Four Bit, Odd Parity Generator Utilizing the Sum of Products Concept

Vincent Martin
TUID: 913012274
ECE 2613
Lab #: 2 (9/10/2012)

Introduction:
The objective of this lab is to understand bit parity and its relation to sum of products and then to finally implement it via Verilog code, which will be tested and then placed onto a Nexys2 Xilinx test board.

The Theory of Bit Parity:

If we are utilizing odd bit party then we expect that the bits of data along with the parity bit will add up to be an odd number. Similarly, if we were to implement an even bit parity then we would the sum of all of the bits to be an even number. This is quite useful as it can be used as a rudimentary way to verify communications between two parties or even results within a digital circuit.

Bit parity itself can be calculated using the concept known as the sum of products. The sum of products is defined as multiple AND statements joined with OR statements.

For example if we wanted to transmit 2 data bits (A and B) with odd parity we would need to calculate one parity bit C. This can be represented below via a truth table.

Truth Table for odd parity

A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

We can then use the truth table to aid us in creating a formula, which can represent calculating our odd parity bit.

C = A'B' + A'B + AB'

Applying the Theory to Hardware:

In order to transfer our understanding of theory to our Nexys2 hardware board we will have to write a Verilog code module that represents the block diagram seen in figure 1.

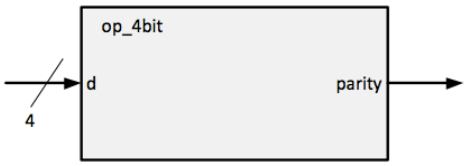

Figure 1: Block diagram for op_4bit module

Additionally we will want to implement a testing scheme based on Figure 2. This scheme will utilize a .txt file, based on our truth table, which will allow us to test all of our expected outcomes. This text file will be included in the lab report.

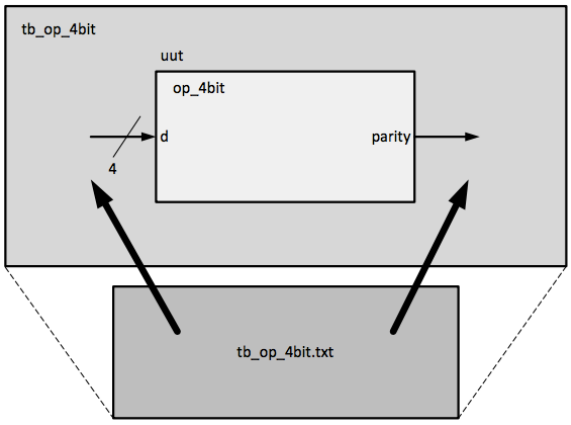

Figure 2: Testing methods

Procedure:

Create the Truth Table and Formula

1. Create a 4-bit truth table (figure 3).
2. Calculate the F parity bit that will make the sum odd.
3. Use the results of the truth table to create a Sum of Products Equation (figure 4).

Implement the Design in Software

1. Make a secure connection to electro9.eng.temple.edu using the no machine client.
2. Once terminal opens on the local workstation type the 'remote_xilinx.sh' shell command to launch the ISE development environment.
3. Open the lab2 bit party project in ~/Xilinx/lab2/ directory.
4. Modify the op_4bit.v source code to implement your algorithm by navigating to
    - View: Implementation
        - xc3s500e-4f6320
            - lab2_top_io_wrapper
5. Save all files.

Prepare for Testing the Design

1. Modify the tb_op_4bit.txt testing to suit the needs of a 4 bit odd parity test by navigating to
    - View: Implementation
2. Modify the tb_op_4bit.txt file to contain all possible input bit combinations.
3. Modify the tb_op_4bit.txt file to contain all expected output bit combinations.

4. Save all files.

Test the Design with iSim

1. Switch to Simulation mode by clicking on
    a. View:Simulation
        i. Xc3s500e-4fg320.
        ii. Tb_op_4bit
2. Run iSim simulator by clicking on
    a. iSim Simulator
    b. Right click Simulate Behavioral Model and then run.
3. Once iSim runs, verify that the Mismatch—index messages match what you are expecting in your test bench text file.
4. If the results are not what you expect either edit your module code or your test bench code and then attempt to test again.
5. If the results are what you expected move on to the Compile to .bit file step.

Compile to .bit file

1. Compile to .bit file by navigating to
    a. Implementation
        i. Xc3s500e-4g320
            1. Lab2_top_io_wrapper
                a. Implement design
                b. Generation programming file

Transfer .bit file to Board

1. Use your favorite network transfer program to move the .bit file from the development server to your local workstation.
2. Plug the board into USB port.
3. Launch the Digilent Adept application on your local workstation.
4. Click the config tab.
5. Click on browse by the PROM icon.
6. Select your transferred .bit file.
7. Click program.
8. Once complete press the reset button on the board.
9. Test your outcome physically on the board to make sure that it matches expectations.

Results:

Below you will find the truth table and equation representing our 4 bit odd parity generator.

Truth Table (figure 3):

d[3]	d[2]	d[1]	d[0]	F Parity bit
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Sum of Products Equation that Satisfies the Truth Table:

F = (d[3]' d[2]' d[1]' d[0]') + (d[3]' d[2]' d[1] d[0]) + (d[3]' d[2] d[1]' d[0]) + (d[3]'d[2] d[1] d[0]') + (d[3] d[2]' d[1]' d[0]) + (d[3] d[2]' d[1] d[0]') + (d[3] d[2] d[1]' d[0]') + (d[3] d[2] d[1] d[0])

Discussion:

This was a good introduction to Verilog for me. It made more clear to me that Verilog is not a programming language but rather a descriptor language; perhaps more along the lines of something like CSS than a procedural language such as C. On top of this it also made me wonder if it was possible to implement some traditional programming techniques along with the Verilog code that may give some more flexibility, such as loops and such to generate the Verilog code. I am sure we will discuss this sort of thing later in the year.

The lab overall was successful for me but success did not come without first making some mistakes. My first mistake was thinking that the parity bit was just something that would tell me if the 4 data bits added up to even or odd rather than actually being included along with the 4 data bits to create an odd summation. Once this mistake was found I had to edit my test bed text file and Verilog module code to fix this problem. Once I changed these values my simulation and physical board worked correctly. These changes can be found in the comments section of my Verilog code.

My second mistake was less about the design and more about my use of the no machine client software. Often times my Internet connection was weak and would kick me off the server, leaving my sessions open. I am going to investigate the best ways to terminate these old abandoned sessions.

In closing, I feel that this was an excellent lab and I learned quite a bit. I hope to continue learning in the future and look forward to this lab class. FPGA design is a topic that interests me greatly as there may be a chance I will make it my career.

Source Code:
Please see attached documents.

Extras:

a. To make alteration to the hardware mappings we would have to modify the 2612_ver3.ucf file. This is where names get mapped to physical hardware parts on our board.

OP_4BIT.V module code

```
//
// lab2 : version 09/04/2012
//
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
////////
////////////////////////////////////////////////////////////////////////////////
////////
module op_4bit(
   input [3:0] d,
   output parity
);

// design parity as sum of products

//assign statement for parity led

/* Old Code that I put in wrong. I think this actually works for even
parity... But I choose to replace it and keep this here for documentation reasons

assign parity = (~d[3] & ~d[2] & ~d[1] & d[0]) |
            (~d[3] & ~d[2] & d[1] & ~d[0]) |
            (~d[3] & d[2] & ~d[1] & ~d[0]) |
            (~d[3] & d[2] & d[1] & d[0]) |
            (d[3] & ~d[2] & ~d[1] & ~d[0]) |
            (d[3] & ~d[2] & d[1] & d[0]) |
            (d[3] & d[2] & ~d[1] & d[0]) |
            (d[3] & d[2] & d[1] & ~d[0]);
*/

// New code for odd parity that works
assign parity = (~d[3] & ~d[2] & ~d[1] & ~d[0])|
            (~d[3] & ~d[2] & d[1] & d[0])|
            (~d[3] & d[2] & ~d[1] & d[0])|
            (~d[3] & d[2] & d[1] & ~d[0])|
            (d[3] & ~d[2] & ~d[1] & d[0])|
            (d[3] & ~d[2] & d[1] & ~d[0])|
            (d[3] & d[2] & ~d[1] & ~d[0])|
            (d[3] & d[2] & d[1] & d[0]);

endmodule
```

tb_op_4bit.txt

```
//
// lab2 : version 9/04/2012 Modified by Vincent Martin for odd parity test
//
// This file contains the test vectors for the
// 4 bit odd parity generator
// The first four columns are the inputs: d[3:0]
// The fourth column is the output: parity
// It needs to be 16 lines long to cover all possibilities
//
// modify your truth table to include the parity bit.
0000_1
0001_0
0010_0
0011_1
0100_0
0101_1
0110_1
0111_0
1000_0
1001_1
1010_1
1011_0
1100_1
1101_0
1110_0
1111_1
```