ECEN 248 - Lab Report

Lab Number: 5

Lab Title: Introduction to Logic Simulation and Verilog

Section Number: 513

Student's Name: Abhishek Bhattacharyya

Student's UIN: 731002289

Date: 3/9/2023

TA: Hesam Mazaheri

Objectives:

The objective of this lab is to introduce the fundamental concepts of digital logic design, simulation, and implementation using the Verilog hardware description language (HDL). In this lab, starter code is provided for a two-one mux, four-bit two-one mux, full adder, and add/sub unit. During this lab, this starter code was completed and tested using provided testbench files. The last step of this lab is combining each of these modules as well as dataflow logic to code a 4-bit ALU from the minimal skeleton code which is provided.

The main end goal here is to gain hands-on experience with designing and simulating simple digital circuits in Verilog, by translating what was done in Lab 4 (constructing a 4-bit ALU on a breadboard) into Verilog HDL. This lab helped teach the fundamentals of how to use Verilog to describe digital circuits, as well as how to use the simulator to test each module as well as the full circuit. By the end of the lab, a solid understanding of the basics of Verilog was gained, which will be expanded upon in Lab 6 and all subsequent labs this semester in ECEN-248.

Design:

Following the Lab 5 manual, a new project was created in Vivado for Lab 5 and each module used to build the 4-bit ALU, as well as the 4-bit ALU itself, were defined in Verilog HDL code. The final code is included below in Figure 1 - Figure 5.

```
1  `timescale 1ns / 1ps
2  `default_nettype none
3  /*This module describes a 1-bit wide multiplexer using structural constructs
4  and gate-level primitives built into Verilog*/
5
6  module two_one_mux (Y, A, B, S);
7
8      // declare output and input ports
9      output wire Y;
10     input wire A, B, S;
11
12     // declare internal nets
13     wire notS; // inverse of S
14     wire andA; // output of and0 gate
15     wire andB; // output of and1 gate
16
17     // instantiate gate-level modules
18     not not0(notS, S);
19     and and0(andA, notS, A);
20     and and1(andB, S, B);
21     or or0(Y, andA, andB);
22
23 endmodule // designate end of module
```

Figure 1: Two-One Mux Code

```
1   `timescale 1ns / 1ps
2   `default_nettype none
3   /* This module connects four 1-bit, 2:1 MUXs together to
4   create a single 4-bit, 2:1 MUX*/
5
6   module four_bit_mux (Y, A, B, S);
7
8       // declare output and input ports
9       // output is a 4-bit wide witre
10      input wire [3:0] A, B;  //A and B are 4-bit wide wires
11      input wire S;           // select is still 1-bit wide
12      output wire [3:0] Y;
13
14      // instantiate user-defined modules
15      two_one_mux MUX0(Y[0], A[0], B[0], S);
16      two_one_mux MUX1(Y[1], A[1], B[1], S);
17      // you need two more module instantiations here...
18      two_one_mux MUX2(Y[2], A[2], B[2], S);
19      two_one_mux MUX3(Y[3], A[3], B[3], S);
20
21  endmodule
```

Figure 2: *Four-Bit Two-One Mux Code*

```
1   `timescale 1ns / 1ps
2   `default_nettype none
3   /* This module describes the gate-level model of
4   a full-adder in Verilog*/
5
6   module full_adder (S, Cout, A, B, Cin);
7
8       // declare intput and output ports
9       //1-bit wires
10      input wire A, B, Cin; //1-bit wires
11      output wire S, Cout;
12
13      // declare internal nets
14      wire andBCin, andACin, andAB; // 1-bit wires (missing something???)
15
16      // use dataflow to describe gate-level activity
17      assign S = A ^ B ^ Cin; // the hat (^) is for XOR
18      assign andAB = A & B;
19      // filled in code for andBC, andAC
20      assign andBCin = B & Cin;
21      assign andACin = A & Cin;
22
23      // this line is missing something
24      assign Cout = andAB | andBCin | andACin;  // pipe (|) is for OR operation
25
26  endmodule
```

Figure 3: *Full Adder Code*

```
1   `timescale 1ns / 1ps
2   `default_nettype none
3   /* This Verilog module describes a 4-bit addition/substraction
4   unit using full-adder modules which have already been
5   designed and tested/ */
6
7   module add_sub(
8       // declare output and input ports
9       output wire [3:0] Sum,  // 4-bit result
10      output wire Overflow,  // 1-bit wire for overflow
11      input wire [3:0] opA, opB,  // 4-bit operands
12      input wire opSel);           // opSel = 1 for subtract
13      // in Verilog, we can desgibe a module itnerface in this manner as well!
14
15      //declare internal nets
16      wire [3:0] notB;
17      wire c0, c1, c2, c3;
18
19      // create complement logic
20      assign notB[0] = opB[0] ^ opSel;    // if opSel ==1, complement
21      assign notB[1] = opB[1] ^ opSel;    // if opSel ==1, complement
22      assign notB[2] = opB[2] ^ opSel;    // if opSel ==1, complement
23      assign notB[3] = opB[3] ^ opSel;    // if opSel ==1, complement
24
25
26
27      // wire up full adders to create a ripple carry adder
28      full_adder adder0(Sum[0], c0, opA[0], notB[0], opSel);
29      full_adder adder1(Sum[1], c1, opA[1], notB[1], c0);
30      full_adder adder2(Sum[2], c2, opA[2], notB[2], c1);
31      full_adder adder3(Sum[3], c3, opA[3], notB[3], c2);
32
33      //overflow detecton logic
34      assign Overflow = c2 ^ c3;
35
36  endmodule
```

Figure 4: *Add_Sub Unit Code*

```
1   `timescale 1ns / 1ps
2   `default_nettype none
3
4   module four_bit_alu(
5       // declare input and output ports
6       output wire [3:0] Result,
7       output wire Overflow,
8       input wire [3:0] opA, opB,
9       input wire [1:0] ctrl
10  );
11      // declare internal nets
12      wire[3:0] AddSub_to_Mux, And_to_Mux;
13      wire Out_to_Overflow;
14
15      // use dataflow to describe gate-level activity for AND portion of ALU
16      assign And_to_Mux[0] = opA[0] & opB[0];
17      assign And_to_Mux[1] = opA[1] & opB[1];
18      assign And_to_Mux[2] = opA[2] & opB[2];
19      assign And_to_Mux[3] = opA[3] & opB[3];
20
21      // Use the premade add_sub and four-bit MUX to complete the ALU
22      // Wires are connected according to the internal nets for the ALU
23      add_sub adder_sub0(AddSub_to_Mux, Out_to_Overflow, opA, opB, ctrl[1]);
24      four_bit_mux mux_unit0(Result, AddSub_to_Mux, And_to_Mux, ctrl[0]);
25
26      // Use dataflow to program the overflow detection logic
27      assign Overflow = Out_to_Overflow & ctrl[0];
28
29  endmodule
```

Figure 5: *4-Bit ALU Code*

After each module was built and completed using the starter code, each module was tested using Vivaldo's simulation feature using its respective testbench (.tb) file. All modules passed testing, and the results of each testbench simulation can be found in the next section, **Results**.

Results:

Part 1:

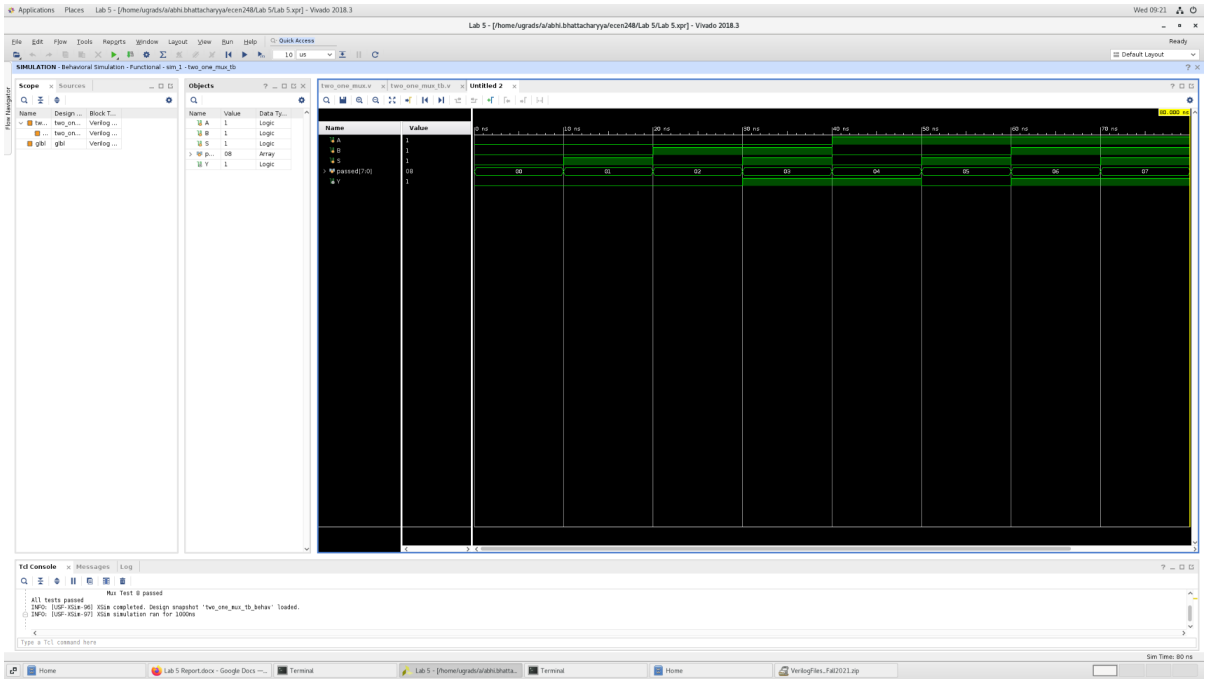

Figure 6: Simulating the 2:1 MUX using the Test Bench

Part 2:

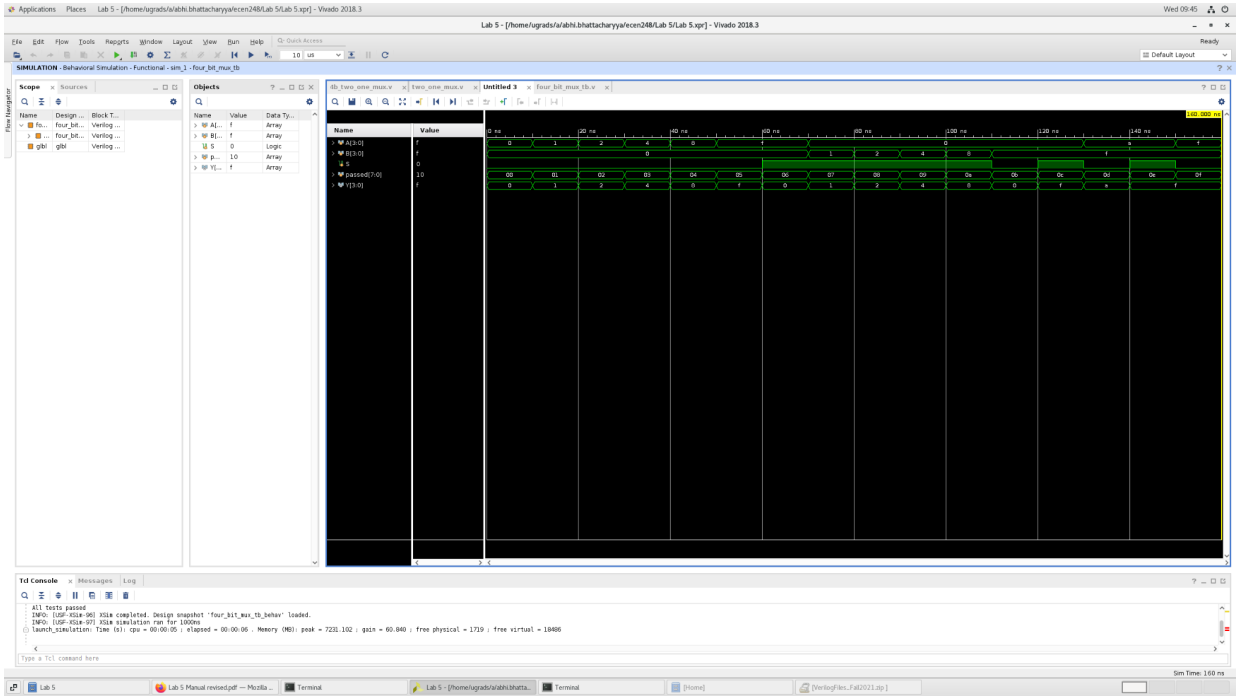

Figure 7: Simulating the Four-Bit Mux using the Test Bench

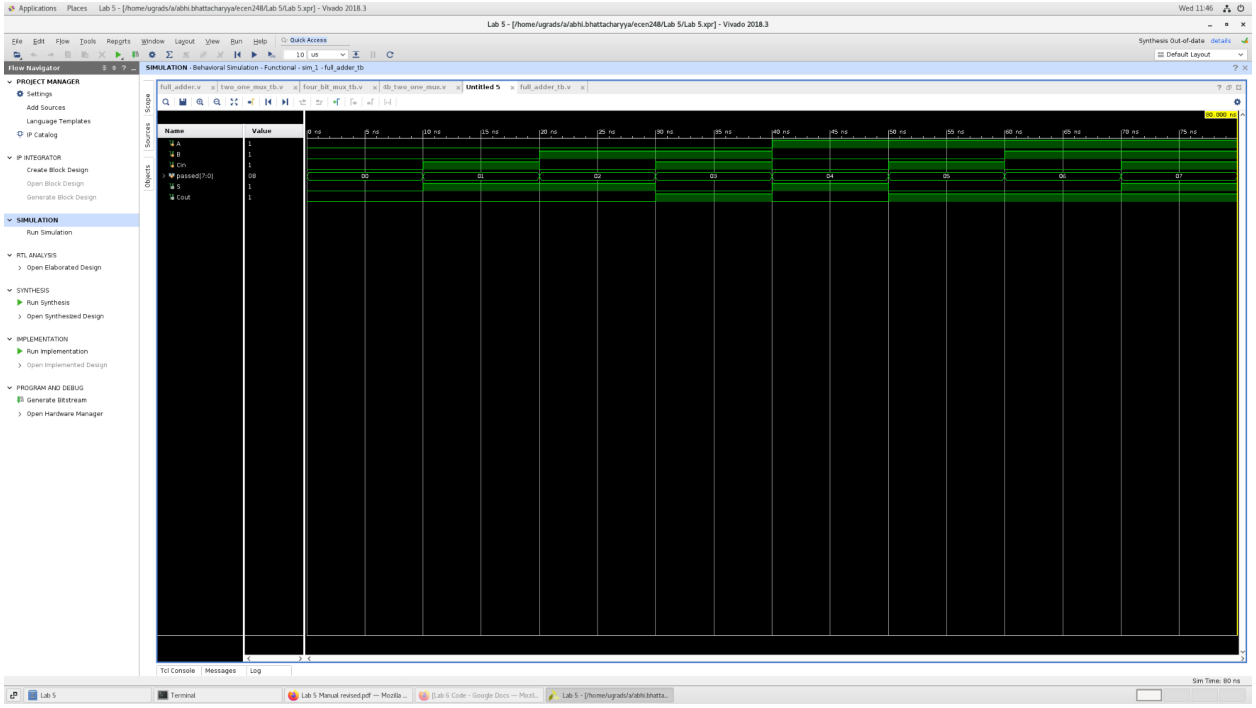

Figure 8: Simulating the Full-Adder using the Test Bench

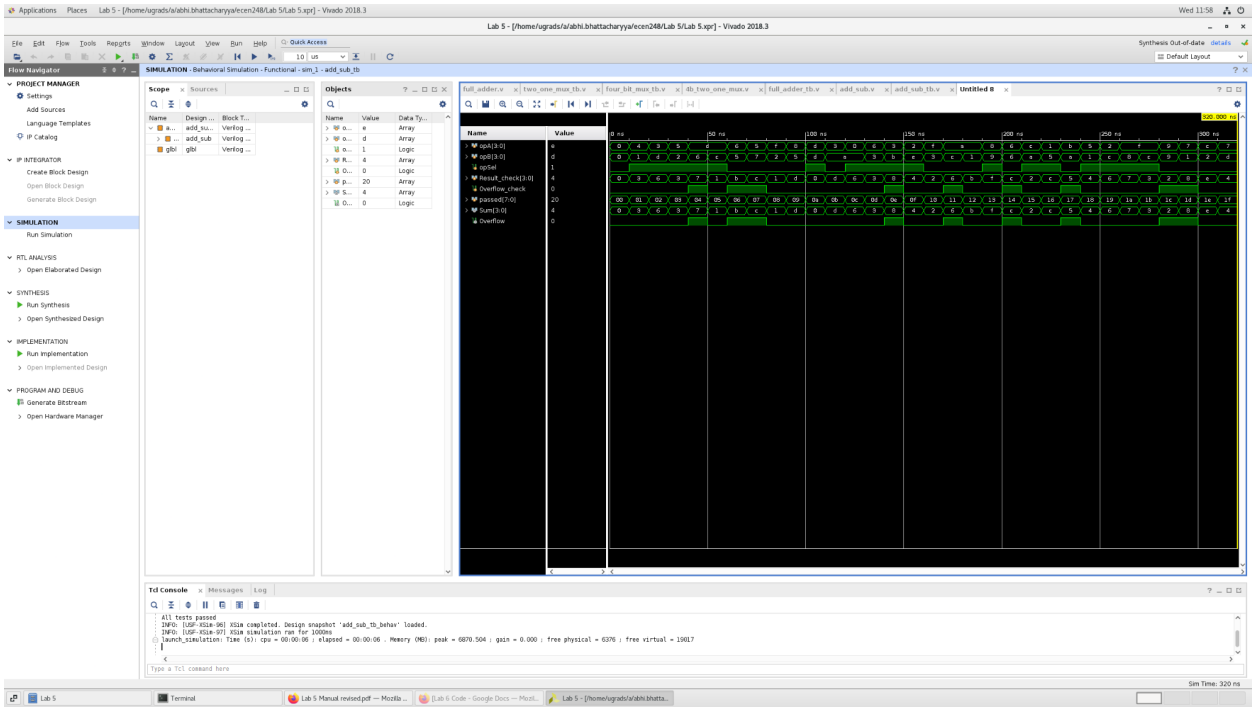

Figure 9: Simulating the Add-Sub Unit using the Test Bench

Part 3:

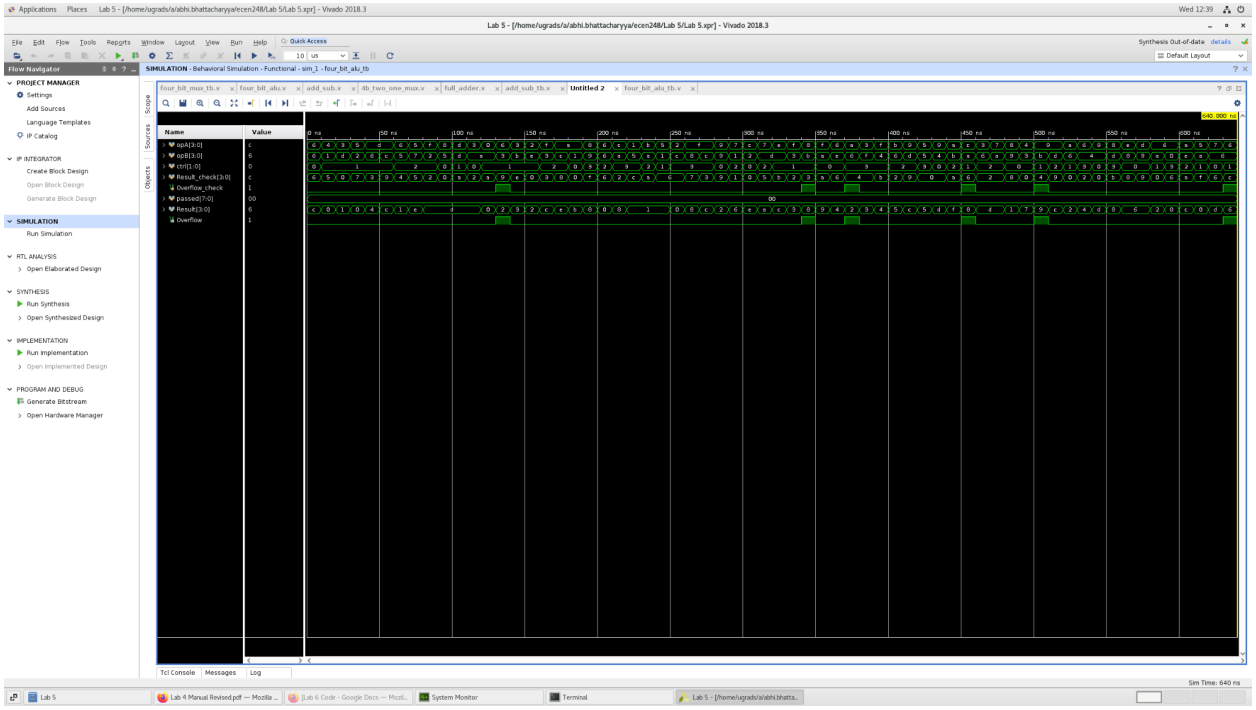

Figure 10: Simulating the 4-bit ALU using the Test Bench

Lab 5 Demo:

For this lab, tasks 1-5 defined in the Lab 5 Note were completed through Parts 1, 2, and 3 of the lab. The full code was provided in the **Design** section and the full simulations for each module were provided in the **Results** section. This completes all demonstration requirements for Lab 5.

Conclusion:

In conclusion, this lab report has introduced digital logic simulation and Verilog, two essential tools for the design and verification of digital circuits. This report has explored the fundamentals of designing digital logic circuits in Verilog by completing modules to build a 4-bit ALU in Verilog HDL. This report has also showcased how to utilize simulation software to validate the designs for each of these modules. In addition, this lab has looked at these tools' benefits and drawbacks in comparison to traditional Breadboarding and Schematic-based design, as well as when the structural and dataflow approaches to Verilog HDL are best employed and how they can be used together to design digital circuits. In all, this lab has successfully introduced the foundational components of Verilog and has served well to prepare for future labs dealing with Verilog this semester.

Post-lab Deliverables

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

Included in the Design section.

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.

Included 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 reads much more like a programming language. Explain briefly what it is the test bench is doing.

```
`timescale 1ns / 1ps//time unit is 1 nano second and the simulation steps are in peco seconds
`define STRLEN 32

/*This test bench is full of non-synthesizable constructs, which basically means it is
 *restricted to simulation only!*/
module two_one_mux_tb;

   /*A task is similar to a procedure in the traditional programming language*/
   /*This particular task simply checks the output of our circuit against a
     known answer and prints a message based on the outcome. Additionally,
     this task increments the variable we are using to keep track of the
     number of tests successfully passed.*/
   task passTest;
       input actualOut, expectedOut;
       input [`STRLEN*8:0] testType;
       inout [7:0] passed;

       if(actualOut == expectedOut) begin $display ("%s passed", testType); passed = passed + 1; end
       else $display ("%s failed: %x should be %x", testType, actualOut, expectedOut);
    endtask

    /*this task simply informs the user of the final outcome of the test*/
    task allPassed;
       input [7:0] passed;
       input [7:0] numTests;

       if(passed == numTests) $display ("All tests passed");
       else $display("Some tests failed");
    endtask

    // Inputs
    reg A;
    reg B;
    reg S;
    reg [7:0] passed;

    // Outputs
    wire Y;

    // Instantiate the Unit Under Test (UUT)
    two_one_mux uut (
       .Y(Y),
       .A(A),
       .B(B),
       .S(S)
    );

    initial begin
       // Initialize Inputs
       A = 0;
       B = 0;
       S = 0;
       passed = 0;

       // Add stimulus here
       // Add stimulus here
       /*set input (i.e. stimulus)*/
       /*wait 10 time units*/
       /*check output against known answer*/
       {A, B, S} = {3'b000}; #10; passTest(Y, 1'b0, "Mux Test 1", passed);
       {A, B, S} = {3'b001}; #10; passTest(Y, 1'b0, "Mux Test 2", passed);
       {A, B, S} = {3'b010}; #10; passTest(Y, 1'b0, "Mux Test 3", passed);
       {A, B, S} = {3'b011}; #10; passTest(Y, 1'b1, "Mux Test 4", passed);
       {A, B, S} = {3'b100}; #10; passTest(Y, 1'b1, "Mux Test 5", passed);
       {A, B, S} = {3'b101}; #10; passTest(Y, 1'b0, "Mux Test 6", passed);
       {A, B, S} = {3'b110}; #10; passTest(Y, 1'b1, "Mux Test 7", passed);
       {A, B, S} = {3'b111}; #10; passTest(Y, 1'b1, "Mux Test 8", passed);
       allPassed(passed, 8);//did all the tests pass???
       $stop;//that's all folks!
    end

endmodule
```

Figure 11: 2:1 MUX Test Code

In the test bench code, as shown in Figure 11, the task called “passTest” takes in parameters “expectedOut” and “actualOut” to compare them and test if they are equal. The test passes only if the inputs are equal. Additionally, the task “allPassed” on line 23 simply prints out to console if all tests passed or if some tests failed. On line 41, each of the inputs are initialized and then used to run the task “passTest” to check if the output from the MUX code matches the expected output. If the actual output matches the expected output, each test passes.

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

```
1   `timescale 1ns / 1ps //time unit is 1 nano second and the simulation steps are in peco seconds
2   `define STRLEN 32
3
4   /*This test bench is full of non-synthesizable constructs, which basically means it is
5     *restricted to simulation only!*/
6   module four_bit_mux_tb;
7
8      /*A task is similar to a procedure in the traditional programming language*/
9      /*This particular task simply checks the output of our circuit against a
10      known answer and prints a message based on the outcome. Additionally,
11      this task increments the variable we are using to keep track of the
12      number of tests successfully passed.*/
13     task passTest;
14         input [3:0] actualOut, expectedOut;
15         input [`STRLEN*8:0] testType;
16         inout [7:0] passed;
17
18         if(actualOut == expectedOut) begin $display ("%s passed", testType); passed = passed + 1; end
19         else $display ("%s failed: %x should be %x", testType, actualOut, expectedOut);
20     endtask
21
22     /*this task simply informs the user of the final outcome of the test*/
23     task allPassed;
24         input [7:0] passed;
25         input [7:0] numTests;
26
27         if(passed == numTests) $display ("All tests passed");
28         else $display("Some tests failed");
29     endtask
30
31     // Inputs
32     reg [3:0] A;
33     reg [3:0] B;
34     reg S;
35     reg [7:0] passed;
36
37     // Outputs
38     wire [3:0] Y;
39
40     // Instantiate the Unit Under Test (UUT)
41     four_bit_mux uut (
42         .Y(Y),
43         .A(A),
44         .B(B),
45         .S(S)
46     );
47
48     initial begin
49         // Initialize Inputs
50         A = 0;
51         B = 0;
52         S = 0;
53         passed = 0;
54
55         // Add stimulus here
56         /*perform a walking one's test on inputs*/
57         /*wait 10 time units*/
58         /*check output against known answer*/
59         {A, B, S} = {4'b0000, 4'b0000, 1'b0}; #10; passTest(Y, 4'b0000, "4-bit Mux Test 1", passed);
60         {A, B, S} = {4'b0001, 4'b0000, 1'b0}; #10; passTest(Y, 4'b0001, "4-bit Mux Test 2", passed);
61         {A, B, S} = {4'b0010, 4'b0000, 1'b0}; #10; passTest(Y, 4'b0010, "4-bit Mux Test 3", passed);
62         {A, B, S} = {4'b0100, 4'b0000, 1'b0}; #10; passTest(Y, 4'b0100, "4-bit Mux Test 4", passed);
63         {A, B, S} = {4'b1000, 4'b0000, 1'b0}; #10; passTest(Y, 4'b1000, "4-bit Mux Test 5", passed);
64         {A, B, S} = {4'b1111, 4'b0000, 1'b0}; #10; passTest(Y, 4'b1111, "4-bit Mux Test 6", passed);
65         {A, B, S} = {4'b1111, 4'b0000, 1'b1}; #10; passTest(Y, 4'b0000, "4-bit Mux Test 7", passed);
66         {A, B, S} = {4'b0000, 4'b0001, 1'b1}; #10; passTest(Y, 4'b0001, "4-bit Mux Test 8", passed);
67         {A, B, S} = {4'b0000, 4'b0010, 1'b1}; #10; passTest(Y, 4'b0010, "4-bit Mux Test 9", passed);
68         {A, B, S} = {4'b0000, 4'b0100, 1'b1}; #10; passTest(Y, 4'b0100, "4-bit Mux Test 10", passed);
69         {A, B, S} = {4'b0000, 4'b1000, 1'b1}; #10; passTest(Y, 4'b1000, "4-bit Mux Test 11", passed);
70         {A, B, S} = {4'b0000, 4'b1111, 1'b0}; #10; passTest(Y, 4'b0000, "4-bit Mux Test 12", passed);
71         {A, B, S} = {4'b0000, 4'b1111, 1'b1}; #10; passTest(Y, 4'b1111, "4-bit Mux Test 13", passed);
72         {A, B, S} = {4'b1010, 4'b1111, 1'b0}; #10; passTest(Y, 4'b1010, "4-bit Mux Test 14", passed);
73         {A, B, S} = {4'b1010, 4'b1111, 1'b1}; #10; passTest(Y, 4'b1111, "4-bit Mux Test 15", passed);
74         {A, B, S} = {4'b1111, 4'b1111, 1'b0}; #10; passTest(Y, 4'b1111, "4-bit Mux Test 16", passed);
75         allPassed(passed, 16);//did all the tests pass???
76         $stop;//that's all folks!
77     end
78
79  endmodule
```

Figure 12: 4-Bit 2:1 MUX Test Code

As shown in Figure 12 - No, not all the possible input cases are being tested, because the test bench goes through each input and then compares it to the input of the expected value. So this only tests *some* input cases, not all of them.

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?

Circuit design using breadboarding techniques is much simpler to understand and more accessible. Since it is hands-on, you can easily build a circuit on a breadboard, debug it, and get it running/tested. One of the most important parts is being able to measure voltages all over and inside the circuit, which is harder to do inside HDL as it requires advanced debugging and looking at individual variables.

Conversely, using an HDL has its own advantages. More complex circuits are much easier to make in HDL as you only need to define the component modules once in structural Verilog, and then you can build massive circuits by defining many instances of the same basic modules and connecting them together.

The main disadvantage of breadboarding is large circuits quickly add up, taking large amounts of space and being extremely complex and difficult to debug - any mistakes you make in one of the building blocks will ruin the whole circuit. The main disadvantage of HDL is that it takes time to learn and is not as accessible or easy to understand as breadboarding. It also has limitations as it only simulates real hardware, so it is harder to evaluate delay or get a final hardware design from HDL.

Personally, I prefer breadboarding and hardware schematics as they better fit the work I do.

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?

HDLs might be preferred over schematics when a circuit is designed for use in an FPGA. Since HDLs are designed to build circuits on FPGAs, it is vastly easier to just program a digital circuit in an HDL and then burn it onto an FPGA. The main disadvantage to using an HDL compared to a schematic is that specialized devices which use specific ICs and gates and which are not suitable for FPGAs must be described in a schematic. One example of this is power electronics or motherboards for a more complex device which have to interface with multiple different circuits and systems in a device. In my opinion, I much prefer schematic design as I am usually dealing with designing boards to work with or add onto existing devices, but HDL does have its place in specialized hardware design.

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?

Using the structural approach, Verilog code is written as a hierarchy of interconnected modules. Each module represents a physical component of the design and is using wires for its inputs, outputs, and internal logic. These wires are connected between different modules to create logic circuits. The structural approach is particularly useful when designing large and complex circuits that consist of many interconnected components.

Using the dataflow approach, Verilog code is written as a set of concurrent assignments that define how data flows through the circuit. In this way, the dataflow approach is similar to combinatorial logic in code. The outputs of a circuit are determined solely by the provided values of the inputs. The dataflow approach is particularly useful when designing smaller and simpler circuits that rely heavily on arithmetic and logical operations.

You would most often use the structural approach when designing complex circuits with many interconnected components because the structural approach allows for a modular and hierarchical design approach that can be easier to understand and modify. Conversely, the dataflow approach is often used when designing smaller and simpler circuits that rely heavily on arithmetic and logical operations, as it allows for a more concise and efficient design. In the real world, many designs (such as the code we used in this lab) use a combination of both approaches to design the full circuit.

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 most enjoyed the feeling of satisfaction I got as I finished each part. Seeing the simulation run and return all results passing and all green was very satisfying.

I least enjoyed the tedium of having to go through code and complete small parts. I would have preferred if the comments were more helpful rather than just giving minor hints as to what may be missing. I also really did not like having to write all the code rather than just being able to copy it. The code should have been provided in the ECEN-248 repository as retyping it all is a major waste of time.

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?

The provided code was missing parts, but it was very unclear what was missing and how to fix the code if things went wrong. For example, some modules had a one-line module () instantiation, while others had multiple lines inside of the parenthesis and were missing components inside the parentheses. This was extremely confusing. I think that more time needs to be spent explaining the code and that items such as input parameters should just be given as if they are done wrong, the whole file will not run and will be filled with syntax errors.

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

I think this lab needs more hands-on help in lectures or from the TAs. Having never coded in Verilog before, I found it very confusing to fill in the missing code. Revisiting it after having spent more time in Verilog, this lab was very easy, but since this is supposed to be an introduction to Verilog, having more guidance would be very helpful. I did get a one-day extension from my TA which was really helpful and I would not have been able to finish on time without this extension.