// and3 dataflow
module and3_df (x1, x2, x3, z1);

input x1, x2, x3;
output z1;

wire x1, x2, x3; // define signals as wire for dataflow
wire z1;

// continuous assignment for dataflow
assign z1 = x1 & x2 & x3;
endmodule

Figure 7.1 Module for a 3-input AND gate using continuous assignment.

// test bench for the 3-input AND gate
module and3_df_tb;

reg x1, x2, x3; // inputs are reg for test bench
wire z1; // outputs are wire for test bench

// 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", 
                   {x1, x2, x3}, z1);
    end
end

// instantiate the module into the test bench
and3_df inst1 (.
    .x1(x1),
    .x2(x2),
    .x3(x3),
    .z1(z1)
);
endmodule

Figure 7.2 Test bench for the 3-input AND gate using continuous assignment.
\[ \begin{align*}
&x_1 \ x_2 \ x_3 = 000, \ z_1 = 0 \\
&x_1 \ x_2 \ x_3 = 001, \ z_1 = 0 \\
&x_1 \ x_2 \ x_3 = 010, \ z_1 = 0 \\
&x_1 \ x_2 \ x_3 = 011, \ z_1 = 0 \\
&x_1 \ x_2 \ x_3 = 100, \ z_1 = 0 \\
&x_1 \ x_2 \ x_3 = 101, \ z_1 = 0 \\
&x_1 \ x_2 \ x_3 = 110, \ z_1 = 0 \\
&x_1 \ x_2 \ x_3 = 111, \ z_1 = 1
\end{align*} \]

**Figure 7.3** Outputs for the 3-input AND gate using continuous assignment.

**Figure 7.4** Waveforms for the 3-input AND gate using continuous assignment.
// a sum-of-products expression using continuous assignment statements
module log_eqn_sop11 (x1, x2, x3, x4, z1);

input x1, x2, x3, x4;
output z1;
wire x1, x2, x3, x4;
wire z1;
assign z1 = (x1 & ~x2) | (~x3 & x4) | (x2 & x3);
endmodule

Figure 7.7 Module to implement the logic diagram of Figure 7.6.

// a sum-of-products expression using continuous assignment statements
module log_eqn_sop11a (x1, x2, x3, x4, z1);

input x1, x2, x3, x4;
output z1;
wire x1, x2, x3, x4;
wire net1, net2, net3; // define internal nets
wire z1;
assign net1 = x1 & ~x2;
assign net2 = ~x3 & x4;
assign net3 = x2 & x3;
assign z1 = net1 | net2 | net3;
endmodule

Figure 7.8 Alternative method to implement the logic diagram of Figure 7.6.
//test bench for the sum-of-products expression
module log_eqn_sop11a_tb;

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

//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", 
                   {x1, x2, x3, x4}, z1);
    end
end

//instantiate the module into the test bench
log_eqn_sop11a inst1 ( 
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .x4(x4),
  .z1(z1)
);
endmodule

Figure 7.9  Test bench for the modules of Figure 7.7 and Figure 7.8.

<table>
<thead>
<tr>
<th>x1 x2 x3 x4 = 0000, z1 = 0</th>
<th>x1 x2 x3 x4 = 1000, z1 = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1 x2 x3 x4 = 0001, z1 = 1</td>
<td>x1 x2 x3 x4 = 1001, z1 = 1</td>
</tr>
<tr>
<td>x1 x2 x3 x4 = 0010, z1 = 0</td>
<td>x1 x2 x3 x4 = 1010, z1 = 1</td>
</tr>
<tr>
<td>x1 x2 x3 x4 = 0011, z1 = 0</td>
<td>x1 x2 x3 x4 = 1011, z1 = 1</td>
</tr>
<tr>
<td>x1 x2 x3 x4 = 0100, z1 = 0</td>
<td>x1 x2 x3 x4 = 1100, z1 = 0</td>
</tr>
<tr>
<td>x1 x2 x3 x4 = 0101, z1 = 1</td>
<td>x1 x2 x3 x4 = 1101, z1 = 1</td>
</tr>
<tr>
<td>x1 x2 x3 x4 = 0110, z1 = 1</td>
<td>x1 x2 x3 x4 = 1110, z1 = 1</td>
</tr>
<tr>
<td>x1 x2 x3 x4 = 0111, z1 = 1</td>
<td>x1 x2 x3 x4 = 1111, z1 = 1</td>
</tr>
</tbody>
</table>

Figure 7.10  Outputs for the modules of Figure 7.7 and Figure 7.8.
Figure 7.11  Waveforms for the modules of Figure 7.7 and Figure 7.8.

//module to illustrate the use of reduction operators
module reduction1 (a, red_and, red_nand, red_or, red_nor,
                     red_xor, red_xnor);
input [7:0] a;
output red_and, red_nand, red_or, red_nor, red_xor, red_xnor;
wire [7:0] a;
wire red_and, red_nand, red_or, red_nor, red_xor, red_xnor;
assign red_and = &a, //reduction AND
       red_nand = ~&a, //reduction NAND
       red_or = |a, //reduction OR
       red_nor = ~|a, //reduction NOR
       red_xor = ^a, //reduction exclusive-OR
       red_xnor = ^~a; //reduction exclusive-NOR
endmodule

Figure 7.12  Module using continuous assignment to demonstrate the reduction operators.
// test bench for reduction module
module reduction1_tb;

reg [7:0] a;
wire red_and, red_nand, red_or, red_nor, red_xor, red_xnor;

initial
$monitor ("a=%b, red_and=%b, red_nand=%b, red_or=%b,
red_nor=%b, red_xor=%b, red_xnor=%b",
a, red_and, red_nand, red_or, red_nor, red_xor, red_xnor);

// apply input vectors
initial begin
  #0  a = 8'b1100_0011;
  #10 a = 8'b1001_0111;
  #10 a = 8'b0000_0000;
  #10 a = 8'b0100_1111;
  #10 a = 8'b1111_1111;
  #10 $stop;
end

// instantiate the module into the test bench
reduction1 inst1 (
  .a(a),
  .red_and(red_and),
  .red_nand(red_nand),
  .red_or(red_or),
  .red_nor(red_nor),
  .red_xor(red_xor),
  .red_xnor(red_xnor)
);

endmodule

Figure 7.13  Test bench for the continuous assignment module of Figure 7.12.
a=11000011,
red_and=0, red_nand=1, red_or=1,
red_nor=0, red_xor=0, red_xnor=1

a=10010111,
red_and=0, red_nand=1, red_or=1,
red_nor=0, red_xor=1, red_xnor=0

a=00000000,
red_and=0, red_nand=1, red_or=0,
red_nor=1, red_xor=0, red_xnor=1

a=01001111,
red_and=0, red_nand=1, red_or=1,
red_nor=0, red_xor=1, red_xnor=0

a=11111111,
red_and=1, red_nand=0, red_or=1,
red_nor=0, red_xor=0, red_xnor=1

Figure 7.14  Outputs for the test bench of Figure 7.13.

Figure 7.15   Waveforms for the test bench of Figure 7.13.
//an octal-to-binary encoder
module encoder_8_to_3 (oct, bin);

input [0:7] oct;
output [2:0] bin;

wire [0:7] oct;
wire [2:0] bin;

endmodule

Figure 7.19 Module for the octal-to-binary encoder using continuous assignment statements.

//test bench for the 8-to-3 encoder
module encoder_8_to_3_tb;

reg [0:7] oct;
wire [2:0] bin;

//display variables
initial
$monitor("octal = %b, binary = %b", oct [0:7], bin [2:0]);

//apply input vectors
initial
begin
#0 oct [0:7] = 8'b1000_0000;
#10 oct [0:7] = 8'b0100_0000;
#10 oct [0:7] = 8'b0010_0000;
#10 oct [0:7] = 8'b0001_0000;
#10 oct [0:7] = 8'b0000_1000;
#10 oct [0:7] = 8'b0000_0100;
#10 oct [0:7] = 8'b0000_0010;
#10 oct [0:7] = 8'b0000_0001;
#10 $stop;
end

//instantiate the module into the test bench
encoder_8_to_3 inst1 (.
.oct(oct),
.bin(bin)
);
endmodule

Figure 7.20 Test bench for the octal-to-binary encoder.
octal = 10000000, binary = 000
octal = 01000000, binary = 001
octal = 00100000, binary = 010
octal = 00010000, binary = 011
octal = 00001000, binary = 100
octal = 00000100, binary = 101
octal = 00000010, binary = 110
octal = 00000001, binary = 111

**Figure 7.21** Outputs for the test bench of Figure 7.20 for the octal-to-binary encoder.

**Figure 7.22** Waveforms for the octal-to-binary encoder.
module mux4_df (s, d, enbl, z1); // dataflow 4:1 multiplexer
input [1:0] s;
input [3:0] d;
input enbl;
output z1;
wire [1:0] s;
wire [3:0] d;
wire enbl;
wire z1;
assign z1 = (~s[1] & ~s[0] & d[0] & enbl) |
            (~s[1] &  s[0] & d[1] & enbl) |
            ( s[1] &  s[0] & d[3] & enbl);
endmodule

Figure 7.24  Design module for a 4:1 multiplexer using a continuous assignment statement.

Page 313

// dataflow mux4_df test bench
module mux4_df_tb;
reg [1:0] s;
reg [3:0] d;
reg enbl;
wire z1;

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

// apply input vectors
initial
begin
    #0 s = 2'b00;
    d = 4'b1111;
    enbl = 1'b0;
end

// continued on next page

Figure 7.25  Test bench for the 4:1 multiplexer of Figure 7.24.
#10  s = 2'b00;
d = 4'b0000;
enbl = 1'b1;

#10  s = 2'b00;
d = 4'b0001;
enbl = 1'b1;

#10  s = 2'b01;
d = 4'b0001;
enbl = 1'b1;

#10  s = 2'b01;
d = 4'b0011;
enbl = 1'b1;

#10  s = 2'b10;
d = 4'b1001;
enbl = 1'b1;

#10  s = 2'b10;
d = 4'b0101;
enbl = 1'b1;

#10  s = 2'b11;
d = 4'b0111;
enbl = 1'b1;

#10  s = 2'b11;
d = 4'b1001;
enbl = 1'b1;

#10  $stop;
end

// instantiate the module into the test bench
mux4_df inst1 (  
  .s(s),
  .d(d),
  .enbl(enbl),
  .z1(z1)
);
endmodule

Figure 7.25  (Continued)
Page 314

select: s = 00, data: d = 1111, out: z1 = 0
select: s = 00, data: d = 0000, out: z1 = 0
select: s = 00, data: d = 0001, out: z1 = 1
select: s = 01, data: d = 0001, out: z1 = 0
select: s = 01, data: d = 0011, out: z1 = 1
select: s = 10, data: d = 1001, out: z1 = 0
select: s = 10, data: d = 0101, out: z1 = 1
select: s = 11, data: d = 0111, out: z1 = 0
select: s = 11, data: d = 1001, out: z1 = 1

Figure 7.26 Outputs of the 4:1 multiplexer of Figure 7.24.

Page 315

Figure 7.27 Waveforms for the 4:1 multiplexer of Figure 7.24.
module mux4to1_cond (out, in0, in1, in2, in3, s0, s1);
    input s0, s1;
    input in0, in1, in2, in3;
    output out;

    assign out = s1 ? (s0 ? in3 : in2) : (s0 ? in1 : in0);
endmodule

Figure 7.28 Module for a 4:1 multiplexer using a conditional operator and a continuous assignment statement.

module mux4to1_cond_tb;
    reg in0, in1, in2, in3, s0, s1; //inputs are reg
    wire out; //outputs are wire

    //display signals
    initial
        $monitor("s1s0 = %b, in0in1in2in3 = %b, out = %b", 
            {s1, s0}, {in0, in1, in2, in3}, out);

    //apply stimulus
    initial begin
        #0 s1 = 1'b0;
        s0 = 1'b0;
        in0 = 1'b0; //out = 0
        in1 = 1'b1;
        in2 = 1'b1;
        in3 = 1'b1; //continued on next page
end

Figure 7.29 Test bench for the conditional operator multiplexer of Figure 7.28.
#10 s1 = 1'b0;
s0 = 1'b1;
in0 = 1'b0;
in1 = 1'b1;    //out = 1
in2 = 1'b1;
in3 = 1'b1;

#10 s1 = 1'b1;
s0 = 1'b0;
in0 = 1'b0;
in1 = 1'b0;
in2 = 1'b0;    //out = 0
in3 = 1'b1;

#10 s1 = 1'b1;
s0 = 1'b1;
in0 = 1'b0;
in1 = 1'b1;
in2 = 1'b0;
in3 = 1'b1;    //out = 0

#10 $stop;
end

//instantiate the module into the test bench
mux4to1_cond inst1 (  
.s0(s0),  
.s1(s1),  
.in0(in0),  
.in1(in1),  
.in2(in2),  
.in3(in3),  
.out(out)  
);
endmodule

Figure 7.29  (Continued)

s1s0 = 00, in0inlin2in3 = 0111, out = 0
s1s0 = 01, in0inlin2in3 = 0111, out = 1
s1s0 = 10, in0inlin2in3 = 0101, out = 0
s1s0 = 11, in0inlin2in3 = 0101, out = 1

Figure 7.30  Outputs for the 4:1 conditional operator multiplexer of Figure 7.28.
//dataflow for a 4-bit adder
module adder4_df (a, b, cin, sum, cout);

//list inputs and outputs
input [3:0] a, b;
input cin;
output [3:0] sum;
output cout;

//define signals as wire for dataflow (or default to wire)
wire [3:0] a, b;
wire cin, cout;
wire [3:0] sum;

//continuous assignment for dataflow
//implement the 4-bit adder as a logic equation
//concatenating cout and sum
assign {cout, sum} = a + b + cin;
endmodule

Figure 7.31  Dataflow module for a 4-bit adder using continuous assignment.

//dataflow 4-bit adder test bench
module adder4_df_tb;

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

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

//apply stimulus
initial begin
  #0 a = 4'b0000;
  b = 4'b0000;
  cin = 1'b0; //sum = 0000

//continued on next page

Figure 7.32  Test bench for Figure 7.31 for a 4-bit adder using continuous assignment.
#10  a = 4'b0001;
b = 4'b0010;
cin = 1'b0; //sum = 0011

#10  a = 4'b0010;
b = 4'b0110;
cin = 1'b0; //sum = 1000

#10  a = 4'b0111;
b = 4'b0111;
cin = 1'b0; //sum = 1110

#10  a = 4'b1001;
b = 4'b0110;
cin = 1'b0; //sum = 1111

#10  a = 4'b1000;
b = 4'b0011;
cin = 1'b0; //sum = 1111

#10  a = 4'b1111;
b = 4'b0000;
cin = 1'b0; //sum = 1111

#10  a = 4'b1111;
b = 4'b0001;
cin = 1'b1; //sum = 1100

#10  a = 4'b1100;
b = 4'b0001;
cin = 1'b0; //sum = 1110

#10  a = 4'b1100;
b = 4'b0001;
cin = 1'b1; //sum = 1100

#10  a = 4'b1011;
b = 4'b1011;
cin = 1'b1; //sum = 1_0111

#10  a = 4'b1111;
b = 4'b1111;
cin = 1'b1; //sum = 1_1111

#10  $stop;

end

//continued on next page
// instantiate the module into the test bench
adder4_df inst1 (
    .a(a),
    .b(b),
    .cin(cin),
    .sum(sum),
    .cout(cout)
);
endmodule

Figure 7.32  (Continued)

Page 321

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>cout</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td>0010</td>
<td>0</td>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>0010</td>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
<td>0</td>
<td>0</td>
<td>1110</td>
</tr>
<tr>
<td>1001</td>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>1111</td>
</tr>
<tr>
<td>1100</td>
<td>0011</td>
<td>0</td>
<td>0</td>
<td>1111</td>
</tr>
<tr>
<td>1111</td>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>1111</td>
</tr>
<tr>
<td>1100</td>
<td>0001</td>
<td>1</td>
<td>0</td>
<td>1110</td>
</tr>
<tr>
<td>0011</td>
<td>1000</td>
<td>1</td>
<td>0</td>
<td>1100</td>
</tr>
<tr>
<td>1011</td>
<td>1011</td>
<td>1</td>
<td>1</td>
<td>0111</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>1</td>
<td>1</td>
<td>1111</td>
</tr>
</tbody>
</table>

Figure 7.33  Outputs for the 4-bit adder of Figure 7.31.
Figure 7.34  Waveforms for the 4-bit adder of Figure 7.31.
Figure 7.36  Design module for a 4-bit carry lookahead adder.
//test bench for the dataflow 4-bit carry lookahead adder
module adder4_cla_tb;

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

//display signals
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; //cout = 0, sum = 0000

    #10 a = 4'b0001;
b = 4'b0010;
cin = 1'b0; //cout = 0, sum = 0011

    #10 a = 4'b0010;
b = 4'b0110;
cin = 1'b0; //cout = 0, sum = 1000

    #10 a = 4'b0111;
b = 4'b0111;
cin = 1'b0; //cout = 0, sum = 1110

    #10 a = 4'b1001;
b = 4'b0110;
cin = 1'b0; //cout = 0, sum = 1111

    #10 a = 4'b1100;
b = 4'b1100;
cin = 1'b0; //cout = 1, sum = 1000

    //continued on next page

Figure 7.37  Test bench for the 4-bit carry lookahead adder of Figure 7.36.
#10 a = 4'b1111;
b = 4'b1110;
cin = 1'b0; //cout = 1, sum = 1101

#10 a = 4'b1110;
b = 4'b1110;
cin = 1'b1; //cout = 1, sum = 1101

#10 a = 4'b1111;
b = 4'b1111;
cin = 1'b1; //cout = 1, sum = 1111

#10 $stop;
end

// instantiate the module into the test bench
adder4_cla inst1 (  
  .a(a),  
  .b(b),  
  .cin(cin),  
  .sum(sum),  
  .cout(cout)  
);
endmodule

Figure 7.37  (Continued)

Page 327

a = 0000, b = 0000, cin = 0, cout = 0, sum = 0000
a = 0001, b = 0010, cin = 0, cout = 0, sum = 0011
a = 0010, b = 0110, cin = 0, cout = 0, sum = 1000
a = 0111, b = 0111, cin = 0, cout = 0, sum = 1110
a = 1001, b = 0110, cin = 0, cout = 0, sum = 1111
a = 1100, b = 1100, cin = 0, cout = 1, sum = 1000
a = 1111, b = 1110, cin = 0, cout = 1, sum = 1101
a = 1110, b = 1110, cin = 1, cout = 1, sum = 1111
a = 1111, b = 1111, cin = 1, cout = 1, sum = 1111

Figure 7.38  Outputs for the 4-bit carry lookahead adder of Figure 7.36.
Figure 7.39  Waveforms for the 4-bit carry lookahead adder of Figure 7.36.
//asynchronous sequential machine
module asm (x1, x2, z1);

input x1, x2;
output z1;
wire net1, net2, net3; //define internal nets

assign net1 = x1 & ~x2,
    net2 = x1 & net3,
    net3 = net1 | net2;

assign z1 = net3 & x2;
endmodule

Figure 7.48  Design module for the asynchronous sequential machine of Example 7.8.

// test bench for the asynchronous sequential machine
module asm_tb;
reg x1, x2;
wire z1;

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

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

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

Figure 7.49  Test bench for the asynchronous sequential machine module of Figure 7.48.
Figure 7.50  Waveforms for the asynchronous sequential machine module of Figure 7.48.

Figure 7.58  Design module for the asynchronous machine of Example 7.9.
//test bench for the asynchronous sequential machine
module asm6_tb;

reg x1, x2, rst_n;
wire yle, z1;

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

//apply stimulus
initial
begin
  #0  rst_n = 1'b0;
  x1 = 1'b0;
  x2 = 1'b0;
  #5  rst_n = 1'b1;
  #10 x1 = 1'b1; x2 = 1'b0;
  #10 x1 = 1'b0; x2 = 1'b0;
  #10 x1 = 1'b0; x2 = 1'b1;
  #10 x1 = 1'b0; x2=1'b0;
  #10 x1 = 1'b0; x2=1'b1;
  #10 x1 = 1'b0; x2=1'b0;
  #10 x1 = 1'b1; x2=1'b0;
  #10 x1 = 1'b0; x2=1'b0;
  #10 x1 = 1'b1; x2=1'b0;
  #10 x1 = 1'b0; x2=1'b0;
  #10 $stop;
end

//instantiate the module into the test bench
asm6 inst1 (  
  .x1(x1),  
  .x2(x2),  
  .rst_n(rst_n),  
  .yle(yle),  
  .z1(z1)
);
endmodule

Figure 7.59  Test bench for the asynchronous machine of Example 7.9.
### Chapter 7  Dataflow Modeling

**Figure 7.60**  Outputs for the asynchronous machine of Example 7.9.

<table>
<thead>
<tr>
<th>State Transition</th>
<th>Output z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=0, x2=0, state=0, z1=0</td>
<td></td>
</tr>
<tr>
<td>x1=1, x2=0, state=1, z1=1</td>
<td></td>
</tr>
<tr>
<td>x1=0, x2=0, state=1, z1=1</td>
<td></td>
</tr>
<tr>
<td>x1=0, x2=1, state=0, z1=0</td>
<td></td>
</tr>
<tr>
<td>x1=0, x2=0, state=0, z1=0</td>
<td></td>
</tr>
<tr>
<td>x1=1, x2=0, state=1, z1=1</td>
<td></td>
</tr>
<tr>
<td>x1=0, x2=0, state=1, z1=1</td>
<td></td>
</tr>
<tr>
<td>x1=1, x2=1, state=1, z1=0</td>
<td></td>
</tr>
<tr>
<td>x1=1, x2=0, state=1, z1=1</td>
<td></td>
</tr>
<tr>
<td>x1=0, x2=0, state=1, z1=1</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.61**  Waveforms for the asynchronous machine of Example 7.9.
//behavioral d flip-flop
module d_ff (d, clk, q, q_n, set_n, rst_n);

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

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

always @(posedge clk or negedge rst_n or negedge set_n)
begin
  if (rst_n == 0)
  begin
    q = 1'b0;
    q_n = 1'b1;
  end
  else if (set_n == 0)
  begin
    q = 1'b1;
    q_n = 1'b0;
  end
  else
  begin
    q = d;
    q_n = ~q;
  end
end
endmodule

Figure 7.68  Module for a D flip-flop to be instantiated into the module for the Moore machine of Example 7.10.
//pulse-mode asynchronous sequential machine
module pm_asm3 (x1, x2, set_n, rst_n, y, y_n, z1);

input x1, x2, set_n, rst_n;
output [1:2] y, y_n;
output z1;

//define internal nets
wire net1, net2, net3, net4, net5, net6, net7, net8;

assign net1 = ~(~x1 & ~x2), //design for D flip-flop clock
       net2 = ~(net1);

assign net3 = ~(y[2] & x1), //design for latch Ly1
       net4 = ~(~y[2] & x2),
       net5 = (~net3 | ~net6),
       net6 = ~(net5 & net4 & rst_n);

assign net7 = (x1 | ~net8), //design for latch Ly2
       net8 = ~(net7 & ~x2 & rst_n);

//instantiate D flip-flop for y1
d_ff inst1 (  .d(net5),
               .clk(net2),
               .q(y[1]),
               .q_n(y_n[1]),
               .set_n(set_n),
               .rst_n(rst_n)
               );

//instantiate D flip-flop for y2
d_ff inst2 (  .d(net7),
               .clk(net2),
               .q(y[2]),
               .q_n(y_n[2]),
               .set_n(set_n),
               .rst_n(rst_n)
               );

//design for z1
assign z1 = y[1];

endmodule

Figure 7.69  Design module for the Moore machine of Example 7.10.
module pm_asm3_tb;

reg x1, x2;
reg set_n, rst_n;
wire [1:2] y, y_n;
wire z1;

initial //display inputs and outputs
$monitor ("rst_n = %b, x1x2 = %b, state = %b, z1 = %b",
    rst_n, {x1, x2}, y, z1);

initial //define input sequence
begin
    #0 set_n = 1'b1;
    rst_n = 1'b0;//reset to state_a(00); no output
    x1    = 1'b0;
    x2    = 1'b0;
    #5 rst_n = 1'b1;
    #10 x1 = 1'b1; //go to state_b(01) on posedge of 1st x1
    #10 x1 = 1'b0; //no output
    #10 x1 = 1'b1; //go to state_c(11) on posedge of 2nd x1
    #10 x1 = 1'b0; //assert output z1 on negedge of 2nd x1
    #10 x1 = 1'b1; //remain in state_c(11)
    #10 x1 = 1'b0; //output z1 remains asserted
    #20 x2 = 1'b1; //go to state_d(10) on posedge of 1st x2
    #10 x2 = 1'b0; //output z1 remains asserted
    #10 x1 = 1'b1; //go to state_c(11)
    #10 x1 = 1'b0; //output z1 remains asserted
    #10 x2 = 1'b1; //go to state_d(10) on posedge of 1st x2
    #10 x2 = 1'b0; //output z1 remains asserted
    #10 x2 = 1'b1; //go to state_a(00) on posedge of 2nd x2
    #10 x2 = 1'b0; //deassert output z1 on negedge of 2nd x2
    #30 $stop;
end

//continued on next page

Figure 7.70 Test bench for the Moore machine of Example 7.10.
// instantiate the module into test bench
pm_asm3 inst1 (.x1(x1),
   .x2(x2),
   .set_n(set_n),
   .rst_n(rst_n),
   .y(y),
   .y_n(y_n),
   .z1(z1)
);
endmodule

Figure 7.70  (Continued)

rst_n = 0, x1x2 = 00, state = 00, z1 = 0
rst_n = 1, x1x2 = 00, state = 00, z1 = 0
rst_n = 1, x1x2 = 10, state = 00, z1 = 0
rst_n = 1, x1x2 = 00, state = 01, z1 = 0
rst_n = 1, x1x2 = 10, state = 01, z1 = 0
rst_n = 1, x1x2 = 00, state = 11, z1 = 1
rst_n = 1, x1x2 = 10, state = 11, z1 = 1
rst_n = 1, x1x2 = 00, state = 11, z1 = 1
rst_n = 1, x1x2 = 01, state = 11, z1 = 1
rst_n = 1, x1x2 = 00, state = 10, z1 = 1
rst_n = 1, x1x2 = 10, state = 10, z1 = 1
rst_n = 1, x1x2 = 00, state = 11, z1 = 1
rst_n = 1, x1x2 = 01, state = 11, z1 = 1
rst_n = 1, x1x2 = 00, state = 10, z1 = 1
rst_n = 1, x1x2 = 01, state = 10, z1 = 1
rst_n = 1, x1x2 = 00, state = 00, z1 = 0

Figure 7.71  Outputs for the Moore machine of Example 7.10.
Figure 7.72  Waveforms for the Moore machine of Example 7.10.

Figure 7.73  Module demonstrating an implicit continuous assignment statement.
// test bench for implicit continuous assignment
module impl_cont_assign_tb;

reg [7:0] a, b;
wire [7:0] rslt;

initial // apply stimulus and display variables
begin
  #0 a = 8'b1010_0011; b = 8'b1101_1111;
  #10 $display("a=%b, b=%b, rslt=%b", a, b, rslt);

  #10 a = 8'b1111_0000; b = 8'b0000_1111;
  #10 $display("a=%b, b=%b, rslt=%b", a, b, rslt);

  #10 a = 8'b0101_0101; b = 8'b0101_0101;
  #10 $display("a=%b, b=%b, rslt=%b", a, b, rslt);

  #10 a = 8'b0000_0000; b = 8'b0100_0000;
  #10 $display("a=%b, b=%b, rslt=%b", a, b, rslt);

  #10 a = 8'b0000_0000; b = 8'b0010_0000;
  #10 $display("a=%b, b=%b, rslt=%b", a, b, rslt);

  #10 a = 8'b0000_0000; b = 8'b0001_0000;
  #10 $display("a=%b, b=%b, rslt=%b", a, b, rslt);

  #10 a = 8'b0000_0000; b = 8'b0000_1000;
  #10 $display("a=%b, b=%b, rslt=%b", a, b, rslt);

  #10 $stop;
end

// instantiate the module into the test bench
impl_cont_assign inst1 (.a(a), .b(b), .rslt(rslt));
endmodule

Figure 7.74  Test bench for the implicit continuous assignment module of Figure 7.73.
a = 10100011, b = 11011111, rslt = 01111100
a = 11110000, b = 00001111, rslt = 11111111
a = 01010101, b = 01010101, rslt = 00000000
a = 00000000, b = 01000000, rslt = 01000000
a = 00000000, b = 00100000, rslt = 00100000
a = 00000000, b = 00010000, rslt = 00010000
a = 00000000, b = 00001000, rslt = 00001000

**Figure 7.75** Outputs for the test bench of Figure 7.74.

**Figure 7.76** Waveforms for the test bench of Figure 7.74.
//2-to-4 decoder using NAND gates
module decoder_2to4 (a, b, enbl_n, z);

input a, b, enbl_n;
output [3:0] z;

//define internal nets
wire net1, net2, net3;
assign net1 = ~a;
assign net2 = ~b;
assign net3 = ~enbl_n;

//design for decoder with delay
assign #3 z[0] = ~(net1 & net2 & net3);
assign #3 z[1] = ~(net1 & b & net3);
assign #3 z[2] = ~(a & net2 & net3);
assign #3 z[3] = ~(a & b & net3);
endmodule

Figure 7.78 Module for the 2:4 decoder of Example 7.12.

//test bench for 2-to-4 decoder
module decoder_2to4_tb;

//inputs are reg for test benches
//outputs are wire for test benches
reg a, b, enbl_n;
wire [3:0] z;

//display variables
initial
$monitor ("ab = %b, z = %b", {a, b}, z);

//continued on next page

Figure 7.79 Test bench for the 2:4 decoder of Example 7.12.
//apply input vectors

initial

begin
  #0  enbl_n = 1'b0; //enable is asserted
    a = 1'b0; b = 1'b0;
  #10 a = 1'b0; b = 1'b0;
  #10 a = 1'b0; b = 1'b1;
  #10 a = 1'b1; b = 1'b0;
  #10 a = 1'b1; b = 1'b1;
  #10 a = 1'b0; b = 1'b0;
  #10 enbl_n = 1'b1; //enable is deasserted
  #10 a = 1'b0; b = 1'b1;
  #10 a = 1'b1; b = 1'b0;
  #10 $stop;
end

//instantiate the module into the test bench
decoder_2to4 inst1 (
  .a(a),
  .b(b),
  .enbl_n(enbl_n),
  .z(z)
);
endmodule

Figure 7.79  (Continued)

Figure 7.80  Waveforms for the 2:4 decoder. The output vector z is active low.