ECEN 248: Introduction to Digital Design
Department of Electrical and Computer Engineering
Texas A&M University
Lab Report

Laboratory Exercise #5

Introduction to Logic Simulation and Verilog

ECEN-248-509

Joaquin Salas

731000141

10-12-2022

TA: Sri Hari Pada Chandanam Kodi

Objectives:

In previous labs, we have breadboarded, tested, and debugged our digital designs using ICs. This worked well for simple and small designs since nothing is getting too complex, but at the same time, you cannot achieve much with these simple circuits. As the gate counts and circuits increase, breadboarding seems less attractive due to the time and complexity of making the circuit. Now we will be moving into alternatives to bread-boarding circuits. We can simulate the operation of digital circuits during the design process without being distracted by physically breadboarding the circuit. In this lab we will be introduced to Verilog HDL, a standard in Hardware Description Language.

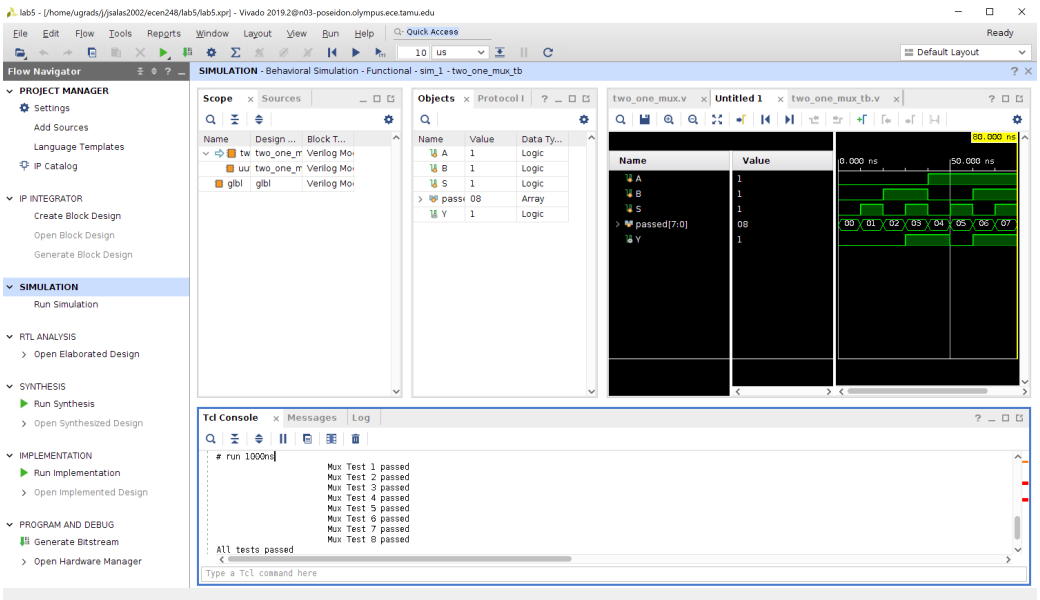
Design:

Experiment Part 1:
The objective of this lab is to be introduced and familiarized with using Vivado. We will be typing in code from the lab manual into Linux and finishing off some of the code on our own. To begin the first part of the experiment, we will start by launching Vivado and creating a new design project. After following the steps listed in the lab manual to create a directory and create a new project folder, we will begin to create a Verilog source file that describes the 2:1 MUX. I added the 2:1 MUX source file and test bench to the Vivado project. After following all the steps for step 3, I proceeded to simulate my file using a test bench. This part successfully tested the 2:1 MUX converter and the waveform panel will be provided down below.

Code 1: *two_one_MUX*

```
`timescale 1ns / 1ps
`default_nettype none


module two_one_mux (Y, A, B, S); //defines the module name and its interface
    //declares output and input ports
    output wire Y;   //output of type wire
    input wire A, B, S;  //declares inputs of type wire

    wire notS;
    wire andA;
    wire andB;


    not not0(notS, S);
    and and0(andA, notS, A);
    and and1(andB, S, B);
    or or0(Y, andA, andB);

endmodule
```

Screenshot 1: *two to one MUX*



Experiment Part 2:

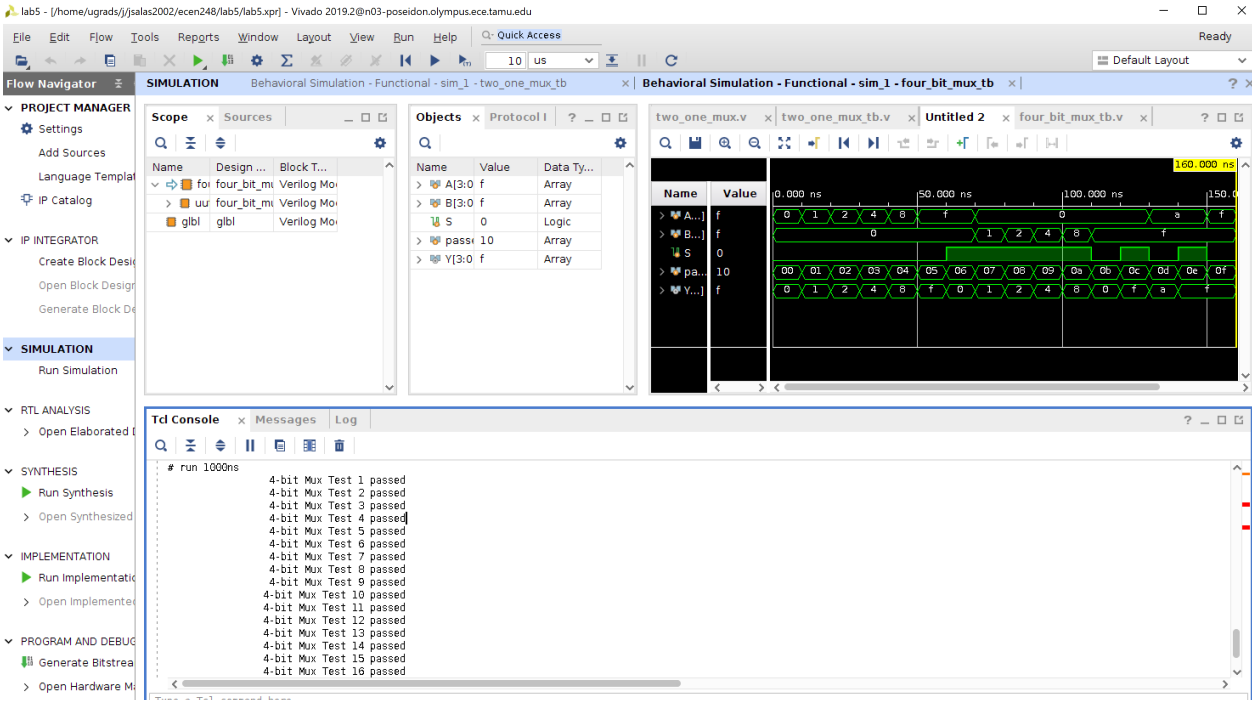
The purpose of this part of the experiment is to design the modules needed to build a simple 4-bit ALU and introduce a slightly higher level of abstraction available in Verilog. Since a single 4-bit is made up of 4 1-bit multiplexers, the following code will describe this.

Code 2: *four_bit_MUX*

```
`timescale 1ns / 1 ps
`default_nettype none

/*This module connects four 1-bit , 2:1 MUXs together to *
*create a single 4-bit, 2:1 MUX */

module four_bit_mux(Y, A, B, S);

    /*declare output and input ports*/
    //output is a 4-bit wide wire
    input wire [3:0] A, B;  //A and B are 4-bit wide wires
    input wire S;  //select is still 1 bit wide
    output wire [3:0] Y;

    /*instantiate user-defined modules*/
    two_one_mux MUX0(Y[0], A[0], B[0], S);
    two_one_mux MUX1(Y[1], A[1], B[1], S);
    two_one_mux MUX2(Y[2], A[2], B[2], S);
    two_one_mux MUX3(Y[3], A[3], B[3], S);

endmodule
```

I then added the file above to my current Vivado project and added a test bench to simulate the logic behavior as in the previous part of this experiment.

Screenshot 2: *four bit MUX*



For the next step in this part, I now have my 4-bit, 2:1 MUX from the 1-bit MUX module and I will be adding an addition/subtraction unit, this is called dataflow and this introduces the **assign** which allows users to describe how data should flow from one wire to the next using arithmetic and logic operators. One thing to keep in mind is the width of the wire. As an example, when describing a bit-wise AND operation for two 4-bit wires, the result wire will also be 4-bits in width. For this part I used the following code:

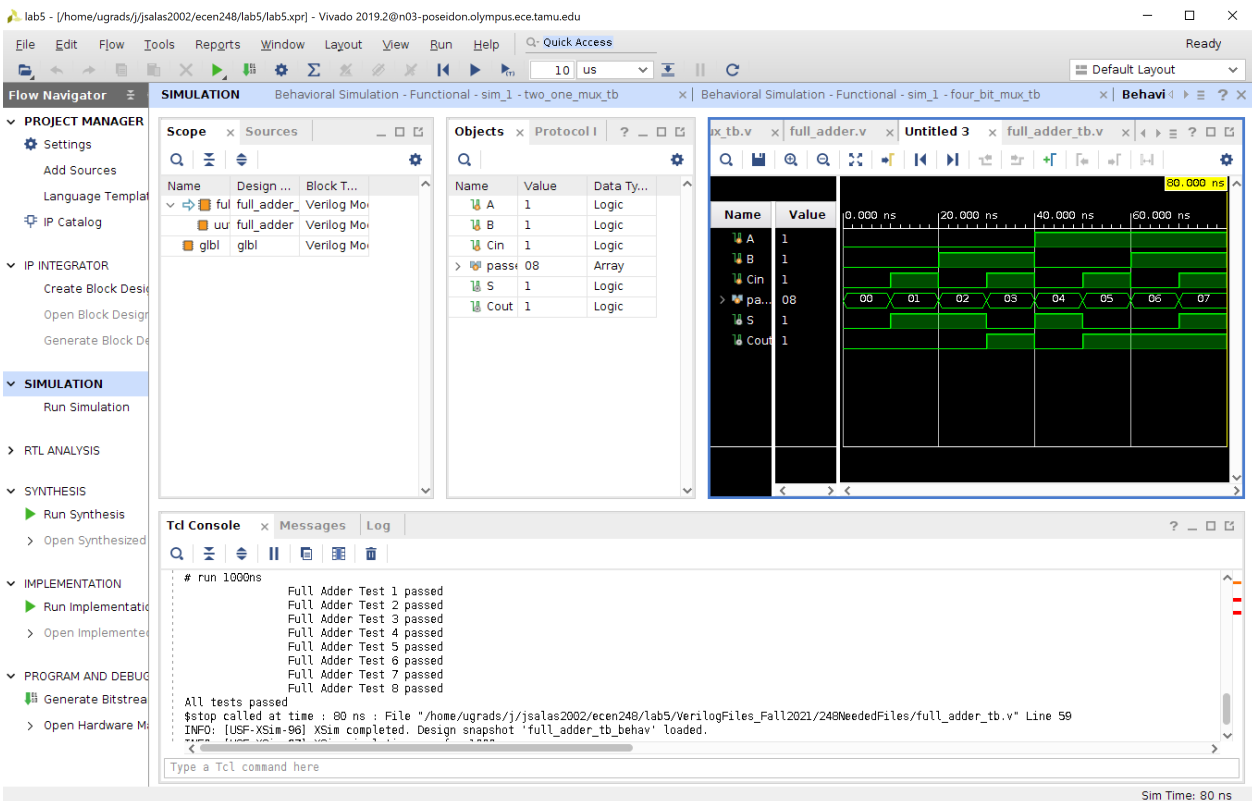
Code 3: *full adder*

```
`timescale 1ns / 1 ps
`default_nettype none
/*This module describes the gate-level model of *
*a full-adder in Verilog  */

module full_adder(S, Cout, A, B, Cin);

    /*declare output and input ports*/
    //1-bit wires
    input wire A, B, Cin;  //1-bit wires
    output wire S, Cout;
```

```
    /*declare internal nets*/
    wire andBCin. andACin, andAB;

    /*use dataflow to describe the gate-level activity */
    assign S = A ^ B ^ Cin; //the hat (^) is for XOR
    assign andAB = A & B;
    assign andBC = B & Cin;
    assign andAC = A & Cin;
    //filled out the code for andBC and andAC
    assign Cout = andAB | andBCin | andACin; //pipe (|) is for or

endmodule
```

After debugging the code, I proceeded to copy over the full adder test bench file and added it to my Vivado project and checked to make sure it worked properly. Attached below is a screenshot of this waveform along with the console input from the test bench.

Screenshot 3: *full adder*



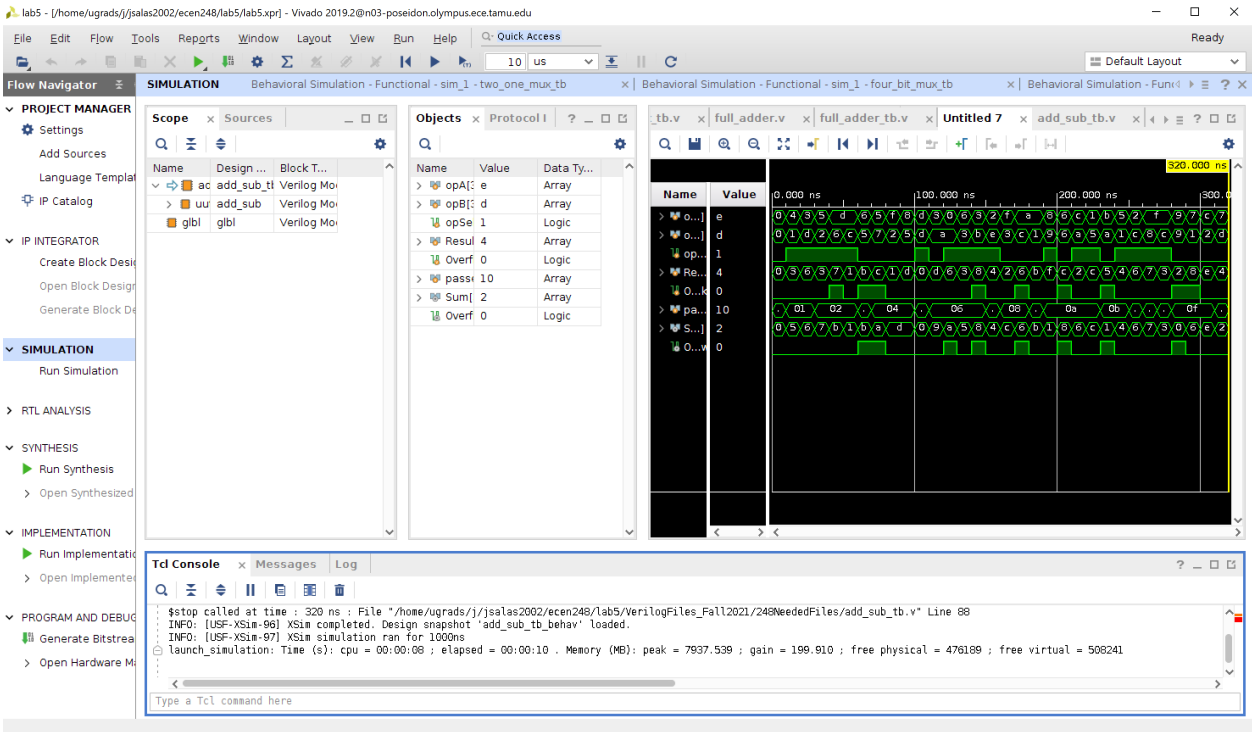
Now to implement two abstraction levels to create addition/subtraction, I started using this code below as a starting point:

Code 4: *adder and subtractor*

```
`timescale 1ns / 1 ps
`default_nettype none
/*This verilog module describes a 4-bit addition/subtraction *
*unit using full-adder modules which have already been *
* designed and tested.  */

module add_sub(
    /*declare output and input ports*/
    output wire [3:0] Sum,  //4-bit result
    output wire Overflow,   //1-bit wire for overflow
    input wire [3:0] opA, opB, //4-bit operands
    input wire opSel    //opSel = 1 for subtract
    ); //in Verilog, we can describe a module interface in this manner as well

    /*declare internal nets */
    wire [3:0] notB;
    wire [3:0] c;

    /*create complement logic */
    assign notB[0] = opB[0] ^ opSel; //if opSel == 1, complement
    assign notB[1] = opB[1] ^ opSel;
    assign notB[2] = opB[2] ^ opSel;
    assign notB[3] = opB[3] ^ opSel;

    /*wire up full adders to create a ripple carry adder*/
    full_adder adder0(Sum[0], c[0], opA[0], notB[0], opSel);
    full_adder adder1(Sum[1], c[1], opA[1], notB[1], opSel);
    full_adder adder2(Sum[2], c[2], opA[2], notB[2], opSel);
    full_adder adder3(Sum[3], c[3], opA[3], notB[3], c[2]);

    /*overflow detection logic*/
    assign Overflow = c[3] ^ c[2]; //overflow is the last two outputs

endmodule
```

After debugging and copying ‘add_sub_tb.v’ from the course directory into the lab 5 directory and simulating the bench. And after the simulation, the waveform and console output of the test bench simulation is attached below.

Screenshot 4: *adder and subtractor*



Experiment Part 3:

For this part of the experiment, I will be creating a simple 4-bit ALU by creating a top-level module with the following interface shown below:

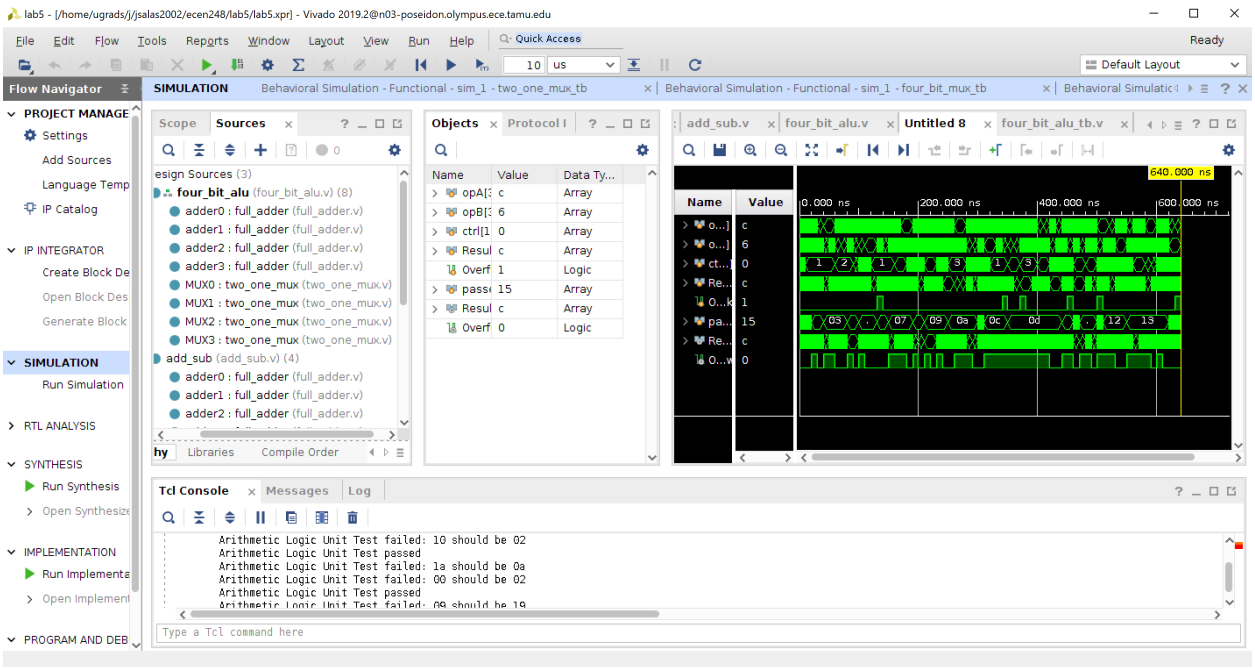
Code 4: *four bit ALU*

```
`timescale 1ns / 1 ps
`default_nettype none
module four_bit_alu(
    output wire [3:0] Result,  //4-bit output
    output wire Overflow, //1-bit signal for overflow
    input wire [3:0] opA, opB, //4-bit operands
    input wire [1:0] ctrl); //2-bit operation select

    wire[3:0] andAB;  //4-bit wire
    wire[3:0] Sum;
    wire[3:0] Cout;

    assign andAB[0] = opA[0] & opB[0]; //assign A and B
    assign andAB[1] = opA[1] & opB[2];
    assign andAB[2] = opA[2] & opB[2];
    assign andAB[3] = opA[3] & opB[3];
```

```
    full_adder adder0(Sum[0], Cout[0], opA[0], opB[0], ctrl[1]); //implement the full
adder
    full_adder adder1(Sum[1], Cout[1], opA[1], opB[1], Cout[0]);
    full_adder adder2(Sum[2], Cout[2], opA[2], opB[2], Cout[1]);
    full_adder adder3(Sum[3], Overflow, opA[3], opB[3], Cout[2]);

    two_one_mux MUX0 (Result[0], andAB[0], Sum[0], ctrl[0]); //implement two to one
MUX
    two_one_mux MUX1 (Result[1], andAB[1], Sum[1], ctrl[0]);
    two_one_mux MUX2 (Result[2], andAB[2], Sum[2], ctrl[0]);
    two_one_mux MUX3 (Result[3], andAB[3], Sum[3], ctrl[0]);

endmodule
```

I then simulated the test bench for the four_bit_alu and added the simulation, waveform, and console output of the test bench simulation is attached below.

Screenshot 5: *four_bit_ALU*



I then simulated the test bench and added the simulation, waveform, and console output of the test bench simulation is attached below.

Results:

My lab resulted in getting 5 screenshots of the waveforms and console outputs for all the circuits we coded. My TA Sri checked this information and gave me the OK.

Conclusion:

In this lab I was introduced to learning how to use Linux without crashing on my computer, although it took quite a while. I also learned to begin coding in Verilog and running simulations for my code using the test bench file.

Post-lab Deliverables:

1. Include the source code with comments for **all** modules you simulated. You do **not** have to include test bench code. Code without comments will not be accepted!

   All source code is listed above and labeled in the experiment parts.

2. Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation. Please ensure these are legible in your report. Waveforms need to be properly fit.

   All screenshots are shown above and labeled in the experiment parts

3. Examine the 1-bit, 2:1 MUX test bench code. Attempt to understand what is going on in the code. The test bench is written using behavioral Verilog, which reads much more like a programming language. Explain briefly what it is the test bench is doing.

   The 2:1 MUX test bench code is inserting a set of Boolean numbers and running them through my code. The test bench knows what the right and wrong answers are for all the tests so that is how you know if your code is right or not, if you pass all tests.

4. Examine the 4-bit, 2:1 MUX test bench code. Are all of the possible input cases being tested? Why or why not?

   Yes, all inputs are tested because there are 3 inputs so the truth table will be 16 rows long.

5. In this lab, we approached circuit design in a different way compared to previous labs. Compare and contrast breadboarding techniques with circuit simulation. Discuss the advantages and disadvantages of both. Which do you prefer? Similarly, provide some insight as to why HDLs might be preferred over schematics for circuit representation. Are there any disadvantages to describing a circuit using an HDL compared to a schematic? Again, which would you prefer?

   I personally prefer the breadboarding compared to HDL, since I am a more hands on person, visualizing the circuit in front of me helps me understand what is happening. HDL are definitely preferred because you can create larger circuits in code and not have to deal with a large amount of wires.

6. Two different levels of abstraction were introduced in this lab, namely structural and dataflow. Provide a comparison of these approaches. When might you use one over the other?

   I would most likely use structural more because it is easier for me to understand when I compare it to a circuit and code. If I know what the circuit looks like I can easily code It in structural.

Important Student Feedback

1. What did you like most about the lab assignment and why? What did you like least about it and why?

I believe this lab was simple and straightforward. The way the lab is set up in a row with the TA in front is a bad system in my opinion. The people further back in the row are not able to hear the TA speaking at the front of the row.

2. Were there any sections of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?

All sections on the lab manual were clear, if there was anything I was confused about it was because of my under-preparedness for the lab. I did not read over the lab the night prior, so I was confused at first but with help of classmates and the TA, I was luckily able to figure it out well.

3. What suggestions do you have to improve the overall lab assignment?

I have noticed a lot of the smaller components used to perform the lab are broken or defective, and it makes building the circuit a hassle since the students don’t know if the wiring is at fault or the components are.