Chapter 5

Gate-Level Modeling

Verilog HDL: Digital Design and Modeling

Chapter 5

Gate-Level Modeling
//gate-level modeling for and/or gates
module and3_or3 (x1, x2, x3, and3_out, or3_out);

input x1, x2, x3;
output and3_out, or3_out;
and (and3_out, x1, x2, x3);
or (or3_out, x1, x2, x3);
endmodule

Figure 5.1 Verilog code for a 3-input AND gate and a 3-input OR gate using built-in primitives.

//test bench for and3_or3 module
module and3_or3_tb;

reg x1, x2, x3;
wire and3_out, or3_out;

//monitor variables
initial
$monitor ("x1x2x3 = %b, and3_out = %b, or3_out = %b",
{x1, x2, x3}, and3_out, or3_out);

initial
begin
#0 x1=1'b0; x2=1'b0; x3=1'b0;
#10 x1=1'b0; x2=1'b0; x3=1'b1;
#10 x1=1'b0; x2=1'b1; x3=1'b0;
#10 x1=1'b0; x2=1'b1; x3=1'b1;
#10 x1=1'b1; x2=1'b0; x3=1'b0;
#10 x1=1'b0; x2=1'b1; x3=1'b1;
#10 x1=1'b1; x2=1'b0; x3=1'b0;
#10 x1=1'b1; x2=1'b1; x3=1'b1;
#10 $stop;
end

//continued on next page

Figure 5.2 Test bench for Figure 5.1 for a 3-input AND gate and a 3-input OR gate.
// instantiate the module into the test bench
and3_or3 inst1 (  
    .x1(x1),  
    .x2(x2),  
    .x3(x3),  
    .and3_out(and3_out),  
    .or3_out(or3_out)  
);  
endmodule

Figure 5.2 (Continued)

Page 162

x1x2x3 = 000, and3_out = 0, or3_out = 0
x1x2x3 = 001, and3_out = 0, or3_out = 1
x1x2x3 = 010, and3_out = 0, or3_out = 1
x1x2x3 = 011, and3_out = 0, or3_out = 1
x1x2x3 = 100, and3_out = 0, or3_out = 1
x1x2x3 = 101, and3_out = 0, or3_out = 1
x1x2x3 = 110, and3_out = 0, or3_out = 1
x1x2x3 = 111, and3_out = 1, or3_out = 1

Figure 5.3 Outputs for the test bench of Figure 5.2.

Figure 5.4 Waveforms for the and3_or3 module of Figure 5.1.
module xor_xnor (x1, x2, x3, x4, xor_out, xnor_out);
input x1, x2, x3, x4;
output xor_out, xnor_out;
xor (xor_out, x1, x2, x3, x4);
xnor (xnor_out, x1, x2, x3, x4);
endmodule

//test bench for xor/xnor module
module xor_xnor_tb;
reg x1, x2, x3, x4;
wire xor_out, xnor_out;

//monitor variables
initial
$monitor ("x1x2x3x4 = %b, xor_out = %b, xnor_out = %b",
{x1, x2, x3, x4}, xor_out, xnor_out);
initial
begin
#0 x1=1'b0; x2=1'b0; x3=1'b0; x4=1'b0;
#10 x1=1'b0; x2=1'b0; x3=1'b0; x4=1'b1;
#10 x1=1'b0; x2=1'b0; x3=1'b1; x4=1'b1;
#10 x1=1'b0; x2=1'b1; x3=1'b0; x4=1'b1;
#10 x1=1'b0; x2=1'b1; x3=1'b1; x4=1'b0;
#10 x1=1'b0; x2=1'b1; x3=1'b1; x4=1'b1;
#10 x1=1'b1; x2=1'b0; x3=1'b0; x4=1'b0;
#10 x1=1'b1; x2=1'b0; x3=1'b0; x4=1'b1;
#10 x1=1'b1; x2=1'b0; x3=1'b1; x4=1'b0;
#10 x1=1'b1; x2=1'b0; x3=1'b1; x4=1'b1;
#10 x1=1'b1; x2=1'b1; x3=1'b0; x4=1'b0;
#10 x1=1'b1; x2=1'b1; x3=1'b0; x4=1'b1;
#10 x1=1'b1; x2=1'b1; x3=1'b1; x4=1'b0;
#10 x1=1'b1; x2=1'b1; x3=1'b1; x4=1'b1;
#10 $stop;
end
//continued on next page

Figure 5.5 Verilog module illustrating the xor and xnor built-in primitives.

Figure 5.6 Test bench for Figure 5.5.
// instantiate the module into the test bench
xor_xnor inst1 (  
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .x4(x4),
  .xor_out(xor_out),
  .xnor_out(xnor_out)
);
endmodule

Figure 5.6  (Continued)

Page 165

\[
\begin{align*}
x_1x_2x_3x_4 &= 0000, \ xor\_out = 0, \ xnor\_out = 1 \\
x_1x_2x_3x_4 &= 0001, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 0010, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 0011, \ xor\_out = 0, \ xnor\_out = 1 \\
x_1x_2x_3x_4 &= 0100, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 0101, \ xor\_out = 0, \ xnor\_out = 1 \\
x_1x_2x_3x_4 &= 0110, \ xor\_out = 0, \ xnor\_out = 1 \\
x_1x_2x_3x_4 &= 0111, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 1000, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 1001, \ xor\_out = 0, \ xnor\_out = 1 \\
x_1x_2x_3x_4 &= 1010, \ xor\_out = 0, \ xnor\_out = 1 \\
x_1x_2x_3x_4 &= 1011, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 1100, \ xor\_out = 0, \ xnor\_out = 1 \\
x_1x_2x_3x_4 &= 1101, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 1110, \ xor\_out = 1, \ xnor\_out = 0 \\
x_1x_2x_3x_4 &= 1111, \ xor\_out = 0, \ xnor\_out = 1
\end{align*}
\]

Figure 5.7  Outputs for the test bench of Figure 5.6.
Figure 5.8  Waveforms for the xor_xnor module of Figure 5.5.

Page 167

```verilog
//logic diagram using built-in primitives
module log_eqn_sop7 (x1, x2, x3, x4, x5, z1);

input x1, x2, x3, x4, x5;
output z1;

and  inst1 (net1, ~x2, ~x4, ~x5),
      inst2 (net2, ~x1, ~x2, ~x4),
      inst3 (net3, x1, ~x2, ~x5),
      inst4 (net4, x2, x3, ~x5),
      inst5 (net5, x2, x4,x5);

or   inst6 (z1, net1, net2, net3, net4, net5);

endmodule
```

Figure 5.11  Module for the sum-of-products equation of Equation 5.1 that represents the logic diagram of Figure 5.10.
//test bench for log_eqn_sop7
module log_eqn_sop7_tb;

reg x1, x2, x3, x4, x5;
wire z1;

//apply input vectors
initial
begin: apply_stimulus
    for (invect=0; invect<32; invect=invect+1)
        begin
            {x1, x2, x3, x4, x5} = invect [6:0];
            #10 $display("x1x2x3x4x5 = %b, z1 = %b", 
                           {x1, x2, x3, x4, x5}, z1);
        end

end

//instantiate the module into the test bench
log_eqn_sop7 inst1 ( 
    .x1(x1),
    .x2(x2),
    .x3(x3),
    .x4(x4),
    .x5(x5),
    .z1(z1)
);

endmodule

Figure 5.12  Test bench for the module of Figure 5.11.
Figure 5.13  Outputs for the test bench of Figure 5.12 for the module of Figure 5.11.

Figure 5.16  Module for the product-of-sums logic diagram of Figure 5.15.
// test bench for product of sums
module log_eqn_pos3_tb;

reg x1, x2, x3, x4, x5;
wire z1;

// apply input vectors
initial begin
  apply_stimulus
    reg [6:0] invect;
    for (invect=0; invect<32; invect=invect+1)
      begin
        {x1, x2, x3, x4, x5} = invect [6:0];
        #10 $display("x1x2x3x4x5 = %b, z1 = %b", {x1, x2, x3, x4, x5}, z1);
      end
end

// instantiate the module into the test bench
log_eqn_pos3 inst1 (
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .x4(x4),
  .x5(x5),
  .z1(z1)
);
endmodule

Figure 5.17  Test bench for the product-of-sums module of Figure 5.16.
\[ \begin{array}{c|c}
\text{x1x2x3x4x5} & \text{z1} \\
00000 & 1 \\
00001 & 1 \\
00010 & 0 \\
00011 & 0 \\
00100 & 1 \\
00101 & 1 \\
00110 & 0 \\
00111 & 0 \\
01000 & 0 \\
01001 & 0 \\
01010 & 0 \\
01011 & 1 \\
01100 & 1 \\
01101 & 0 \\
01110 & 0 \\
01111 & 1 \\
10000 & 1 \\
10001 & 0 \\
10010 & 1 \\
10011 & 0 \\
10100 & 1 \\
10101 & 0 \\
10110 & 1 \\
10111 & 0 \\
11000 & 0 \\
11001 & 0 \\
11010 & 0 \\
11011 & 1 \\
11100 & 1 \\
11101 & 0 \\
11110 & 0 \\
11111 & 1 \\
\end{array} \]

**Figure 5.18** Outputs for the test bench of Figure 5.17 for the product-of-sums module of Figure 5.16.

//5-input majority circuit
module majority (x1, x2, x3, x4, x5, z1);
input x1, x2, x3, x4, x5;
output z1;
and inst1 (net1, x3, x4, x5),
inst2 (net2, x2, x3, x5),
inst3 (net3, x1, x3, x5),
inst4 (net4, x2, x4, x5),
inst5 (net5, x1, x4, x5),
inst6 (net6, x1, x2, x5),
inst7 (net7, x1, x2, x4),
inst8 (net8, x2, x3, x4),
or inst9 (net9, x1, x3, x4);
endmodule

**Figure 5.20** Module for the majority circuit of Figure 5.19.
//test bench for 5-input majority circuit
module majority_tb;
reg x1, x2, x3, x4, x5;
wire z1;

//apply input vectors
initial
begin: apply_stimulus
  reg [6:0] invect;
  for (invect=0; invect<32; invect=invect+1)
    begin
      {x1, x2, x3, x4, x5} = invect [6:0];
      #10 $display("x1x2x3x4x5 = %b, z1 = %b",
        (x1, x2, x3, x4, x5), z1);
    end
end

//instantiate the module into the test bench
majority inst1 (
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .x4(x4),
  .x5(x5),
  .z1(z1)
);
endmodule

Figure 5.21 Test bench for the majority circuit module of Figure 5.20.
\begin{tabular}{|c|c|}
\hline
\textbf{x1x2x3x4x5} & \textbf{z1} \\
00000 & 0 \\
00001 & 0 \\
00010 & 0 \\
00011 & 0 \\
00100 & 0 \\
00101 & 0 \\
00110 & 0 \\
00111 & 1 \\
01000 & 0 \\
01001 & 0 \\
01010 & 0 \\
01011 & 1 \\
01100 & 0 \\
01101 & 1 \\
01110 & 1 \\
01111 & 1 \\
10000 & 0 \\
10001 & 0 \\
10010 & 0 \\
10011 & 1 \\
10100 & 0 \\
10101 & 1 \\
10110 & 1 \\
10111 & 1 \\
11000 & 0 \\
11001 & 1 \\
11010 & 1 \\
11011 & 1 \\
11100 & 1 \\
11101 & 1 \\
11110 & 1 \\
11111 & 1 \\
\hline
\end{tabular}

Figure 5.22  Outputs for the majority circuit of Figure 5.20.

Page 175

```vhs
// a 4:1 multiplexer using built-in primitives
module mux_4to1 (d, s, enbl, z1);

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

not inst1 (net1, s[0]),
      inst2 (net2, s[1]);

and inst3 (net3, d[0], net1, net2, enbl),
        inst4 (net4, d[1], s[0], net2, enbl),
        inst5 (net5, d[2], net1, s[1], enbl),
        inst6 (net6, d[3], s[0], s[1], enbl);

or inst7 (z1, net3, net4, net5, net6);
endmodule
```

Figure 5.24  Module for a 4:1 multiplexer using built-in primitives.
//test bench for 4:1 multiplexer
module mux_4to1_tb;
reg [3:0] d;
reg [1:0] s;
reg enbl;
wire z1;

initial
$monitor ($time,"ns, select:s=%b, inputs:d=%b, output:z1=%b", s, d, z1);
initial
begin
    #0 s[1]=1'b0;  s[0]=1'b0;
    d[3]=1'b1;  d[2]=1'b0;  d[1]=1'b1;  d[0]=1'b0;
    enbl=1'b1; //d[0]=0; z1=0

    #10 s[1]=1'b0;  s[1]=1'b0;
    d[3]=1'b1;  d[2]=1'b0;  d[1]=1'b1;  d[0]=1'b1;
    enbl=1'b1; //d[0]=1; z1=1

    #10 s[1]=1'b0;  s[0]=1'b1;
    d[3]=1'b1;  d[2]=1'b0;  d[1]=1'b1;  d[0]=1'b1;
    enbl=1'b1; //d[1]=1; z1=1

    #10 s[1]=1'b0;  s[0]=1'b1;
    d[3]=1'b1;  d[2]=1'b0;  d[1]=1'b1;  d[0]=1'b1;
    enbl=1'b1; //d[1]=1; z1=1

    #10 s[1]=1'b0;  s[0]=1'b0;
    d[3]=1'b1;  d[2]=1'b0;  d[1]=1'b1;  d[0]=1'b1;
    enbl=1'b1; //d[1]=1; z1=1

    #10 $stop;
end
//continued on next page

Figure 5.25  Test bench for the 4:1 multiplexer of Figure 5.24.
// instantiate the module into the test bench
mux_4to1 inst1 (  
  .d(d),  
  .s(s),  
  .z1(z1),  
  .enbl(enbl)  
);

endmodule

Figure 5.25 (Continued)

Page 177

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>Select</th>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s=00</td>
<td>d=1010</td>
<td>z1=0</td>
</tr>
<tr>
<td>10</td>
<td>s=00</td>
<td>d=1011</td>
<td>z1=1</td>
</tr>
<tr>
<td>20</td>
<td>s=01</td>
<td>d=1011</td>
<td>z1=1</td>
</tr>
<tr>
<td>30</td>
<td>s=10</td>
<td>d=1011</td>
<td>z1=0</td>
</tr>
<tr>
<td>40</td>
<td>s=01</td>
<td>d=1011</td>
<td>z1=1</td>
</tr>
<tr>
<td>50</td>
<td>s=11</td>
<td>d=1011</td>
<td>z1=1</td>
</tr>
<tr>
<td>60</td>
<td>s=11</td>
<td>d=0011</td>
<td>z1=0</td>
</tr>
</tbody>
</table>

Figure 5.26 Outputs for the 4:1 multiplexer test bench of Figure 5.25.
//set / reset latch using NOR gates
module latch_nor (set, rst, q, qbar);
input set, rst;
output q, qbar;
nor (qbar, set, q);
nor (q, qbar, rst);
endmodule

Figure 5.28 Module for an SR latch using NOR logic.

//test bench for NOR latch
module latch_nor_tb;
reg set, rst;
wire q, qbar;
initial //display signals
$monitor ("set = %b, rst = %b, q = %b, qbar = %b",
set, rst, q, qbar);
initial //apply stimulus
begin
    #0 set = 1'b1; rst = 1'b0;
    #10 set = 1'b0; rst = 1'b0;
    #10 set = 1'b0; rst = 1'b1;
    #10 set = 1'b1; rst = 1'b1;
    #10 $stop;
end
//instantiate the module into the test bench
latch_nor inst1 (.
    .set(set),
    .rst(rst),
    .q(q),
    .qbar(qbar)
);
endmodule

Figure 5.29 Test bench for the SR latch of Figure 5.28.
Figure 5.30 Outputs for the test bench of Figure 5.29 for the SR latch of Figure 5.28.

```
set = 1, rst = 0, q = 1, qbar = 0
set = 0, rst = 0, q = 1, qbar = 0
set = 0, rst = 1, q = 0, qbar = 1
set = 1, rst = 1, q = 0, qbar = 0
```

Figure 5.33 Design module for the 3-bit comparator of Figure 5.32.

```vhdl
//a 3-bit comparator using built-in primitives
module comparator3 (a, b, a_lt_b, a_eq_b, a_gt_b);

input [2:0] a, b;
output a_lt_b, a_eq_b, a_gt_b;

and inst1 (net1, ~a[2], b[2]);
xnor inst2 (net2, a[2], b[2]);
xnor inst3 (net3, a[1], b[1]);
xnor inst4 (net4, a[0], b[0]);
and inst5 (net5, a[2], ~b[2]);
and inst6 (net6, net2, ~a[1], b[1]);
and inst7 (net7, net2, net3, ~a[0], b[0]);
and inst9 (net9, net2, a[1], ~b[1]);
and inst10 (net10, net2, net3, a[0], ~b[0]);

//generate the outputs
or inst11 (a_lt_b, net1, net6, net7);
and inst8 (a_eq_b, net2, net3, net4);
or inst12 (a_gt_b, net5, net9, net10);

endmodule
```
module comparator3_tb;

reg [2:0] a, b;
wire a_lt_b, a_eq_b, a_gt_b;

//apply input vectors
initial
begin: apply_stimulus
    reg [7:0] invect;
    for (invect=0; invect<64; invect=invect+1)
    begin
        {a, b} = invect [7:0];
        #10 $display ("a = %b, b = %b, a_lt_b = %b, a_eq_b = %b, a_gt_b = %b",
            a, b, a_lt_b, a_eq_b, a_gt_b);
    end
end

//instantiate the module into the test bench
comparator3 inst1 (.a(a),
                   .b(b),
                   .a_lt_b(a_lt_b),
                   .a_eq_b(a_eq_b),
                   .a_gt_b(a_gt_b));
endmodule

Figure 5.34  Test bench for the 3-bit comparator of Figure 5.33.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a_lt_b</th>
<th>a_eq_b</th>
<th>a_gt_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>101</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>111</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.35  Partial outputs for the 3-bit comparator of Figure 5.33.
module log_eqn_sop8 (x1, x2, x3, x4, z1);
  input x1, x2, x3, x4;
  output z1;
  and  #3 inst1 (net1, x1, x2);
  and  #3 inst2 (net2, ~x3, x4);
  or    #5 inst3 (z1, net1, net2);
endmodule

Figure 5.37 Module for Equation 5.5 with propagation delays assigned to the gates.

module log_eqn_sop8_tb;
  reg x1, x2, x3, x4;
  wire z1;
  //display variables
  initial $monitor("x1x2x3x4 = %b, z1 = %b", {x1, x2, x3, x4}, z1);
  //apply input vectors
  initial begin
    #0 x1=1'b0; x2=1'b0; x3=1'b0; x4=1'b0;
    #10 x1=1'b1; x2=1'b1; x3=1'b1; x4=1'b0;
    #10 x1=1'b0; x2=1'b1; x3=1'b0; x4=1'b1;
    #10 x1=1'b1; x2=1'b1; x3=1'b1; x4=1'b1;
    #10 x1=1'b0; x2=1'b0; x3=1'b0; x4=1'b0;
    #10 $stop;
  end

Figure 5.38 Test bench for the module of Figure 5.37.
// instantiate the module into the test bench
log_eqn_sop8 inst1 (  
  .x1(x1),  
  .x2(x2),  
  .x3(x3),  
  .x4(x4),  
  .z1(z1)
);

endmodule

Figure 5.38  (Continued)

Page 187

\[ \begin{align*}
  x1x2x3x4 & = 1110, z1 = 0 \quad // z1 will not change until time 18 \\
  x1x2x3x4 & = 1110, z1 = 1 \quad // z1 is 1 after 18 time units \\
  x1x2x3x4 & = 0101, z1 = 1 \quad // z1 does not change \\
  x1x2x3x4 & = 1111, z1 = 1 \quad // z1 does not change \\
  x1x2x3x4 & = 0000, z1 = 1 \quad // z1 will not change until time 48 \\
  x1x2x3x4 & = 0000, z1 = 0 \quad // z1 is 0 at time 48
\end{align*} \]

Figure 5.39  Outputs for the test bench of Figure 5.38.

Page 188

Figure 5.40  Waveforms for the test bench of Figure 5.38 for Equation 5.5.
//half adder using built-in primitives
module half_adder (a, b, sum, cout);
input a, b;
output sum, cout;
xor #4 inst1 (sum, a, b);
and #2 inst2 (cout, a, b);
endmodule

Figure 5.42 Module for the half adder of Figure 5.41.

//test bench for the half adder
module half_adder_tb;
reg a, b;
wire sum, cout;
initial
$monitor("ab = %b, sum = %b, cout = %b", {a, b, sum, cout});

initial  //apply input vectors
begin
  #0 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 $stop;
end

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

Figure 5.43 Test bench for the half adder module of Figure 5.42.
ab = 00, sum = 0, cout = 0 //time unit 0
ab = 01, sum = 0, cout = 0 //time unit 10
ab = 01, sum = 1, cout = 0 //time unit 14
ab = 10, sum = 1, cout = 0 //time unit 20
ab = 11, sum = 1, cout = 0 //time unit 30
ab = 11, sum = 1, cout = 1 //time unit 32
ab = 11, sum = 0, cout = 1 //time unit 34

Figure 5.44    Binary outputs for the half adder module of Figure 5.42.

Figure 5.45    Waveforms for the half adder module of Figure 5.42.
//full adder using built-in primitives with delay
module full_adder_hi_spd (a, b, cin, sum, cout);

input a, b, cin;
output sum, cout;

and #2 inst1 (net1, ~a, ~b, cin);
and #2 inst2 (net2, ~a, b, ~cin);
and #2 inst3 (net3, a, ~b, ~cin);
and #2 inst4 (net4, a, b, cin);
and #2 inst5 (net5, a, b);
and #2 inst6 (net6, a, cin);
and #2 inst7 (net7, b, cin);

or #3 inst8 (sum, net1, net2, net3, net4);
or #3 inst9 (cout, net5, net6, net7);
endmodule

Figure 5.48   Module for the full adder of Figure 5.47 with delays assigned.

//test bench for full adder using
//built-in primitives with delays
module full_adder_hi_spd_tb;

reg a, b, cin;
wire sum, cout;

initial
$monitor ("abcin = %b, sum = %b, cout = %b",
        {a, b, cin}, sum, cout);

//apply input vectors
initial begin
    #0 a=1'b0; b=1'b0; cin=1'b0;
    #10 a=1'b0; b=1'b0; cin=1'b1;
    #10 a=1'b0; b=1'b1; cin=1'b0;
    #10 a=1'b0; b=1'b1; cin=1'b1;
    //continued on next page
end

Figure 5.49   Test bench for the module of Figure 5.48 for the full adder of Figure 5.47.
Figure 5.49  (Continued)

Figure 5.50  Waveforms for the full adder of Figure 5.47 with delays assigned to the logic gates.
//module to show that a narrow pulse will not propagate
//through a gate with a large propagation delay
module no_glitch (x1, x2, out);

input x1, x2;
output out;

and #10   (out, x1, x2);
endmodule

Figure 5.51 Module to illustrate inertial delay.

//test bench for the no_glitch module
module no_glitch_tb;

reg x1, x2;
wire out;

//apply input vectors
initial begin
    #0   x1=1'b0; x2=1'b0;
        $display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

    #10  x1 = 1'b1; x2 = 1'b1;
        $display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

    #10  x1 = 1'b1; x2 = 1'b1;
        $display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

    #10  x1 = 1'b0; x2 = 1'b1;
        $display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

    #10  x1 = 1'b0; x2 = 1'b1;
        $display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

    #3   x1 = 1'b1; x2 = 1'b1;
        $display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

    #4   x1 = 1'b0; x2 = 1'b1;
        $display ($time, " x1x2=%b, out=%b", {x1, x2}, out);
endmodule

Figure 5.52 Test bench to illustrate inertial delay.
#3  x1 = 1'b0; x2 = 1'b1;
$display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

#10 x1 = 1'b0; x2 = 1'b1;
$display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

#10 x1 = 1'b1; x2 = 1'b1;
$display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

#10 x1 = 1'b1; x2 = 1'b1;
$display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

#10 x1 = 1'b1; x2 = 1'b1;
$display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

#3  x1 = 1'b1; x2 = 1'b0;
$display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

#10 x1 = 1'b1; x2 = 1'b1;
$display ($time, " x1x2=%b, out=%b", {x1, x2}, out);

$stop;

end

//instantiate the module into the test bench
no_glitch inst1 (  
   .x1(x1),
   .x2(x2),
   .out(out)
);

endmodule

Figure 5.52  (Continued)
Page 198

|   | x1x2=00, out=0   |   | x1x2=11, out=0   |   | x1x2=11, out=1   |   | x1x2=01, out=1   |   | x1x2=11, out=0   |   | x1x2=01, out=0   |   | x1x2=01, out=0   |   | x1x2=01, out=0   |   | x1x2=11, out=0   |   | x1x2=11, out=1   |   | x1x2=10, out=1   |   | x1x2=11, out=1   |   | x1x2=11, out=1   |   | x1x2=11, out=1   |   | x1x2=11, out=1   |
|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|---|----------------|
| 0 | x1x2=00, out=0 | 10|x1x2=11, out=0 | 20|x1x2=11, out=1 | 30|x1x2=01, out=1 | 40|x1x2=01, out=0 | 43|x1x2=11, out=0 | 47|x1x2=01, out=0 | 50|x1x2=01, out=0 | 60|x1x2=01, out=0 | 70|x1x2=11, out=0 | 80|x1x2=11, out=1 | 83|x1x2=10, out=1 | 87|x1x2=11, out=1 | 97|x1x2=11, out=1 | 107|x1x2=11, out=1 |

**Figure 5.53**  Outputs for the inertial delay module of Figure 5.51.

Page 199

**Figure 5.54**  Waveforms for the inertial delay module of Figure 5.51.
//module to illustrate inertial delay and
//transport delay using built-in primitives
module inert_trans_dly1 (x1, x2, x3, x4, net1, net2, net3, z1);

input x1, x2, x3, x4;
output net1, net2, net3, z1;

//transport delay
wire #2 net1, net2, net3;

//inertial delay
and #3 inst1 (net1, x1, ~x2);
xor #5 inst2 (net2, x3, x4);
or #2 inst3 (net3, net1, net2);
buf #1 inst4 (z1, net3);
endmodule

Figure 5.57  Module to model inertial delay and transport delay.

//module path delay
module module_path_dly (x1, x2, x3, x4, z1);

input x1, x2, x3, x4;
output z1;

and   inst1 (net1, x1, x2);
xor   inst2 (net2, x3, x4);
or    inst3 (z1, net1, net2);

specify
  (x1, x2 => z1) = 6;
  (x3, x4 => z1) = 8;
endspecify
endmodule

Figure 5.60  Module to illustrate module path delay.
// test bench for module path delay
module module_path_dly_tb;
reg x1, x2, x3, x4;
wire z1;

initial $monitor ("x1x2x3x4 = %b, z1 = %b", {x1, x2, x3, x4}, z1);
initial //apply input vectors
begin
  #0 x1=1'b0; x2=1'b0; x3=1'b0; x4=1'b0;
  #10 x1=1'b1; x2=1'b1; x3=1'b0; x4=1'b0;
  #10 x1=1'b0; x2=1'b0; x3=1'b0; x4=1'b0;
  #10 x1=1'b0; x2=1'b0; x3=1'b1; x4=1'b0;
  #10 $stop;
end
module_path_dly inst1 ( // instantiate the module
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .x4(x4),
  .z1(z1)
);
endmodule

Figure 5.61 Test bench for module path delay.

Figure 5.62 Waveforms to illustrate module path delay.
module sngl_bit_detect (x1, x2, x3, x4, z1);
input x1, x2, x3, x4;
output z1;

//cell 1 ************************************************
not inst1 (net1, x1);

//cell 2 ************************************************
not inst2 (net2, x2);
and inst3 (net3, net2, x1);
and inst4 (net4, x2, net1);
and inst5 (net5, net2, net1);
or inst6 (net6, net3, net4);

//cell 3 ************************************************
not inst7 (net7, x3);
and inst8 (net8, net7, net6);
and inst9 (net9, x3, net5);
and inst10 (net10, net7, net5);
or inst11 (net11, net8, net9);

//cell 4 ************************************************
not inst12 (net12, x4);
and inst13 (net13, net12, net11);
and inst14 (net14, x4, net10);
or inst15 (z1, net13, net14);
endmodule

Figure 5.66 Design module for the iterative network of Figure 5.65.
// test bench for single bit detection
module sngl_bit_detect_tb;

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

initial // apply input vectors
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

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

Figure 5.67 Test bench for the iterative network module of Figure 5.66.

<table>
<thead>
<tr>
<th>x1x2x3x4</th>
<th>z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, z1 = 0</td>
<td>x1x2x3x4 = 1000, z1 = 1</td>
</tr>
<tr>
<td>0001, z1 = 1</td>
<td>x1x2x3x4 = 1001, z1 = 0</td>
</tr>
<tr>
<td>0010, z1 = 1</td>
<td>x1x2x3x4 = 1010, z1 = 0</td>
</tr>
<tr>
<td>0011, z1 = 0</td>
<td>x1x2x3x4 = 1011, z1 = 0</td>
</tr>
<tr>
<td>0100, z1 = 1</td>
<td>x1x2x3x4 = 1100, z1 = 0</td>
</tr>
<tr>
<td>0101, z1 = 0</td>
<td>x1x2x3x4 = 1101, z1 = 0</td>
</tr>
<tr>
<td>0110, z1 = 0</td>
<td>x1x2x3x4 = 1110, z1 = 0</td>
</tr>
<tr>
<td>0111, z1 = 0</td>
<td>x1x2x3x4 = 1111, z1 = 0</td>
</tr>
</tbody>
</table>

Figure 5.68 Outputs for the iterative network module of Figure 5.66.
Figure 5.69  Waveforms for the iterative network module of Figure 5.66.
module sngl_bit_detect1 (x1, x2, x3, x4, z1);

input x1, x2, x3, x4;
output z1;

//cell 1 ************************************************
not inst1 (net1, x1);
and inst2 (net2, net1, 1'b0);
and inst3 (net3, x1, 1'b1);
and inst4 (net4, net1, 1'b1);
or inst5 (net5, net2, net3);

//cell 2 ************************************************
not inst6 (net6, x2);
and inst7 (net7, net6, net5);
and inst8 (net8, x2, net4);
and inst9 (net9, net6, net4);
or inst10 (net10, net7, net8);

//cell 3 ************************************************
not inst11 (net11, x3);
and inst12 (net12, net11, net10);
and inst13 (net13, x3, net9);
and inst14 (net14, net11, net9);
or inst15 (net15, net12, net13);

//cell 4 ************************************************
not inst16 (net16, x4);
and inst17 (net17, net16, net15);
and inst18 (net18, x4, net14);
and inst19 (net19, net16, net14);
or inst20 (z1, net17, net18);
endmodule

Figure 5.72 Design module for the single-bit detection circuit of Example 5.14 using four identical cells.
// test bench for single bit detection
module sngl_bit_detect1_tb;

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

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

// instantiate the module into the test bench
sngl_bit_detect1 inst1 (  
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .x4(x4),
  .z1(z1)
);
endmodule

Figure 5.73  Test bench for the single-bit detection module of Figure 5.72.

<table>
<thead>
<tr>
<th>x1x2x3x4</th>
<th>z1</th>
<th>x1x2x3x4</th>
<th>z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1011</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>1100</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1110</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>1111</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.74  Outputs for the single-bit detection module of Figure 5.72.
Figure 5.75  Waveforms for the single-bit detection module of Figure 5.72.

Page 215

//typical cell for single-bit detection
module sngl_bit_cell (x1_in, y1_in, y0_in, y1_out, y0_out);

input x1_in, y1_in, y0_in;
output y1_out, y0_out;

not  inst1 (net1, x1_in);
and  inst2 (net2, net1, y1_in);
and  inst3 (net3, x1_in, y0_in);
and  inst4 (y0_out, net1, y0_in);
or  inst5 (y1_out, net2, net3);

endmodule

Figure 5.79  Typical cell that is instantiated four times to detect a single bit in an input vector.
module sngl_bit_detect2 (x1, x2, x3, x4, z1);
input x1, x2, x3, x4;
output z1;

// instantiate the single-bit cell modules
// cell 1 ************************************************
  sngl_bit_cell inst1(
    .x1_in(x1),
    .y1_in(1'b0),
    .y0_in(1'b1),
    .y1_out(net1_1),
    .y0_out(net1_0)
  );

// cell 2 ************************************************
  sngl_bit_cell inst2(
    .x1_in(x2),
    .y1_in(net1_1),
    .y0_in(net1_0),
    .y1_out(net2_1),
    .y0_out(net2_0)
  );

// cell 3 ************************************************
  sngl_bit_cell inst3(
    .x1_in(x3),
    .y1_in(net2_1),
    .y0_in(net2_0),
    .y1_out(net3_1),
    .y0_out(net3_0)
  );

// cell 4 ************************************************
  sngl_bit_cell inst4(
    .x1_in(x4),
    .y1_in(net3_1),
    .y0_in(net3_0),
    .y1_out(z1)
  );
endmodule

Figure 5.80 Module to detect a single bit in a 4-bit input vector x[1:4] in which the typical cell of Figure 5.79 is instantiated four times.
// test bench for the single-bit detection
// using a typical cell instantiation
module sngl_bit_detect2_tb;

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

// apply input vectors
initial begin:
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
sngl_bit_detect2 inst1 (.x1(x1),
.x2(x2),
.x3(x3),
.x4(x4),
.z1(z1))
);
endmodule

Figure 5.81  Test bench for the single bit detection module of Figure 5.80.

<table>
<thead>
<tr>
<th>x1x2x3x4 = 0000, z1 = 0</th>
<th>x1x2x3x4 = 1000, z1 = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1x2x3x4 = 0001, z1 = 1</td>
<td>x1x2x3x4 = 1001, z1 = 0</td>
</tr>
<tr>
<td>x1x2x3x4 = 0010, z1 = 1</td>
<td>x1x2x3x4 = 1010, z1 = 0</td>
</tr>
<tr>
<td>x1x2x3x4 = 0011, z1 = 0</td>
<td>x1x2x3x4 = 1011, z1 = 0</td>
</tr>
<tr>
<td>x1x2x3x4 = 0100, z1 = 1</td>
<td>x1x2x3x4 = 1100, z1 = 0</td>
</tr>
<tr>
<td>x1x2x3x4 = 0101, z1 = 0</td>
<td>x1x2x3x4 = 1101, z1 = 0</td>
</tr>
<tr>
<td>x1x2x3x4 = 0110, z1 = 0</td>
<td>x1x2x3x4 = 1110, z1 = 0</td>
</tr>
<tr>
<td>x1x2x3x4 = 0111, z1 = 0</td>
<td>x1x2x3x4 = 1111, z1 = 0</td>
</tr>
</tbody>
</table>

Figure 5.82  Outputs for the single-bit detector module of Figure 5.80.
Figure 5.83  Waveforms for the single-bit detector module of Figure 5.80.

Figure 5.86  Eight-bit priority encoder module.
//test bench for 8-bit priority encoder
module priority_encoder_tb;
reg x0, x1, x2, x3, x4, x5, x6, x7, enbl;
wire z1, z2, z4, valid;

initial //display variables
$monitor("x0x1x2x3x4x5x6x7 = %b, z4z2z1 = %b, valid = %b",
{x0, x1, x2, x3, x4, x5, x6, x7}, {z4, z2, z1}, valid);
initial //apply input vectors
begin
#0 x0=1'b0;   x1=1'b0;   x2=1'b0;   x3=1'b0;
    x4=1'b0;   x5=1'b0;   x6=1'b0;   x7=1'b0;  enbl=1'b1;

#10 x0=1'b0;   x1=1'b0;   x2=1'b1;   x3=1'b0;
    x4=1'b0;   x5=1'b0;   x6=1'b0;   x7=1'b0;  enbl=1'b1;

#10 x0=1'b0;   x1=1'b0;   x2=1'b0;   x3=1'b0;
    x4=1'b0;   x5=1'b1;   x6=1'b0;   x7=1'b0;  enbl=1'b1;

#10 x0=1'b0;   x1=1'b0;   x2=1'b1;   x3=1'b0;
    x4=1'b0;   x5=1'b0;   x6=1'b1;   x7=1'b0;  enbl=1'b1;

#10 x0=1'b1;   x1=1'b0;   x2=1'b1;   x3=1'b1;
    x4=1'b0;   x5=1'b0;   x6=1'b0;   x7=1'b0;  enbl=1'b1;

#10 x0=1'b1;   x1=1'b1;   x2=1'b1;   x3=1'b1;
    x4=1'b1;   x5=1'b1;   x6=1'b1;   x7=1'b1;  enbl=1'b1;

#10 $stop;
end

priority_encoder inst1 ( //instantiate the module
    .x0(x0),
    .x1(x1),
    .x2(x2),
    .x3(x3),
    .x4(x4),
    .x5(x5),
    .x6(x6),
    .x7(x7),
    .enbl(enbl),
    .z1(z1),
    .z2(z2),
    .z4(z4),
    .valid(valid)
    );
endmodule

Figure 5.87 Test bench for the 8-bit priority encoder module of Figure 5.86.
\begin{itemize}
\item $x_0x_1x_2x_3x_4x_5x_6x_7 = 10000000, z_4z_2z_1 = 000, \text{ valid } = 1$
\item $x_0x_1x_2x_3x_4x_5x_6x_7 = 00000000, z_4z_2z_1 = 000, \text{ valid } = 0$
\item $x_0x_1x_2x_3x_4x_5x_6x_7 = 00100000, z_4z_2z_1 = 010, \text{ valid } = 1$
\item $x_0x_1x_2x_3x_4x_5x_6x_7 = 00000100, z_4z_2z_1 = 101, \text{ valid } = 1$
\item $x_0x_1x_2x_3x_4x_5x_6x_7 = 00100010, z_4z_2z_1 = 110, \text{ valid } = 1$
\item $x_0x_1x_2x_3x_4x_5x_6x_7 = 10110000, z_4z_2z_1 = 011, \text{ valid } = 1$
\item $x_0x_1x_2x_3x_4x_5x_6x_7 = 11111111, z_4z_2z_1 = 111, \text{ valid } = 1$
\end{itemize}

**Figure 5.88** Outputs for the 8-bit priority encoder module of Figure 5.86.

**Figure 5.89** Waveforms for the 8-bit priority encoder of Figure 5.86.