Introduction to Digital Design
Using Digilent FPGA Boards
─ Block Diagram / Verilog Examples

Richard E. Haskell
Darrin M. Hanna

Oakland University, Rochester, Michigan

NOTE: For the FPGA Labs you will NOT be using the Altec Active HDL software. Instead, you will be using the Xilinx tool chain as described in the Screen-shot document.

LBE Books
Rochester Hills, MI

Example 3

Multiple-Input Gates

In this example we will design a circuit containing multiple-input gates. We will create a logic circuit containing 4-input AND, OR, and XOR gates. We will leave it as a problem for you to create a logic circuit containing 4-input NAND, NOR, and XNOR gates.

Prerequisite knowledge:
Appendix C – Basic Logic Gates
Appendix A – Use of Aldec Active-HDL

3.1 Behavior of Multiple-Input Gates

The AND, OR, NAND, NOR, XOR, and XNOR gates we studied in Example 1 had two inputs. The basic definitions hold for multiple inputs. A multiple-input AND gate is shown in Fig. 2.19. The output of an AND gate is HIGH only if all inputs are HIGH. There are three ways we could describe this multiple-input AND gate in Verilog. First we could simply write the logic equation as

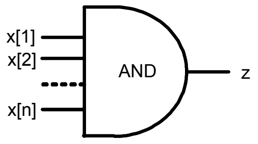


Figure 3.1
Multiple-input AND gate.

.
```
assign z = x[1] & x[2] & ... & x[n];          (3.1)
```

Alternatively, we could use the & symbol as a *reduction operator* by writing

```
assign z = &x;                                (3.2)
```

This produces the same result as the statement (3.1) with much less writing. Finally, we could use the following *gate instantiation statement* for an AND gate.

```
and(z,x[1],x[2],...,x[n]);                    (3.3)
```

In this statement the first parameter in the parentheses is the name of the output port. This is followed by a list of all input signals.

A multiple-input OR gate is shown in Fig. 3.2. The output of an OR gate is LOW only if all inputs are LOW. Just as with the AND gate there are three ways we can describe this multiple-input OR gate in Verilog. We can write the logic equation as

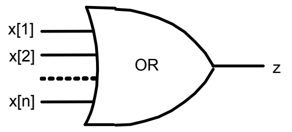


Figure 3.2
Multiple-input OR gate.

.
```
assign z = x[1] | x[2] | ... | x[n];
```

or we can use the | symbol as a *reduction operator* by writing

```
assign z = |x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
or(z,x[1],x[2],...,x[n]);
```

A multiple-input NAND gate is shown in Fig. 3.3. The output of a NAND gate is LOW only if all inputs are HIGH. We can write the logic equation as
.

```
assign z = ~(x[1] & x[2] & ... & x[n]);
```

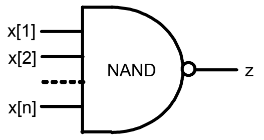


Figure 3.3
Multiple-input NAND gate.

or we can use the ~& symbol as a *reduction operator* by writing

```
assign z = ~&x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
nand(z,x[1],x[2],...,x[n]);
```

A multiple-input NOR gate is shown in Fig. 3.4. The output of a NOR gate is HIGH only if all inputs are LOW. We can write the logic equation as
.

```
assign z = ~(x[1] | x[2] | ... | x[n]);
```

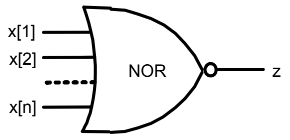


Figure 3.4
Multiple-input NOR gate.

or we can use the ~| symbol as a *reduction operator* by writing

```
assign z = ~|x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
nor(z,x[1],x[2],...,x[n]);
```

A multiple-input XOR gate is shown in Fig. 3.5. What is the meaning of this multiple-input gate? Following the methods we used for the previous multiple-input gates we can write the logic equation as
.

```
assign z = x[1] ^ x[2] ^ ... ^ x[n];
```

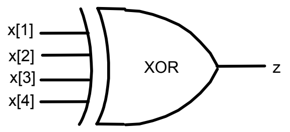


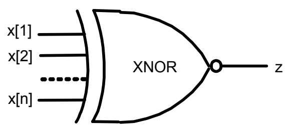
Figure 3.5
Multiple-input XOR gate.

or we can use the ^ symbol as a *reduction operator* by writing

```
assign z = ^x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
xor(z,x[1],x[2],...,x[n]);
```



We will create a 4-input XOR gatge in this example to determine its meaning but first consider the multiple-input XNOR gate shown in Fig. 3.6. What is the meaning of this multiple-input gate? (See the problelm at the end of this

Figure 3.6
Multiple-input XNOR gate.

example for the answer.) Following the methods we used for the previous multiple-input gates we can write the logic equation as

.

```
        assign z = ~(x[1] ^ x[2] ^ ... ^ x[n]);
```

or we can use the ~^ symbol as a *reduction operator* by writing

```
        assign z = ~^x;
```

or we can use the following gate *instantiation statement* for an XOR gate.

```
        xnor(z,x[1],x[2],...,x[n]);
```

3.2 Generating the Design File *gates4.bde*

Use the block diagram editor (BDE) in Active-HDL to create the logic circuit called *gates4.bde* shown in Fig. 3.7. A simulation of this circuit is shown in Fig. 3.8. From this simulation we see that *the output of an XOR gate is HIGH only if the number of HIGH inputs is ODD*.

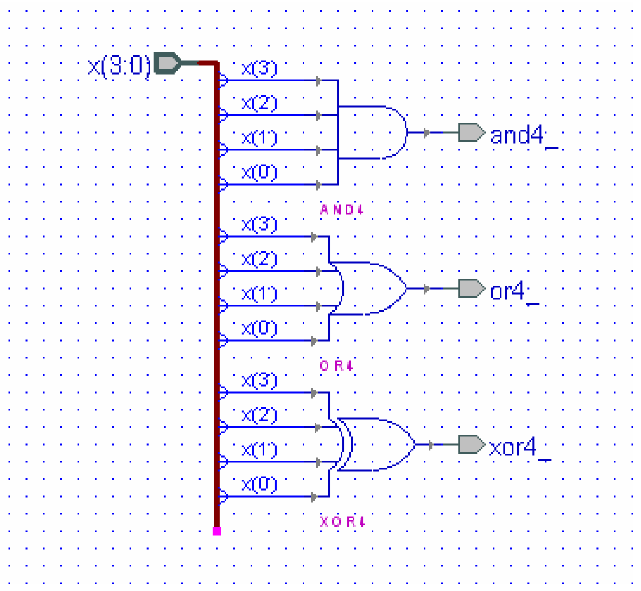


Figure 3.7 Block diagram for *gates4.bde*

If you look at the file *gates4.v* that is generated when you compile *gates4.bde* you will see that Active-HDL defines separate modules for the 4-input AND, OR, and XOR gates and then uses a Verilog instantiation statement to "wire" them together.

Alternatively, we could use the HDE editor to write the simpler Verilog program called *gates4b.v* shown in Listing 3.1 that uses *reduction operators* to implement the three 4-input gates. This Verilog program will produce the same simulation as shown in Fig. 3.8.

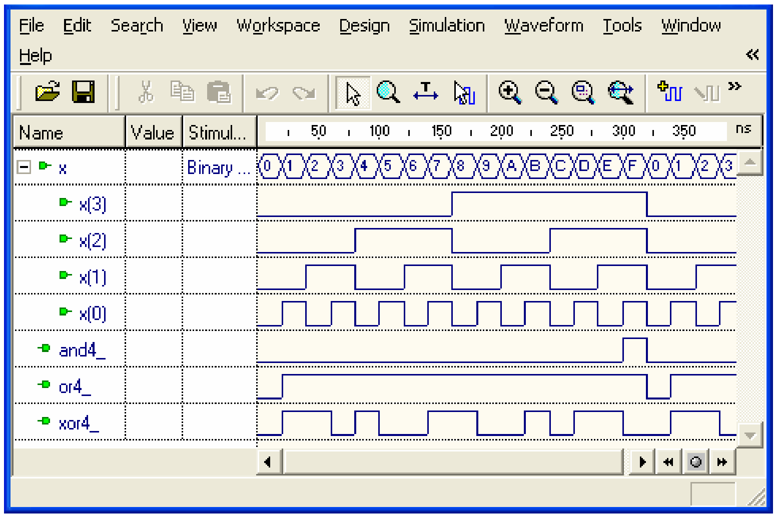

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name
Value
Stimul...
50 100 150 200 250 300 350 ns
x
Binary ...
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3
x(3)
x(2)
x(1)
x(0)
and4_
or4_
xor4_

Figure 3.8 Simulation of the design *gates4.bde* shown in Fig. 3.7

Listing 3.1: gates4b.v

```
// Example 2: 4-input gates
module gates4b (
input wire [3:0] x ,
output wire and4_ ,
output wire or4_ ,
output wire xor4_
);

assign and4_ = &x;
assign or4_ = |x;
assign xor4_ = ^x;

endmodule
```

3.3 Generating the Top-Level Design *gates4_top.bde*

Fig. 3.9 shows the block diagram of the top-level design *gates4_top.bde*. The module *gates4* shown in Fig. 3.9 contains the logic circuit shown in Fig. 3.4. If you compile *gates4_top.bde* the Verilog program *gates4_top.v* shown in Listing 3.2 will be generated. Compile, synthesize, implement, and download this design to the FPGA board.

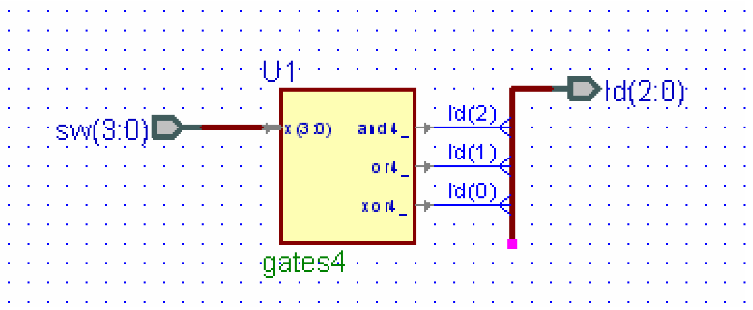


Figure 3.9 Block diagram for the top-level design *gates4_top.bde*

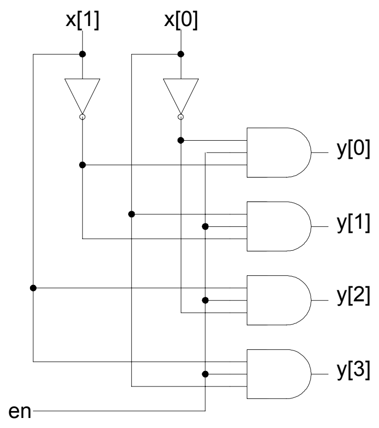
Listing 3.2: gates4_top.v

```
// Example 2: 4-input gates - top level
module gates4_top (
input wire [3:0] sw ,
output wire [2:0] ld
);

gates4 U1
(
      .and4_(ld[2]),
      .or4_(ld[1]),
      .x(sw),
      .xor4_(ld[0])
);

endmodule
```

Problem

3.1 Use the BDE to create a logic circuit containing 4-input NAND, NOR, and XNOR gates. Simulate your design and verify that *the output of an XNOR gate is HIGH only if the number of HIGH inputs is EVEN*. Create a top-level design that connects the four inputs to the rightmost four slide switches and the three outputs to the three rightmost LEDs. Implement your design and download it to the FPGA board.

3.2 The circuit shown at the right is for a 2 x 4 decoder. Use the BDE to create this circuit and simulate it using Active-HDL. Choose a counter stimulator for *x*[1:0] that counts every 20 ns, set *en* to a forced value of 1, and simulate it for 100 ns. Make a truth table with (*x*[1], *x*[0]) as the inputs and *y*[0:3] as the outputs. What is the behavior of this decoder?



In this example we will show how to design a 3-to-8 decoder in Verilog by using a *for* loop. The truth table for a 3-to-8 decoder is shown in Fig. 1.35. Note that the logic equation for each output, y_i, is just the minterm, m_i. You could implement this decoder in Verilog by writing the minterm logic equations or by using a *case* statement to directly implement the truth table in Fig. 1.35.

a2	a1	a0	y0	y1	y2	y3	y4	y5	y6	y7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 1.35 Truth table for a 3-to-8 decoder

Alternatively, the behavior of the 3-to-8 decoder given by the truth table in Fig. 1.35 can be described by the following algorithm:

```
for(i = 0; i <= 7; i = i+1)
  if(a == i)
    y[i] = 1;
  else
    y[i] = 0;
```

We can include these statements in an *always* block of a Verilog program as shown in Listing 1.21. This program will produce the simulation as shown in Fig. 1.36.

Listing 1.21 decode38b.v

```
//  Example 19: 3-to-8 Decoder
module decode38b (
input wire [2:0] a ,
output reg [7:0] y
);
integer i;

always @(*)
  for(i = 0; i <= 7; i = i+1)
    if(a == i)
      y[i] = 1;
    else
      y[i] = 0;
endmodule
```

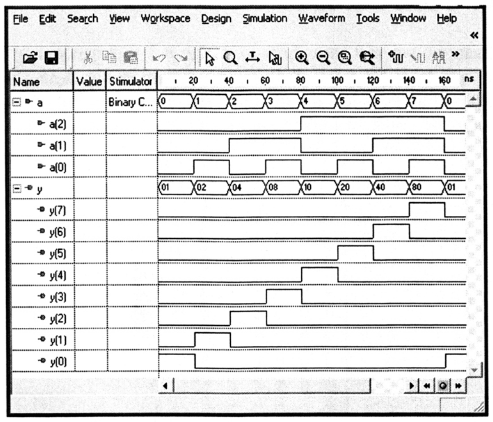


Figure 1.36 Simulation of the Verilog program in Listing 1.21