# Lab 5: Introduction to Logic Simulation and Verilog

ECEN 248 - 505

TA: Younggyun Cho

Date: October 6, 2020

Objectives

The objective of this lab was to gain an understanding of Verilog by describing the circuits using code that we previously used breadboards to execute. This lab required the student to learn the syntax of Verilog language and become familiar with the software itself, and describe basic circuits and point out errors made in the pre-written code.

Design

The code below are the 5 different programs used throughout the lab with proper comments explaining each step. Using the prewritten code, the logic used below helped execute the waveform diagrams, which describes each circuit.

**TwoOne Mux**

```
`timescale 1 ns/ 1 ps
module two_one_mux(Y, A, B, S);
        output wire Y;                  //variable declaration
        input wire A, B, S;
        wire notS;                                      //declaration of wires
        wire andA;
        wire andB;
        not not0(notS, S);              //declaration of  logic gates and what variables used
        and and0(andA, notS,A);
        and and1(andB, S, B);
        or or0(Y, andA, andB);
endmodule               //end of module
```

**FourBit Mux**

```
`timescale 1 ns/ 1 ps
`default_nettype none
//connecting 4 1-bit, 2:1 MUXs to make a 4-bit 2:1 MUX
module four_bit_mux (Y, A, B, S);
        input wire [3:0] A, B, Y;       //A & B represent the 4-bit wide wires
        input wire S;                   //select
        //using the two_one_mux module we provide intanced of  of 4-bits
        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   //end of module
```

**Full Adder**

```
`timescale 1 ns/ 1 ps
`default_nettype none
module full_adder(S, Cout, A, B, Cin);
        input wire A, B, Cin;                   //declaring input & output variables
        output wire S, Cout;
        wire andBCin, andACin, andAB;           //declaring 1-bit wires
        assign S = A ^ B ^ Cin;                         //XOR
        assign andAB = A & B;                           //AND
        assign andBCin = B & Cin;                       //AND for BCin
```

```
        assign andACin = A & Cin;                   //AND for ACin
        assign Cout = andAB | andBCin | andACin;    //OR all AND functions together together
    endmodule // end of module
```

**AddSub**

```
`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(Sum, Overflow, opA, opB, opSel);
        /*declare output and input ports */
        output wire [3:0] Sum; //4-bit sums
        output wire Overflow; //1-bit wire for overflow
        input wire [3:0] opA, opB; //4-bit operands
        input wire opSel; //opSel = 1 for subtract
        /* declare internal nets */
        wire [3:0] notB;
        wire c0, c1, c2, c3;
        /* create complement logic */
        assign notB[0] = opB[0] ^ opSel;
        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], c0, opA[0], notB[0], opSel);
        full_adder adder1(Sum[1], c1, opA[1], notB[1], c0);
        full_adder adder2(Sum[2], c2, opA[2], notB[2], c1);
        full_adder adder3(Sum[3], c3, opA[3], notB[3], c2);
        /* overflow detection logic */
        assign Overflow = c2 ^ c3;
endmodule //end of module
```

**FourBit alu**

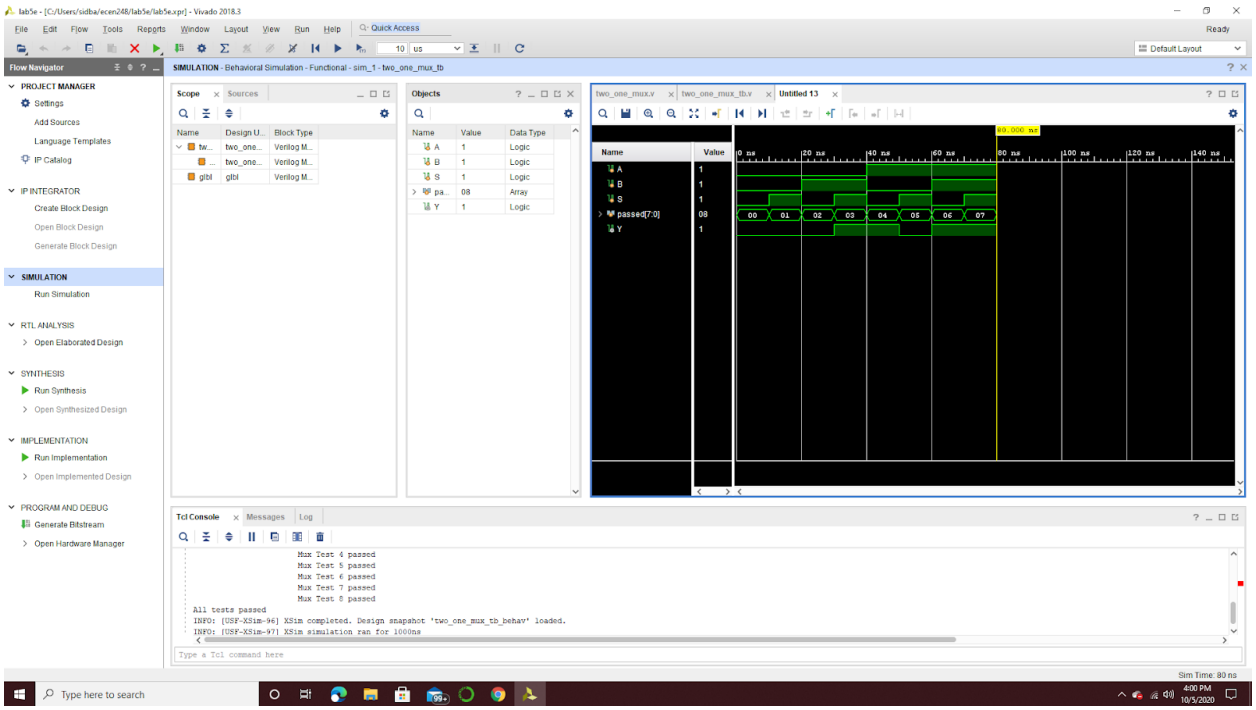
```
`timescale 1 ns/ 1 ps
`default_nettype none
module four_bit_alu(            //variable declaration
        output wire [3:0] Result,
        output wire Overflow,
        input wire [3:0] opA, opB,
        /*crtl | operation*
        * 00   | AND     *
        * 01   | ADD     *
        * 10   | AND      *
        * 11   | SUB      */
        input wire [1:0] ctrl
        );
        wire [3:0] tsum, tand;          //declaring temporary sum and AND blocks
        wire tempover;                  //temporary overflow
        add_sub addsub(tsum, tempover, opA, opB, ctrl[1]);    //calling the add_sub module
```

```
    assign tand = opA&opB;  //definition of a temporary AND
    assign Result[0] = (tsum[0] & ctrl[0])|(~ctrl[0] &tand[0]); //assign answers different scenarios
    assign Result[1] = (tsum[1] & ctrl[0])|(~ctrl[0] &tand[1]); //uses the sum,control,and AND results
    assign Result[2] = (tsum[2] & ctrl[0])|(~ctrl[0] &tand[2]);
    assign Result[3] = (tsum[3] & ctrl[0])|(~ctrl[0] &tand[3]);
    assign Overflow = tempover & ctrl[0];
endmodule       //end of module
```

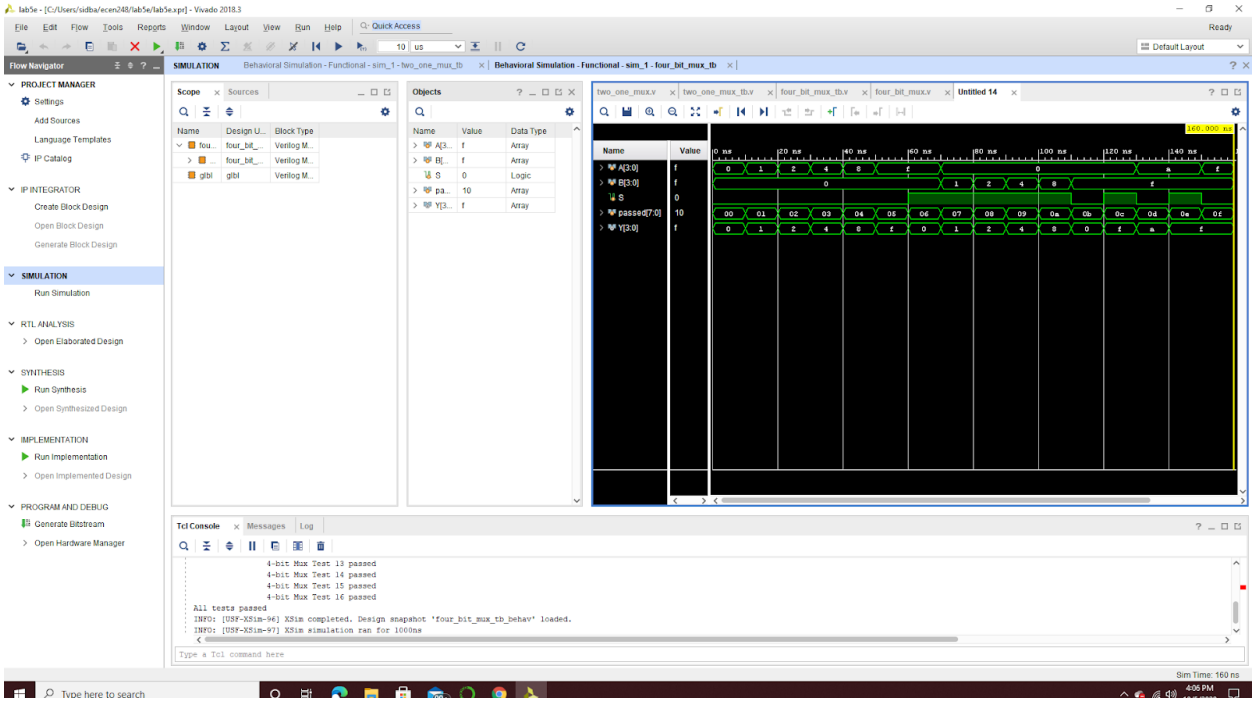
## Results

The results from every source code is shown below. These waveforms are tested by the "Test bench" code and the source code in order to produce a visual representation of the timing for the circuit.

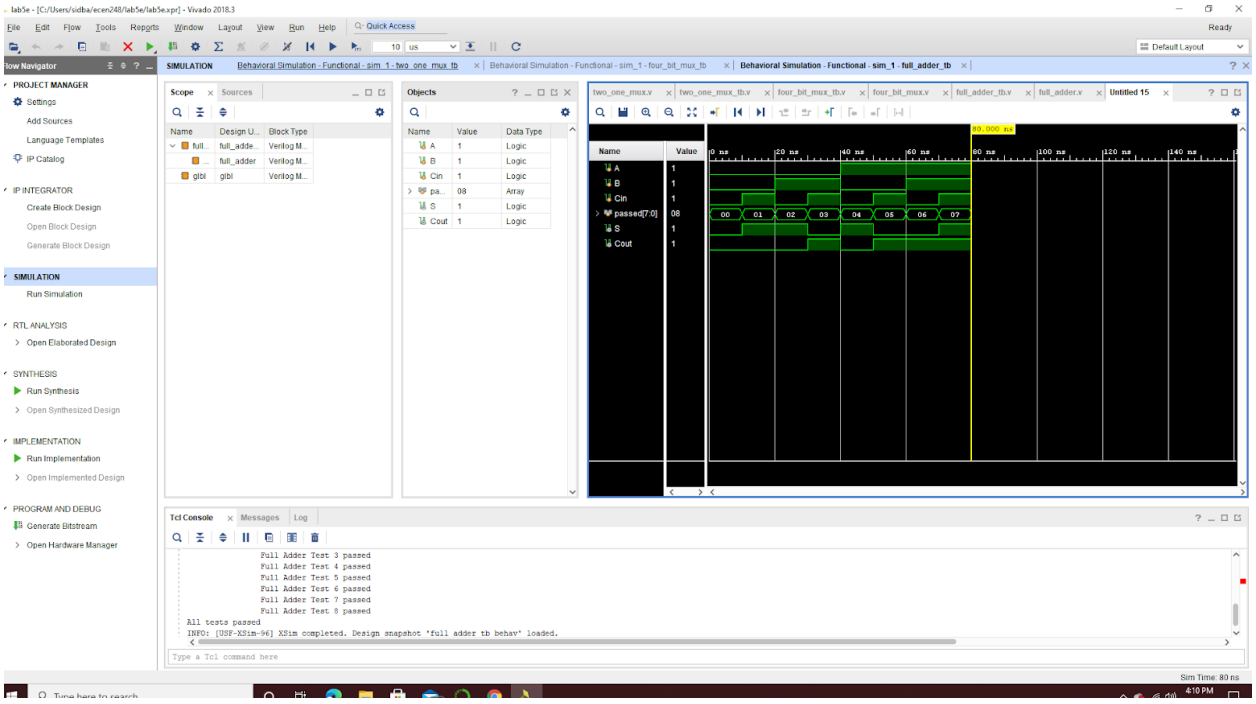
## TwoOne Mux



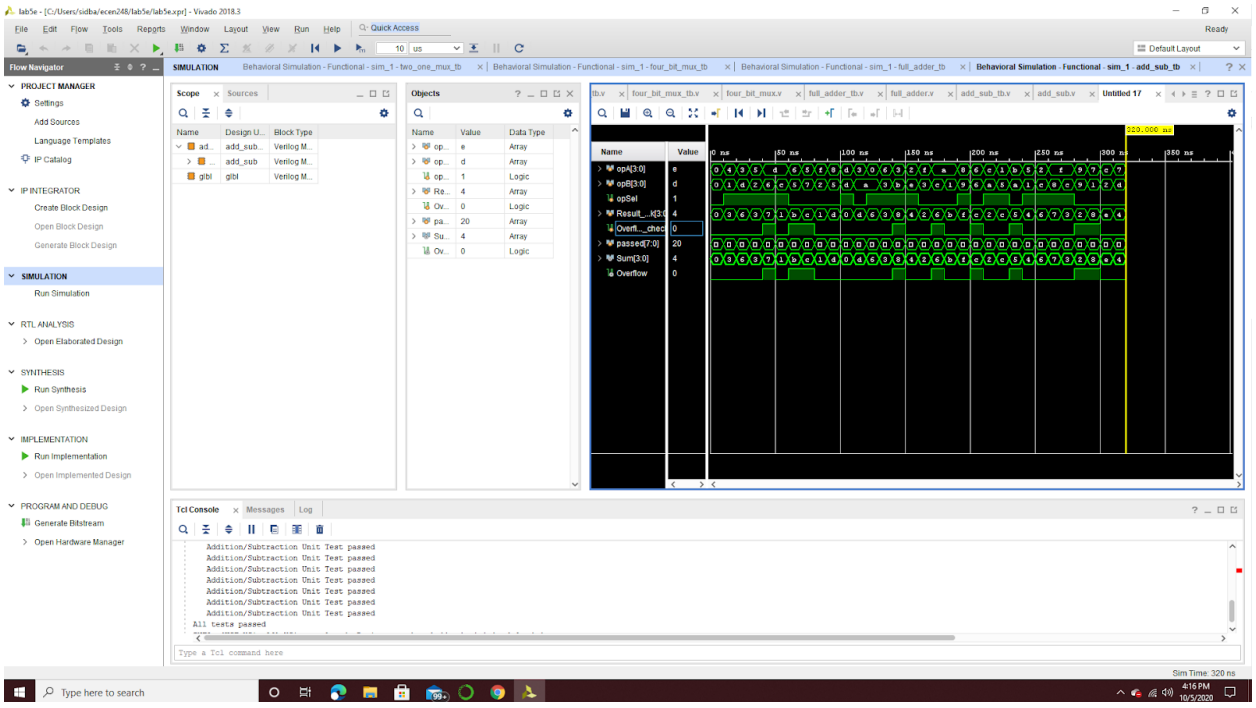

## FourBit Mux



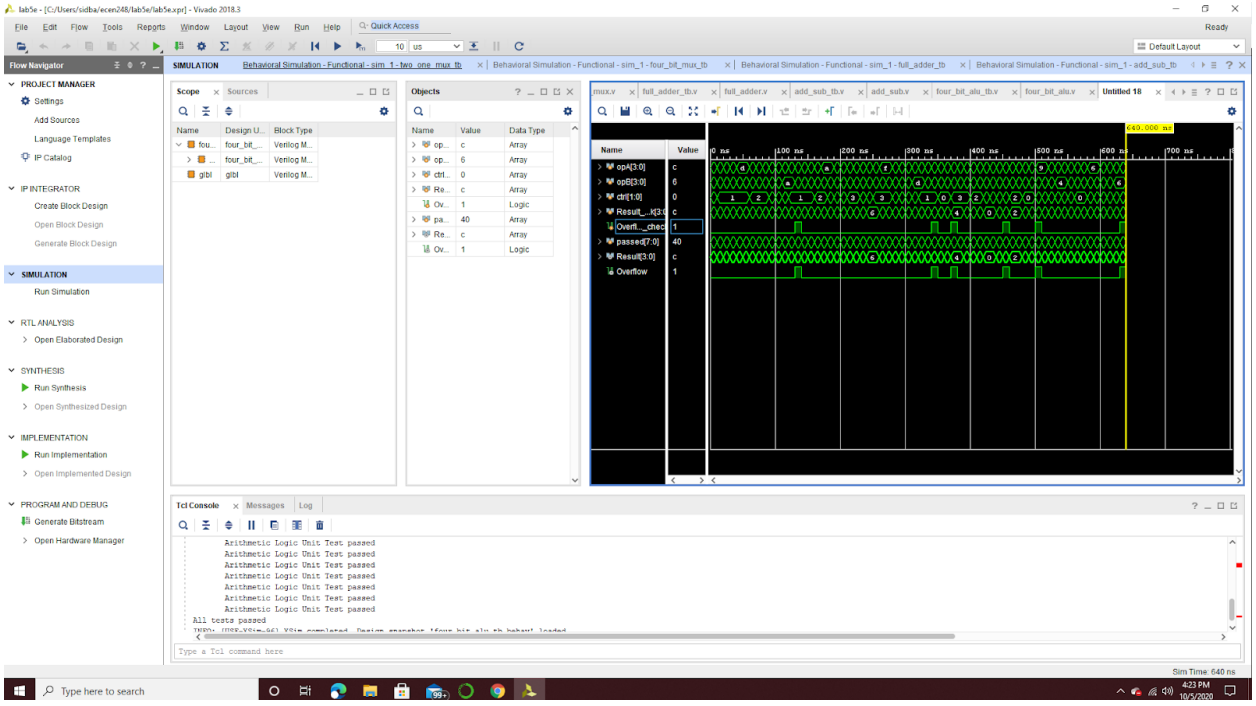

## Full Adder




## AddSub




## FourBit alu




## Conclusions

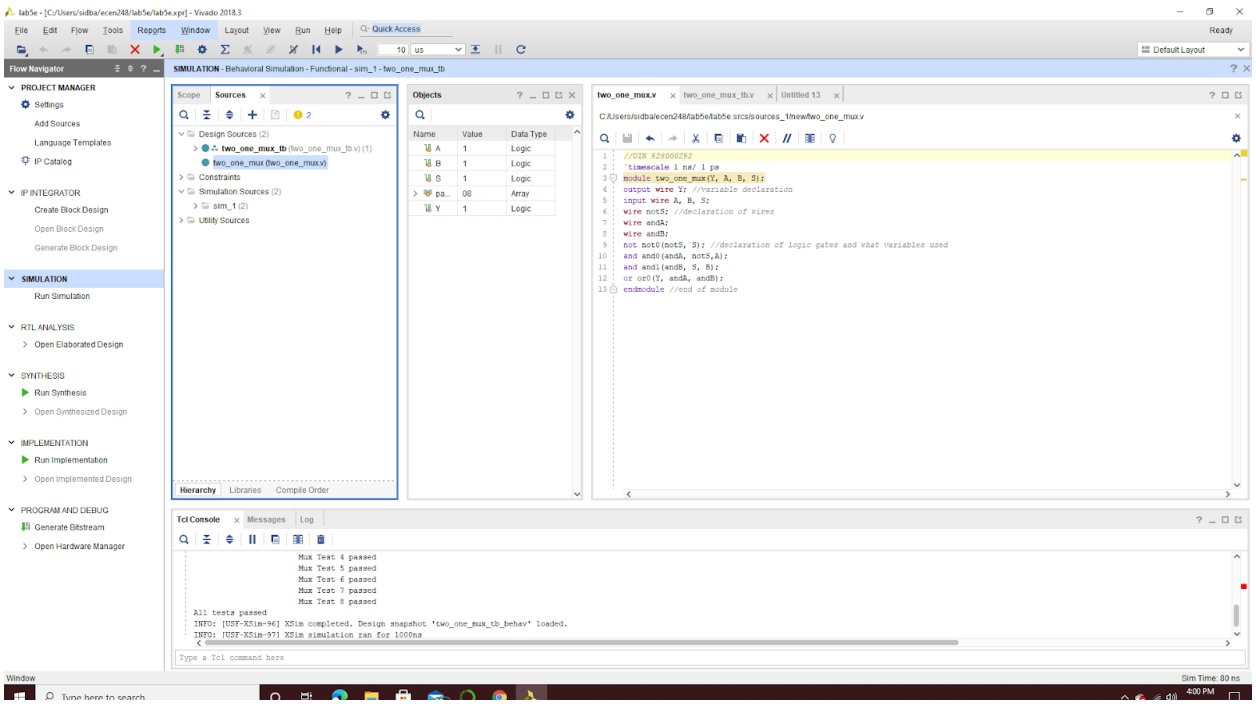
This lab familiarized me with the language verilog and while it took a few trials, I successfully got the source code to run, producing the output it should. The source codes were linked together, as they

called previous modules similar to the usage of a block diagram in regular circuit designs. So this lab was built on top of each other.

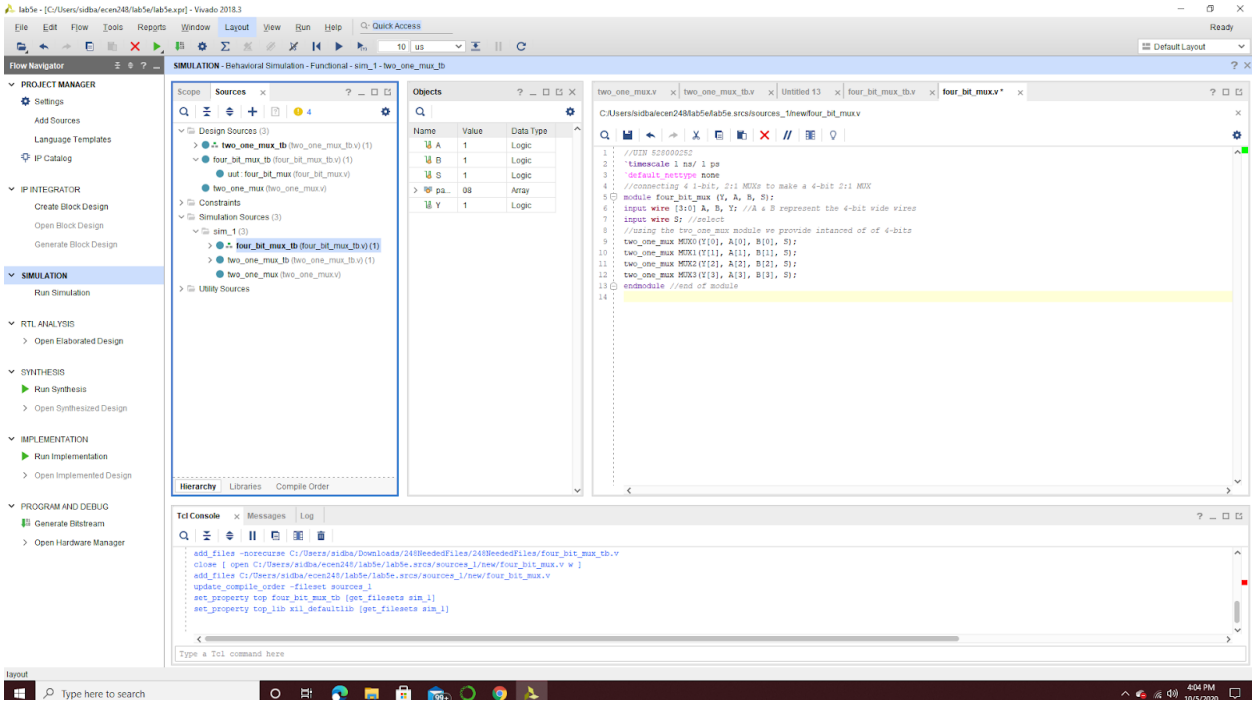
Post-Lab Questions

**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! If the last 4 digits of your UIN cannot be found in your verilog code, you will not receive any point for that design.**

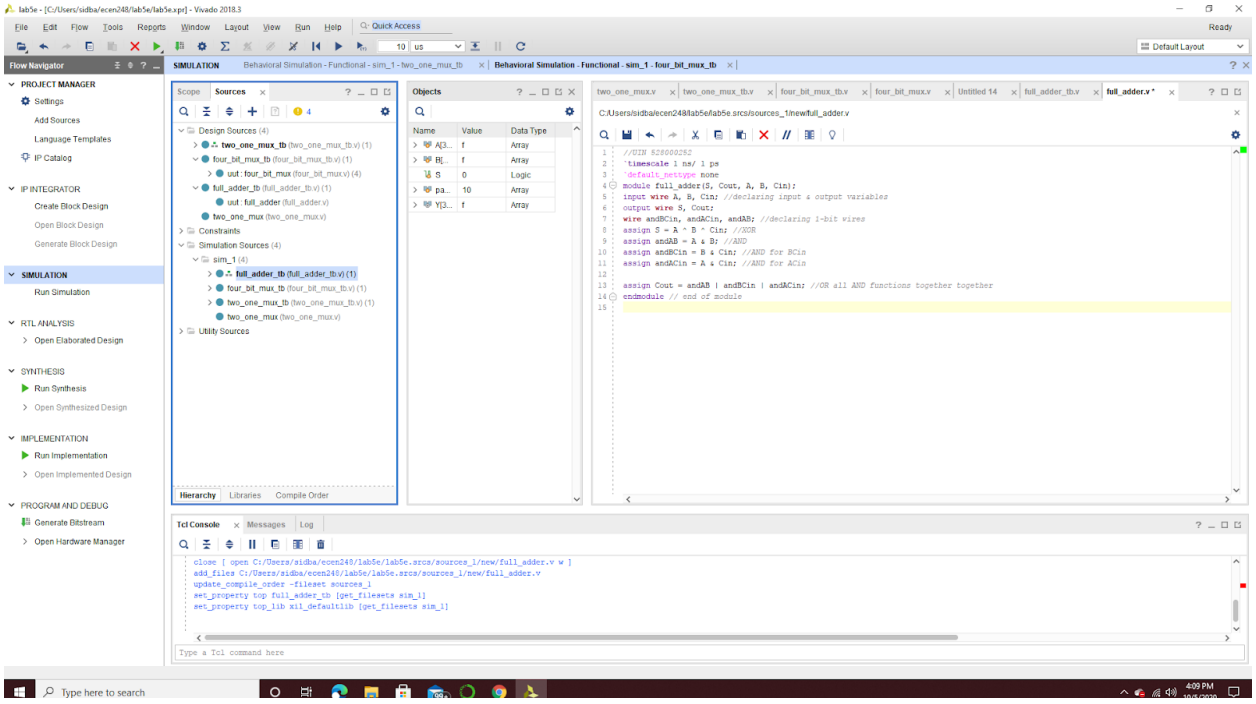
## TwoOne Mux



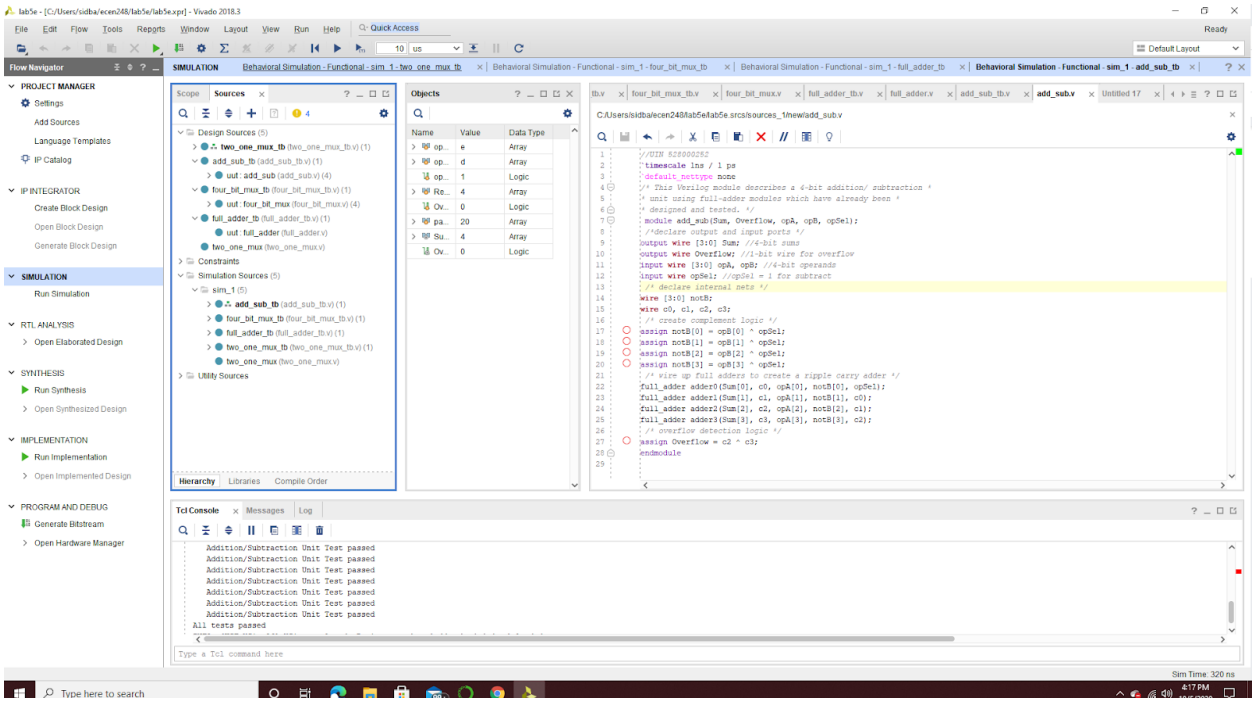

## FourBit Mux



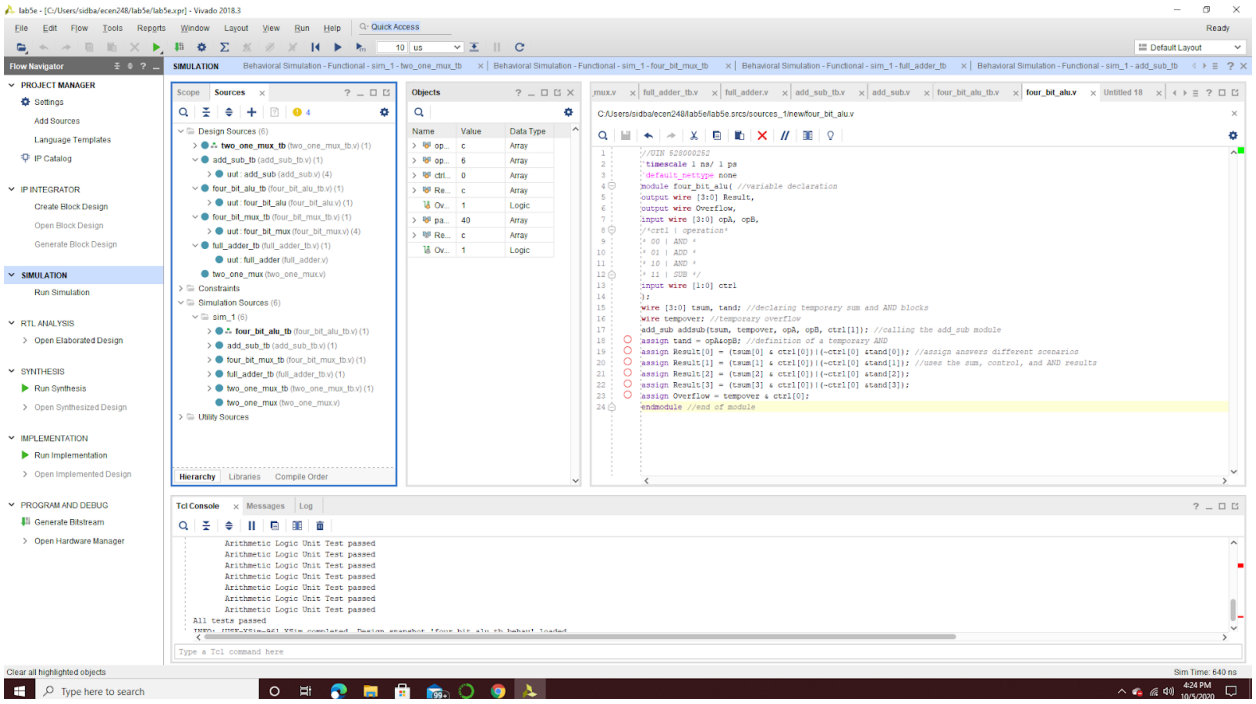

# Full Adder




# AddSub




## FourBit alu




**2. Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.**

In the results section.

**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 will read much like a programming language. Explain briefly what it is the test bench is doing.**

From the two_one_mux_tb test bench source code I understood that the code puts my code through multiple test cases to make sure that the circuit follows the right logic and rules. A test failing could mean that the circuit coded could have a missing element, or incorrect logic, and could require some debugging. This code takes inputs and outputs and tests it against predetermined values of inputs and outputs, which is set by a truth table.

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

No, not all possible inputs are being tested because that means that there are hundreds of outputs since A and B are both four bit wires and every set would need to be tested. However, it isn't necessary to test every input because it takes 16 sets of inputs to show that the circuit's pathways work correctly.

**5. In this lab, we approached circuit design in a different way compared to previous labs. Compare and contrast bread-boarding 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.**

<u>Breadboarding vs simulation</u>

Both techniques have their own disadvantages and advantages. Bread-boarding gives a more hands on feeling, which makes it easier to visualize every connection being made, whereas in the HDL, there could be a syntax error, and the circuit will fail to run. However, an HDL is much easier to debug a more complex circuit because you can check line by line to detect any errors, however on a breadboard it all builds up on each other making it harder to pinpoint where the issue originated from. For example, an ALU is a very complex circuit that builds on itself and can be very messy to look at on a breadboard. But it was easy to develop on the HDL because the circuit isn't complex to understand but tedious to build on a breadboard. I personally prefer to use the Verilog HDL as it isn't chaotic to look at for more complex circuits and makes debugging much easier.

<u>HDL vs Schematics</u>

Similarly there are benefits to both HDL and schematics. Using a schematic can help with physically visualizing the flow of the circuit and helps with the understanding of the circuit at the gate level, but it can be very confusing to follow if it's a complex circuit. However, an HDL design doesn't have a physical representation but it is simple and concise, so it makes it very easy to understand the process and goal of the circuit. I prefer using the HDL for complex circuits because the organization of it is much simpler to comprehend.

**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?**

For the 2:1 and 4:1 MUX source codes the structural approach best represents it and the dataflow approach represents the full adder source code. The data flow is based on a function over structural elements, and it utilizes operands to describe a function of a circuit in place of logic gates. I prefer using the structural method over data flow when the schematic is given and only requires us to describe it in the verilog, stating the gates and inputs and outputs. I would use dataflow when the circuit is described by Boolean algebra or truth tables and have to determine the inputs using boolean operators.

Feedback

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

In this lab I liked that the instructions were very clear and easy to follow. Especially for those who don't know how to use verilog it made it very easy to work through the source code and get the desired results. The first part walked the student through it in detail and then the rest of the parts slowly got less detailed,which tested if we knew what to do but took a lot more time, though overall I didn't have a least favorite part, because every part just repeats itself.

**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?**

This lab manual was very clear and informative for the students to follow.

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

I wouldn't change anything about this lab assignment.