>fd</td>
</tr>
</tbody>
</table>

Figure 8.123  Outputs for the 16-function ALU of Figure 8.121.
Figure 8.124  Waveforms for the 16-function ALU of Figure 8.121.
module moore_ssm3 (clk, rst_n, x1, y, z1);

input clk, rst_n, x1;
output [2:0] y; // y is an array of 3 bits
output z1;

wire clk, rst_n, x1;
reg [2:0] y, next_state; // outputs are reg in behavioral
reg z1;

// assign state codes
parameter state_a = 3'b000, // parameter defines a constant
    state_b = 3'b001, // state names must have at
    state_c = 3'b101, // least two characters
    state_d = 3'b011,
    state_e = 3'b111,
    state_f = 3'b010,
    state_g = 3'b110;

// set next state
always @(posedge clk or negedge rst_n)
begin
    if (~rst_n) // if (~rst_n) is true (1),
        y <= #3 state_a; // then y <= state_a #3 later
    else
        y <= #3 next_state;
end

// continued on next page

Figure 8.126 Behavioral module for the Moore machine of Example 8.31.
//determine output
always @ (y or clk)
begin
  if (y == state_g)
    begin
      if (~clk)
        z1 = 1'b1;
      else
        z1 = 1'b0;
    end
  else
    z1 = 1'b0;
end

//determine next state
always @ (y or x1)
begin
  case (y) //case is a multiple-way
    state_a: //conditional branch.
      if (x1) //if y = state_a, then
        next_state = state_c; //execute if . . . else
      else
        next_state = state_b;
    state_b: next_state = state_d;
    state_c:
      if (x1)
        next_state = state_e;
      else
        next_state = state_d;
    state_d: next_state = state_f;
    state_e:
      if (x1)
        next_state = state_g;
      else
        next_state = state_f;
    state_f:
      next_state = state_a;
    state_g: next_state = state_a;
    default: next_state = state_a;
  endcase
end
endmodule

Figure 8.126 (Continued)
module moore_ssm3_tb;

reg clk, x1, rst_n;
wire [2:0] y;
wire z1;

//define clock
initial begin
  clk = 1'b0;
  forever //forever continually executes
    #10 clk = ~clk; //the procedural statement
  end

//define input vectors
initial begin
  #0 x1 = 1'b0;
  rst_n = 1'b0;
  #5 rst_n = 1'b1;

  @ (posedge clk) //if x1=0 in state_a, go to state_b (001)
    x1 = $random; @ (posedge clk) //if x1=0/1 in state_b, go to state_d (011)

    x1 = $random; @ (posedge clk) //if x1=0/1 in state_d, go to state_f (010)

    x1 = $random; @ (posedge clk) //if x1=0/1 in state_f, go to state_a (000)

    x1 = 1'b1; @ (posedge clk) //if x1=1 in state_a, go to state_c (101)

    x1 = 1'b1; @ (posedge clk) //if x1=1 in state_a, go to state_e (111)

    x1 = 1'b1; @ (posedge clk) //if x1=1 in state_e, go to state_g (110); z1=1

  //continued on next page

Figure 8.127  Test bench for the Moore synchronous sequential machine of Figure 8.126.
x1 = $random; @(posedge clk)
  // if x1=0/1 in state_g, go to state_a (000)

x1 = 1'b1; @(posedge clk)
  // if x1=1 in state_a, go to state_c (101)

x1 = 1'b0; @(posedge clk)
  // if x1=0 in state_c, go to state_d (011)

x1 = $random; @(posedge clk)
  // if x1=0/1 in state_d, go to state_f (010)

x1 = $random; @(posedge clk)
  // if x1=0/1 in state_f, go to state_a (000)

x1 = 1'b1; @(posedge clk)
  // if x1=1 in state_a, go to state_c (101)

x1 = 1'b1; @(posedge clk)
  // if x1=1 in state_c, go to state_e (111)

x1 = 1'b0; @(posedge clk)
  // if x1=0 in state_e, go to state_f (010)

x1 = $random; @(posedge clk)
  // if x1=0/1 in state_f, go to state_a (000)

#150 $stop;

end

// instantiate the module into the test bench
moore_ssm3 inst1 (.
  clk(clk),
  rst_n(rst_n),
  x1(x1),
  y(y),
  z1(z1)
);

endmodule

Figure 8.127 (Continued)
//apply input vectors and display variables
initial
begin: apply_stimulus
  reg [4:0] invect;
  for (invect = 0; invect < 16; invect = invect + 1)
  begin
    {x1, x2, x3, x4} = invect [4:0];
    #10 $display("{x1x2x3x4} = %b, z1 = %b",
               {x1, x2, x3, x4}, z1);
  end
end

Figure 8.129  Code segment using a for loop.

//example of a while loop.
//count the number of 1s in a 16-bit register
module while_loop;

integer count;

initial
begin: number_of_1s
  reg [16:0] reg_a;

  count = 0;

  reg_a = 16'haabb; //set reg_a to a known value

  while (reg_a) //do while reg_a contains 1s
  begin
    if (reg_a[0]) //check low-order bit
      count = count + 1; //if true, add one to count
    reg_a = reg_a >> 1; //shift right 1 bit position
    $display("count = %d", count);
  end
end
endmodule

Figure 8.130  Module to illustrate the use of the while construct.
count = 1  count = 6
count = 2  count = 7
count = 2  count = 7
count = 3  count = 8
count = 4  count = 8
count = 5  count = 9
count = 5  count = 9
count = 6  count = 10

Figure 8.131  Outputs for the module of Figure 8.129 illustrating the use of the while loop.

```verilog
//example of the repeat keyword
module repeat_example;

integer count;

initial
begin
  count = 0;
  repeat (16)
  begin
    $display("count = %d", count);
    count = count + 1;
  end
endmodule
```

Figure 8.132  Module to illustrate the use of the repeat construct.

count = 0  count = 8
count = 1  count = 9
count = 2  count = 10
count = 3  count = 11
count = 4  count = 12
count = 5  count = 13
count = 6  count = 14
count = 7  count = 15

Figure 8.133  Outputs for the module of Figure 8.132.
//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
    end

//define length of simulation
initial
    #100 $finish;

Figure 8.134 Clock generation using a \texttt{forever} statement.

module seq_block (x1, x2, x3, z1, z2, z3, z4, z5, z6);
input x1, x2, x3;
output z1, z2, z3;
output [2:0] z4, z5, z6;
reg z1, z2, z3;
reg [2:0] z4, z5, z6;

always @ (x1 or x2 or x3)
begin
    z1 = #2 (x1 & x2) | x3;
    z2 = #3 (x1 | x2) & x3;
    z3 = #4 ~(x1 ^ x2 ^ x3);
    z4 = {x1, x2, x3};
    z5 = {x3, x1, x2};
    z6 = {x2, x3, x1};
end
endmodule

Figure 8.135 Module to illustrate a sequential block.
// test bench for sequential block
module seq_block_tb;

reg x1, x2, x3;
wire z1, z2, z3;
wire [2:0] z4, z5, z6;

// display variables
initial
$monitor ("x1 x2 x3=%b, z1=%b,z2=%b,z3=%b,z4=%b,z5=%b,z6=%b",
  {x1, x2, x3}, z1, z2, z3, z4, z5, z6);

// apply input vectors
initial
begin
  #0 x1 = 1'b0; x2 = 1'b0; x3 = 1'b1;
  #10 x1 = 1'b0; x2 = 1'b1; x3 = 1'b0;
  #10 x1 = 1'b1; x2 = 1'b0; x3 = 1'b1;
  #10 x1 = 1'b1; x2 = 1'b1; x3 = 1'b1;
  #20 $stop;
end

// instantiate the module into the test bench
seq_block inst1 (  
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .z1(z1),
  .z2(z2),
  .z3(z3),
  .z4(z4),
  .z5(z5),
  .z6(z6)
);
endmodule

Figure 8.136  Test bench for the module of Figure 8.135.
Figure 8.137  Outputs for the module of Figure 8.135.

Figure 8.138  Waveforms for the module of Figure 8.135.
//parallel block
module par_block (z1, z2, z3);

output z1, z2, z3;
reg z1, z2, z3;

//initialize variables
initial
begin
  #0 z1 = 1'b0;
  z2 = 1'b0;
  z3 = 1'b0;
end

initial
don't care
fork
  #10 z1 = 1'b1;
  #30 z2 = 1'b1;
  #20 z1 = 1'b0;
  #40 z2 = 1'b0;
  #60 z3 = 1'b0;
  #50 z3 = 1'b1;
join
endmodule

Figure 8.139  Module to illustrate the use of the keywords fork ... join to form a parallel block.

Figure 8.140  Waveforms for the module of Figure 8.139.
//behavioral d flip-flop
//to illustrate assign . . . deassign
module assign_deassign2 (rst_n, clk, d, q, q_n);

input rst_n, clk, d;
output q, q_n;

wire rst_n, clk, d;
reg q, q_n;

always @ (posedge clk)
begin
    q <= d;
    q_n <= ~d;
end

always @ (rst_n)
begin
    if (rst_n == 0)
        begin
            assign q = 1'b0;
            assign q_n = 1'b1;
        end
    else
        begin
            deassign q;
            deassign q_n;
        end
end
endmodule

Figure 8.141 Behavioral module to demonstrate the use of the assign . . . deassign procedural continuous assignment.
module assign_deassign2_tb;

reg rst_n, clk, d;
wire q;

initial
$monitor("rst_n=%b, clk= %b, d= %b, q=%b", rst_n, clk, d, q);

initial //define clock
begin
  clk = 1'b0;
  forever
    #10 clk = ~clk;
  end

initial //apply inputs
begin
  #0 rst_n = 1'b1;
  #5 d = 1'b0;
  #10 d = 1'b1;
  #10 d = 1'b1;
  #10 d = 1'b0;
  #10 d = 1'b0;
  #10 d = 1'b1;
  #10 d = 1'b0;
  #10 d = 1'b0;
  #10 d = 1'b0;
  #10 d = 1'b0;
  #10 d = 1'b1;
  #10 d = 1'b1;
  #10 d = 1'b1;
  #10 d = 1'b1;
  #10 d = 1'b0;
  #10 $stop;
end

//instantiate the module into the test bench
assign_deassign2 inst1 (.rst_n(rst_n),
  .clk(clk),
  .d(d),
  .q(q))
endmodule

Figure 8.142 Test bench to illustrate the use of the assign . . . deassign statements.
rst_n = 1, clk = 0, d = x, q = x  |  rst_n = 0, clk = 0, d = 1, q = 0
rst_n = 1, clk = 0, d = 0, q = x  |  rst_n = 0, clk = 1, d = 1, q = 0
rst_n = 1, clk = 1, d = 0, q = 0  |  rst_n = 0, clk = 1, d = 0, q = 0
rst_n = 1, clk = 1, d = 1, q = 0  |  rst_n = 0, clk = 0, d = 0, q = 0
rst_n = 1, clk = 0, d = 1, q = 0  |  rst_n = 1, clk = 0, d = 0, q = 0
rst_n = 1, clk = 1, d = 0, q = 0  |  rst_n = 1, clk = 0, d = 0, q = 0

Figure 8.143  Outputs for the module of Figure 8.141.

Figure 8.144  Waveforms for the module of Figure 8.141.
//d flip-flop behavioral
module d_ff_bh (rst_n, clk, d, q, q_n);

input rst_n, clk, d;
output q, q_n;

wire rst_n, clk, d;
reg q;

assign q_n = ~q;

always @(rst_n or posedge clk)
begin
  if (rst_n == 0)
    q <= 1'b0;
  else q <= d;
end
endmodule

Figure 8.145 Behavioral module for a $D$ flip-flop to illustrate the force and release statements.

//d_ff_bh test bench
module d_ff_bh_tb;

reg rst_n, clk, d;
wire q;

//display variables
initial
$monitor ("rst_n = %b, clk = %b, d = %b, q = %b",
          rst_n, clk, d, q);

//define clock
initial begin
  clk = 1'b0;
  forever
    #10clk = ~clk;
end
//continued on next page

Figure 8.146 Test bench for a $D$ flip-flop illustrating the force and release statements.
// apply input vectors

initial
begin
  #0  d = 1'b0;  rst_n = 1'b0;
  #5  d = 1'b1;  rst_n = 1'b1;
  #10 d = 1'b1;  rst_n = 1'b1;
  #10 d = 1'b1;  rst_n = 1'b1;
  #10 d = 1'b0;  rst_n = 1'b1;
  #10 d = 1'b0;  rst_n = 1'b1;
  #50 force q = 1'b1;
  #50 release q;
  #10 d = 1'b0;  rst_n = 1'b1;
  #10 d = 1'b1;  rst_n = 1'b1;
  #10 d = 1'b0;  rst_n = 1'b1;
  #10 $stop;
end

// instantiate the module into the test bench

d_ff_bh inst1 (  
  .rst_n(rst_n),  
  .clk(clk),  
  .d(d),  
  .q(q)  
);
endmodule

Figure 8.146  (Continued)

Page 486

Figure 8.147  Waveforms for the D flip-flop illustrating the force and release statements.