Verilog HDL: Digital Design and Modeling

Chapter 2

Overview
// dataflow and gate with two inputs
module and2 (x1, x2, z1);

input x1, x2;
output z1;
wire x1, x2;
wire z1;

assign z1 = x1 & x2;
endmodule

Figure 2.11 Verilog module for an AND gate with two inputs.
Figure 2.12  Test bench for the 2-input AND gate of Figure 2.11.
Figure 2.13  Binary outputs for the test bench of Figure 2.12 for a 2-input AND gate.

\begin{align*}
x_1 &= 0, \ x_2 = 0, \ z_1 = 0 \\
x_1 &= 0, \ x_2 = 1, \ z_1 = 0 \\
x_1 &= 1, \ x_2 = 0, \ z_1 = 0 \\
x_1 &= 1, \ x_2 = 1, \ z_1 = 1
\end{align*}

Figure 2.14  Waveforms for the test bench of Figure 2.12 for a 2-input AND gate.
Verilog code for the 2-input exclusive-OR gate of Figure 2.17.

```verilog
//dataflow 2-input exclusive-or gate
module xor2 (x1, x2, z1);

input x1, x2;
output z1;

wire x1, x2;
wire z1;

assign z1 = x1 ^ x2;
endmodule
```

Figure 2.18  Verilog code for the 2-input exclusive-OR gate of Figure 2.17.

Test bench for the 2-input exclusive-OR gate module of Figure 2.18.

```verilog
//2-input exclusive-or gate test bench
module xor2_tb;

reg x1, x2;
wire z1;

//apply input vectors
initial
begin: apply_stimulus
    reg [2:0] invect;
    for (invect = 0; invect < 4; invect = invect + 1)
    begin
        {x1, x2} = invect [2:0];
        #10 $display "%{x1x2} = %b, z1 = %b",
            {x1, x2}, z1);
    end
end

//instantiate the module into the test bench
xor2 inst1 (  
    .x1(x1),
    .x2(x2),
    .z1(z1)
);
endmodule
```

Figure 2.19  Test bench for the 2-input exclusive-OR gate module of Figure 2.18.
Figure 2.20  Binary outputs for the test bench of Figure 2.19.

\[
\begin{align*}
(x_1x_2) &= 00, \quad z_1 = 0 \\
(x_1x_2) &= 01, \quad z_1 = 1 \\
(x_1x_2) &= 10, \quad z_1 = 1 \\
(x_1x_2) &= 11, \quad z_1 = 0
\end{align*}
\]

Figure 2.21  Waveforms for the 2-input exclusive-OR gate test bench of Figure 2.19.
Figure 2.23 Verilog code for the logic circuit of Figure 2.22.

```verilog
//dataflow with delay
`timescale 10ns / 1ns
module four_and_delay (x1, x2, z1);
input x1, x2;
output [3:0] z1;
//dataflow with delay
`timescale 10ns / 1ns
module four_and_delay (x1, x2, z1);
input x1, x2;
output [3:0] z1;
wire x1, x2;
wire z1;
assign #2 z1[0] = ~x1 & ~x2;
assign #2 z1[1] = ~x1 & x2;
assign #2 z1[2] = x1 & ~x2;
assign #2 z1[3] = x1 & x2;
endmodule
```

Figure 2.24 Test bench for the Verilog code of Figure 2.23 for the circuit of Figure 2.22.

```verilog
//four_and_delay test bench
module four_and_delay_tb;
reg x1, x2;
wire [3:0] z1;
initial
$monitor("x1 x2=%b, z1=%b",
{x1, x2}, z1);
//apply input vectors
initial begin
#0 x1 = 1'b0;
x2 = 1'b0;
#5 x1 = 1'b0;
x2 = 1'b1;
#5 x1 = 1'b1;
x2 = 1'b0;
#5 x1 = 1'b0;
x2 = 1'b1;
#5 $stop;
end
//instantiate the module
//into the test bench
four_and_delay inst1 (.
x1(x1),
.x2(x2),
.z1(z1)
);
endmodule
```
Figure 2.25  Waveforms for the four AND gates with delay using the test bench of Figure 2.24.
Figure 2.27 Behavioral module for the 3-input OR gate of Figure 2.26.

```verilog
//behavioral 3-input or gate
module or3 (x1, x2, x3, z1);

input x1, x2, x3;
output z1;
wire x1, x2, x3;
reg z1;

always @ (x1 or x2 or x3)
begin
  z1 = x1 | x2 | x3;
end
endmodule
```

Figure 2.28 Test bench for the 3-input OR gate module of Figure 2.27.

```verilog
//or3 test bench
module or3_tb;

reg x1, x2, x3;
wire z1;

//apply input vectors
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 "%{x1x2x3} = %b, z1 = %b", 
    {x1, x2, x3}, z1);
  end
end

//instantiate the module into the test bench
or3 inst1 ( 
  .x1(x1),
  .x2(x2),
  .x3(x3),
  .z1(z1)
);
endmodule
```
Page 29

\[
\begin{align*}
(x_1 \land x_2 \land x_3) &= 000, \quad z_1 = 0 \\
(x_1 \land x_2 \land x_3) &= 001, \quad z_1 = 1 \\
(x_1 \land x_2 \land x_3) &= 010, \quad z_1 = 1 \\
(x_1 \land x_2 \land x_3) &= 011, \quad z_1 = 1 \\
(x_1 \land x_2 \land x_3) &= 100, \quad z_1 = 1 \\
(x_1 \land x_2 \land x_3) &= 101, \quad z_1 = 1 \\
(x_1 \land x_2 \land x_3) &= 110, \quad z_1 = 1 \\
(x_1 \land x_2 \land x_3) &= 111, \quad z_1 = 1
\end{align*}
\]

**Figure 2.29**  Binary outputs for the test bench of Figure 2.28.

Page 30

**Figure 2.30**  Waveforms for the test bench of Figure 2.28 for the 3-input OR gate module of Figure 2.26.
//behavioral intrasegment delay example
`timescale 10ns / 1ns
module intra_stmt_dly_delay (x1,x2);

output x1, x2;
reg x1, x2;

initial
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 2.31 Behavioral module to generate waveforms using intrastatement delays.

Figure 2.32 Waveforms showing intrastatement delays for Figure 2.31.
Figure 2.34  Behavioral module for a 4-bit adder.

```verilog
//behavioral 4-bit adder
module adder_4_behav (a, b, cin, sum);

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

output [4:0] sum;

wire [3:0] a, b;
wire cin;
reg [4:0] sum;

always @ (a or b or cin)
begin
    sum = a + b + cin;
end

endmodule
```

Figure 2.35  Test bench for the 4-bit adder module of Figure 2.34.

```verilog
//behavioral 4-bit adder test
//bench
module adder_4_behav_tb;

reg [3:0] a, b;
reg cin;
wire [4:0] sum;

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

//apply input vectors
initial
begin
    #0 a = 4'b0011;
b = 4'b0100;
cin = 1'b0;
    #0 a = 4'b1100;
b = 4'b0011;
cin = 1'b0;
    #0 a = 4'b0111;
b = 4'b0110;
cin = 1'b1;
    #0 a = 4'b1001;
b = 4'b0111;
cin = 1'b1;
    #0 a = 4'b1101;
b = 4'b0111;
cin = 1'b1;
end

//instantiate the module into
//the test bench
adder_4_behav inst1 (.
a(a),
.b(b),
.cin(cin),
.sum(sum)
);

endmodule
```
Figure 2.36  Binary outputs for the 4-bit adder obtained from the test bench of Figure 2.35.

<table>
<thead>
<tr>
<th>Input</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011_0100_0</td>
<td>00111</td>
</tr>
<tr>
<td>1100_0011_0</td>
<td>01111</td>
</tr>
<tr>
<td>0111_0110_1</td>
<td>01110</td>
</tr>
<tr>
<td>1001_0111_1</td>
<td>10001</td>
</tr>
<tr>
<td>1101_0111_1</td>
<td>10101</td>
</tr>
<tr>
<td>1111_0110_1</td>
<td>10110</td>
</tr>
</tbody>
</table>

Figure 2.37  Waveforms for the 4-bit adder module of Figure 2.34.
Verilog code for a modulo-16 synchronous counter.

```verilog
//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

//modulo-16 counter test bench
module ctr_mod_16_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
  #320 $stop;
end

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

Test bench for the modulo-16 counter of Figure 2.39.
Figure 2.41  Binary outputs for the modulo-16 counter of Figure 2.39 obtained from the test bench of Figure 2.40.

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

Figure 2.42  Waveforms for the modulo-16 counter of Figure 2.39 obtained from the test bench of Figure 2.40.
Figure 2.44  Modules for \textit{and2, and3, and4, and or3} that will be instantiated into the sum-of-products structural module of Figure 2.45.
//structural sum of products
module sop (x1, x2, x3, x4, z1);

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

//define internal nets
wire net1, net2, net3, net4;
wire z1;

assign z1 = net4;

//instantiate the gate modules
//into the structural module
and2 inst1 (  
  .x1(x1),
  .x2(x2),
  .z1(net1)
);

and3 inst2 (  
  .x1(x2),
  .x2(~x3),
  .x3(x4),
  .z1(net2)
);

and4 inst3 (  
  .x1(~x1),
  .x2(~x2),
  .x3(x3),
  .x4(x4),
  .z1(net3)
);

or3 inst4 (  
  .x1(net1),
  .x2(net2),
  .x3(net3),
  .z1(net4)
);

endmodule

Figure 2.45  Structural module for the sum-of-products equation of Equation 2.4.
//structural sum of products test bench
module sop_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];
      #10 $display "{x1x2x3x4} = %b, z1 = %b", {x1, x2, x3, x4}, z1);
    end
end

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

Figure 2.46  Test bench for the structural module of Figure 2.45.

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

Figure 2.47  Binary outputs for the test bench of Figure 2.46.
Figure 2.49  Waveforms for Figure 2.45 using the test bench of Figure 2.46.
Figure 2.53  Verilog code for the half adder of Figure 2.51.

```verilog
//dataflow half_adder
module half_adder_df (a, b, sum, cout);

input a, b; //list which are input
output sum, cout; //list which are output
wire a, b, sum, cout; //all are wire
assign sum = a ^ b;
assign cout = a & b;
endmodule
```

Figure 2.54  Test bench for the half adder module of Figure 2.53.

```verilog
//dataflow half_adder test bench
module half_adder_df_tb;

reg a, b; //inputs are reg for test bench
wire sum, cout; //outputs are wire for test bench

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

initial 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 dataflow module into the test bench
half_adder_df inst1 (.
    a(a),
    b(b),
    sum(sum),
    cout(cout)
);
endmodule
```
Figure 2.55  Binary outputs obtained from the test bench of Figure 2.54 for the half adder module of Figure 2.53.

Page 46

Figure 2.56  Waveforms for the half adder of Figure 2.53.
// structural full adder
module full_adder_struc (a, b, cin, sum, cout);

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

wire [1:0] ha_sum, ha_cy;

// instantiate the half adder
half_adder_df inst1 (  
  .a(a),
  .b(b),
  .sum(ha_sum[1]),
  .cout(ha_cy[1])
);

half_adder_df inst2 (  
  .a(ha_sum[1]),
  .b(cin),
  .sum(ha_sum[0]),
  .cout(ha_cy[0])
);

assign sum = ha_sum[0];
assign cout = ha_cy[0] | ha_cy[1];
endmodule

Figure 2.57 Structural module for the full adder.
```verilog
//structural full adder test
//bench
module full_adder_struc_tb;

//inputs are reg for tb
reg a, b, cin;

//outputs are wire for tb
wire sum, cout;

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

initial
begin
  #0 a = 1'b0;
  b = 1'b0;
  cin = 1'b0;

  #10a = 1'b0;
  b = 1'b0;
  cin = 1'b1;

  #10a = 1'b0;
  b = 1'b1;
  cin = 1'b0;

  #10a = 1'b0;
  b = 1'b1;
  cin = 1'b1;

  #10a = 1'b1;
  b = 1'b0;
  cin = 1'b0;

  #10a = 1'b1;
  b = 1'b0;
  cin = 1'b1;

  #10a = 1'b1;
  b = 1'b1;
  cin = 1'b0;
  $stop;
end

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

**Figure 2.58** Test bench for the full adder module of Figure 2.57.
Figure 2.59  Binary outputs for the test bench of Figure 2.58.

<table>
<thead>
<tr>
<th>Input</th>
<th>Sum</th>
<th>Carry Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.60  Waveforms for the test bench of Figure 2.58 for the structural full adder of Figure 2.57.
Figure 2.62  Verilog code for a full adder.

//dataflow full adder
module full_adder (a, b, cin, sum, cout);

//list inputs and outputs
input a, b, cin;
output sum, cout;

//define wires
wire a, b, cin;
wire sum, cout;

//continuous assign
assign sum = (a ^ b) ^ cin;
assign cout = cin & (a ^ b) | (a & b);

endmodule

Figure 2.63  Test bench for the full adder module of Figure 2.62.

//full adder test bench
module full_adder_tb;

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

//apply input vectors
initial begin: apply_stimulus
reg [3:0] invect;
for (invect = 0; invect < 8; invect = invect + 1)
begin
    {a, b, cin} = invect [3:0];
    #10 $display "{abcin} = %b, sum = %b, cout = %b",
    (a, b, cin), sum, cout);
end
end

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

endmodule
Figure 2.64  Binary outputs for the full adder of Figure 2.62 obtained from the test bench of Figure 2.63.

<table>
<thead>
<tr>
<th>Input</th>
<th>Sum</th>
<th>Carry Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.65  Waveforms for the full adder of Figure 2.62 obtained from the test bench of Figure 2.63.
module ripple_adder_4 (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;
wire [3:0] sum;
wire cout;
wire [3:0] c;//define internal carries

assign cout = c[3];

//instantiate the full adder
full_adder inst1 (  
    .a(a[0]),  
    .b(b[0]),  
    .cin(cin),  
    .sum(sum[0]),  
    .cout(c[0])  
);

full_adder inst2 (  
    .a(a[1]),  
    .b(b[1]),  
    .cin(c[0]),  
    .sum(sum[1]),  
    .cout(c[1])  
);

full_adder inst3 (  
    .a(a[2]),  
    .b(b[2]),  
    .cin(c[1]),  
    .sum(sum[2]),  
    .cout(c[2])  
);

full_adder inst4 (  
    .a(a[3]),  
    .b(b[3]),  
    .cin(c[2]),  
    .sum(sum[3]),  
    .cout(c[3])  
);
endmodule

Figure 2.67 Structural module for a 4-bit ripple adder.
// 4-bit ripple adder test
// bench
module ripple_adder_4_tb;

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

wire [3:0] sum;
wire cout;

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

// apply input vectors
initial
begin
  #0 a = 4'b0011;
b = 4'b0100;
cin = 1'b0;

  #10 a = 4'b1001;
b = 4'b0111;
cin = 1'b1;

  #10 a = 4'b1101;
b = 4'b0111;
cin = 1'b1;

  #10 a = 4'b1111;
b = 4'b0110;
cin = 1'b1;

  #10 $stop;
end

// instantiate the module into
// the test bench
ripple_adder_4 inst1 {
  .a(a),
  .b(b),
  .cin(cin),
  .sum(sum),
  .cout(cout)
};

endmodule

Figure 2.68 Test bench module for the 4-bit ripple adder of Figure 2.67.
Figure 2.69  Binary outputs obtained from the adder test bench of Figure 2.68.

Figure 2.70  Waveforms for the 4-bit ripple adder test bench of Figure 2.68. The values for the variables are shown in hexadecimal.
//mixed-design full adder
module full_adder_mixed (a, b, cin, sum, cout);

//list inputs and outputs
input a, b, cin;
output sum, cout;

//define reg and wires
reg cout;
wire a, b, cin;
wire sum;
wire net1;

//built-in primitive
xor (net1, a, b);

//behavioral
always @ (a or b or cin)
begin
    cout = cin & (a ^ b) | (a & b);
end

//dataflow
assign sum = net1 ^ cin;
endmodule

Figure 2.72 Verilog module for the full adder of Figure 2.71 using mixed-design modeling.
module full_adder_mixed_tb;

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

//display variables
initial
$monitor ("a b cin = %b %b %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;
  #10 a = 1'b1; b = 1'b0; cin = 1'b0;
  #10 a = 1'b1; b = 1'b0; cin = 1'b1;
  #10 a = 1'b1; b = 1'b1; cin = 1'b0;
  #10 a = 1'b1; b = 1'b1; cin = 1'b1;
  #10 $stop;
end

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

Figure 2.73  Test bench for the mixed-design full adder of Figure 2.72.
Figure 2.74  Binary outputs for the mixed-design full adder of Figure 2.72.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>sum</th>
<th>cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.75  Waveforms for the mixed-design full adder of Figure 2.72.