Objective

The purpose of this lab was to move from a total bread board circuit and towards learning and appreciating how modern digital design is implemented. This lab will expose students to Verilog. The student will construct in Verilog some of the circuits from the previous labs including a 1-bit wide, 2:1 multiplexer, a full adder, a ripple carry adder, a 4-bit wide, 2:1 multiplexer and finally a 4-bit Arithmetic Logic Unit. Some Linux knowledge will be thrown in for good measure.

Design

Experiment 1

For **Experiment 1**, the ISE Project Navigator was set up and a simple 1-bit 2:1 multiplexer module was created *(see Code Block 1)*. This module was a basic first introduction into the world of Verilog. Once written, it was then ran with a supplied test bench file.

Code Block 1: 1-bit 2:1 MUX

```
1  `timescale 1ns / 1ps
2  `default_nettype none
3  //////////////////////////////////////////////////////////////////////////////////
4  // Create Date: 15:21:04 10/10/2016
5  // Module Name: two_one_mux
6  // Author: Joseph M Martinsen
7  //
8  //////////////////////////////////////////////////////////////////////////////////
9  module two_one_mux(Y, A, B, S);
10     //declare output and input wires
11     output wire Y;
12     input wire A, B, S;
13
14     // Declare internal nets
```

```
15      wire notS; // Wire to carry not S
16      wire andA; // Wire to carry andA
17      wire andB; // Wire to carry andB
18
19      // instatiate gate-level modules
20      not not0(notS, S); // invert S
21      and and0(andA, notS, A); // and of notS and A
22      and and1(andB, S, B); // and of S and B
23      or or0(Y, andA, andB); // or andA and and B
24  endmodule
```

Experiment 2

Experiment 2 consisted of creating three modules in order to build a simple 4-bit ALU later on in **Experiment 3**. The first component was building upon the 1-bit 2:1 MUX from earlier and creating a 4 bit 2:1 MUX by combing two 1-bit 2:1 MUXs *(see Code Block 2)*. This module was then tested against the supplied full a test bench.

Code Block 2: 4-bit 2:1 MUX

```
1   `timescale 1ns / 1ps
2   `default_nettype none
3   //////////////////////////////////////////////////////////////////////////////////
4   // Create Date: 15:52:35 10/10/2016
5   // Module Name: four_bit_mux
6   // Author: Joseph M Martinsen
7   //
8   //////////////////////////////////////////////////////////////////////////////////
9   module four_bit_mux(Y,A,B,S);
10      // Declare output and input prots
11      output wire [3:0] Y; // Y output wire is 4 bit wide
12      input wire [3:0] A, B; // A and B are 4-bit wires
13      input wire S; // Select wire is 1 bit wide wire
```

```
14
15      // Initialize user-defined 2:1 MUX modules
16      two_one_mux MUX0(Y[0], A[0], B[0], S);
17      two_one_mux MUX1(Y[1], A[1], B[1], S);
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  endmodule
```

Next, a full adder module was created. This module functioned with the same logic as previously built full adders. This module *(Code Block 3)* included the first uses of assign for the student. This module was then tested against the supplied full adder test bench.

Code Block 3: Full Adder

```
1   `timescale 1ns / 1ps
2   `default_nettype none
3   //////////////////////////////////////////////////////////////////
4   // Create Date: 16:06:23 10/10/2016
5   // Module Name: full_adder
6   // Author: Joseph M Martinsen
7   //
8   //////////////////////////////////////////////////////////////////
9   module full_adder(S, Cout, A, B, Cin );
10      // Declare input and output ports
11      input wire A;
12      input wire B;
13      input wire Cin;
14      output wire S
15      output wire Cout;
16
17      // Declare wires
18      wire andBCin;
19      wire andACin;
```

```
20      wire andAB;
21
22      // Use dataflow to create gatelevel commands
23      assign S = A ^ B ^ Cin; // A XOR B XOR Cin
24      assign andAB = A & B; // A AND B
25      assign andBCin = B & Cin; // B AND Cin
26      assign andACin = A & Cin; // A AND Cin
27      // andAB OR andBCin OR andACin
28      assign Cout = andAB | andBCin | andACin;
29  endmodule
```

Finally, an Addition/Subtraction unit was designed much like the !addition/subtraction from the full adder in Lab 05. This unit included the previously created full adder in order to make a ripple carry adder. This module *Code Block 4* was then tested against the supplied Add-Sub test bench.

Code Block 4: Add Sub Module

```
1  `timescale 1ns / 1ps
2  `default_nettype none
3  //////////////////////////////////////////////////////////////////
4  // Create Date: 16:24:12 10/10/2016
5  // Module Name: add_sub
6  // Author: Joseph M Martinsen
7  //
8  //////////////////////////////////////////////////////////////////
9  module add_sub(
10          //Declare output and input
11          output wire [3:0] Sum, // 4-bit Results
12          output wire Overflow, // 1-bit wire for overflow
13          input wire [3:0] opA, opB, // 4-bit opperands
14          input wire opSel // if opSel =1, then subtractadd_sub
15  );
```

```
16      // declare internal nets
17      wire [3:0] notB;
18      wire c0, c1, c2, c3;
19
20      // Create complement logic
21      assign notB[0] = opB[0] ^ opSel;
22      assign notB[1] = opB[1] ^ opSel;
23      assign notB[2] = opB[2] ^ opSel;
24      assign notB[3] = opB[3] ^ opSel;
25
26      // full adders to create a ripple carry adder
27      full_adder adder0(Sum[0], c0, opA[0], notB[0], opSel);
28      full_adder adder1(Sum[1], c1, opA[1], notB[1], c0);
29      full_adder adder2(Sum[2], c2, opA[2], notB[2], c1);
30      full_adder adder3(Sum[3], c3, opA[3], notB[3], c2);
31
32      // Overflow detection logic
33      assign Overflow = c2 ^ c3;
34  endmodule
```

Experiment 3

In the final experiment, all the parts from **Experiment 2** were incorporated in order to create a 4-bit ALU. This module specifically utilizied the 4-bit MUX module as well as the add sum module. This final 4-bit ALU module *(Code Block 5)*, was then tested against the 4-bit ALU test bench.

Code Block 5: 4-Bit ALU

```
1  `timescale 1ns / 1ps
2  `default_nettype none
3  //////////////////////////////////////////////////////////////////
4  // Create Date: 16:47:31 10/10/2016
5  // Module Name: four_bit_alu
```

```
6  // Author: Joseph M Martinsen
7  //
8  ////////////////////////////////////////////////////////////////
9  module four_bit_alu(
10     output wire [3:0] Result, // 4-bit output
11     output wire Overflow, // 1-bit overflow
12     input wire [3:0] opA, opB, // 4-bit operands
13     input wire [1:0] ctrl // 2 bit operation select
14         );
15      wire [3:0] Sum; // 4-bit Sum wire
16      wire [3:0] andAB; // 4-bit A AND B wire
17      wire add_sub0; // Not sure if this was used...
18      wire OverflowInit; // Temp Overflow wire
19      // Initialize addsub Module
20      add_sub addsub(Sum, OverflowInit, opA, opB, ctrl[1]);
21
22       // Determine overflow logic
23      assign Overflow = OverflowInit & ctrl[0];
24      // Values of A AND B for each bit
25      assign andAB[0] = opA[0] & opB[0];
26      assign andAB[1] = opA[1] & opB[1];
27      assign andAB[2] = opA[2] & opB[2];
28      assign andAB[3] = opA[3] & opB[3];
29
30      // Initialize 4-bit MUX
31      four_bit_mux four_bit_mux0(Result,andAB,Sum,ctrl[0]);
32 endmodule
```

Results

The following inclued an image of the test results and another image for the waveforms captured during simmulation.

Experiment 1

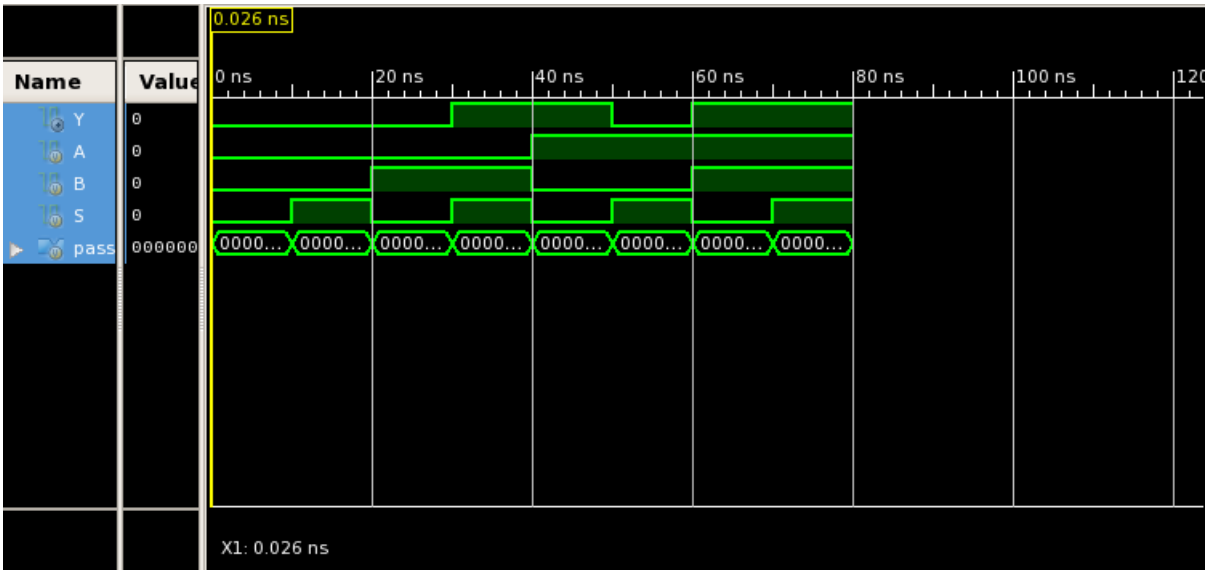

Figure 1: *2-Bit 2:1 MUX Plots*

```
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
                Mux Test 1 passed
                Mux Test 2 passed
                Mux Test 3 passed
                Mux Test 4 passed
                Mux Test 5 passed
                Mux Test 6 passed
                Mux Test 7 passed
                Mux Test 8 passed
All tests passed
Stopped at time : 80 ns :  in File "/home/ugrads/j/josephmart/ecen248/lab06/lab06/two_one_mux_tb.v" Line 69
```

Figure 2: *2-Bit 2:1 MUX Tests*

Experiment 2

Figure 3: *4-Bit 2:1 MUX Plot*

```
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
        4-bit Mux Test 1 passed
        4-bit Mux Test 2 passed
        4-bit Mux Test 3 passed
        4-bit Mux Test 4 passed
        4-bit Mux Test 5 passed
        4-bit Mux Test 6 passed
        4-bit Mux Test 7 passed
        4-bit Mux Test 8 passed
        4-bit Mux Test 9 passed
        4-bit Mux Test 10 passed
        4-bit Mux Test 11 passed
        4-bit Mux Test 12 passed
        4-bit Mux Test 13 passed
        4-bit Mux Test 14 passed
        4-bit Mux Test 15 passed
        4-bit Mux Test 16 passed
All tests passed
Stopped at time : 160 ns :  in File "/home/ugrads/j/josephmart/ecen248/lab06/lab06/four_bit_mux_tb.v" Line 76
```

Figure 4: *4-Bit 2:1 MUX Tests*

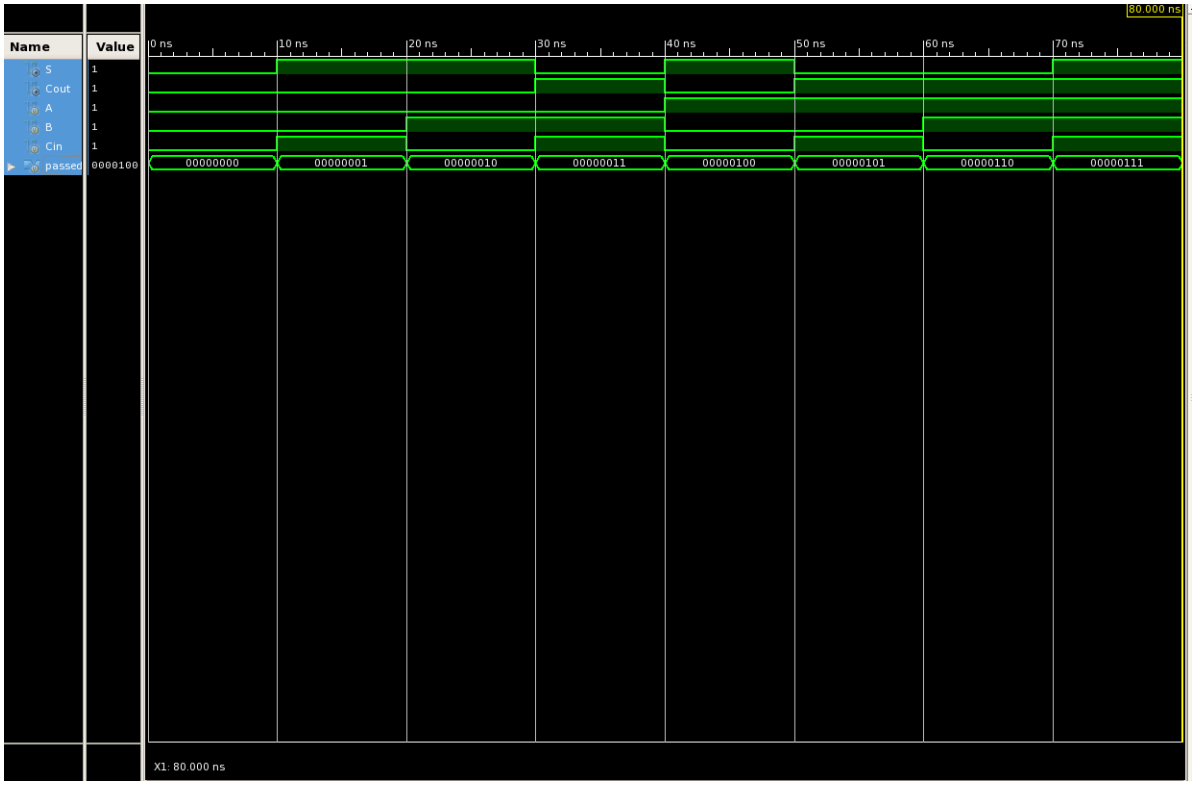

Figure 5: *Full Adder Tests*

```
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
        Full Adder Test 1 passed
        Full Adder Test 2 passed
        Full Adder Test 3 passed
        Full Adder Test 4 passed
        Full Adder Test 5 passed
        Full Adder Test 6 passed
        Full Adder Test 7 passed
        Full Adder Test 8 passed
All tests passed
Stopped at time : 80 ns :  in File "/home/ugrads/j/josephmart/ecen248/lab06/lab06/full_adder_tb.v" Line 59
```

Figure 6: *Full Adder Tests*

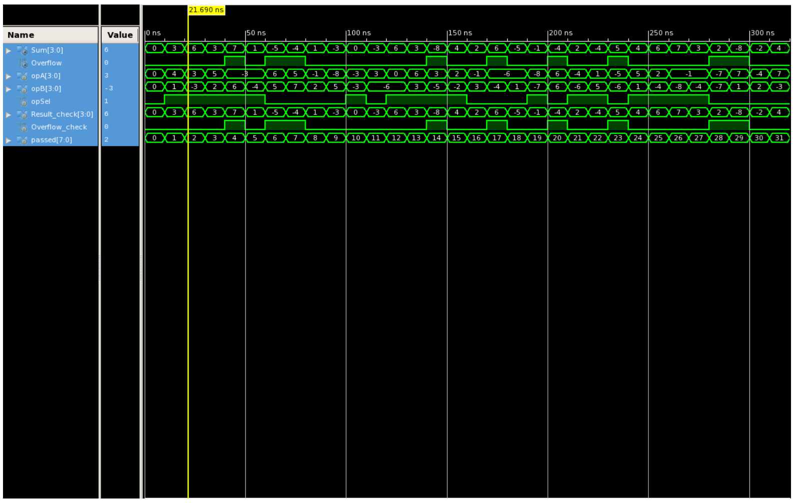

Figure 7: *Add/Sub Tests*

```
Finished circuit initialization process.
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
  Addition/Subtraction Unit Test passed
All tests passed
Stopped at time : 320 ns : in File "/home/ugrads/j/josephmart/ecen248/lab06/lab06/add_sub_tb.v" Line 88
```

Figure 8: *Add/Sub Tests*

Experiment 3

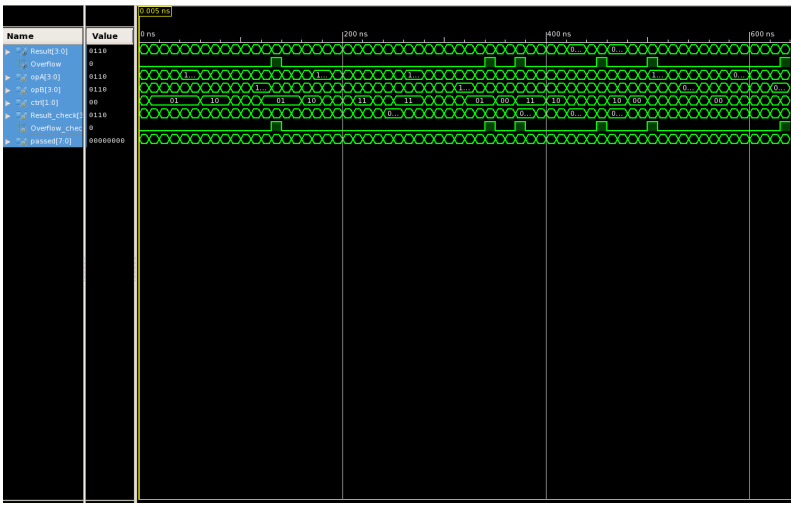

Figure 9: *ALU Plot*

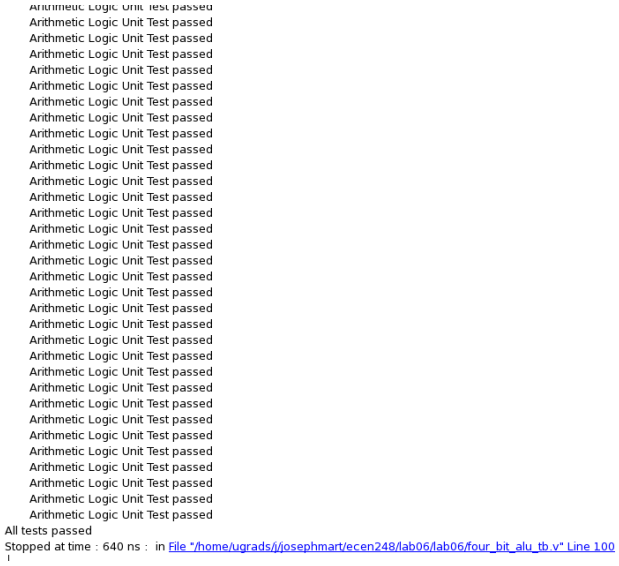

Figure 10: *ALU Test*

Conclutions

In this lab, Verilog was used in lab for the very first time. This lab was very different from the previous labs because no breadboards or wires were used. Also, I was able

to finish this lab. This lab was similar to previous labs because the circuitry and logic exactly the same. The labs were just converted and implemented in a Verilog format.

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!**

   In the report

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

   In the report

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

   The test bench code begins by first creating two tasks. The task looks similiar to a function. The task takes in the output of the student created MUX and compares it to the correct value and lets the user know if they passed or fail a particular test. If the user passed all the tests, let the user know. Next, the wires and MUX is initialized. Finally the output values are compared to actual values through the function created earlier. The other function is then used to compare and see if all checks passed.

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, 3'b000 represents 000000. Only 8 values (3'b000, 3'b001, 3'b010, 3'b011, 3'b100, 3'b101, 3'b110, and 3'b111) are compared. Also, as shown by our clever TA, 6 of the tests could be passed with a simple AND gate.

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.**

   The biggest and most obvius difference is that one is done on a computer (Verilog) and the other is physicaly built (breadboarding). The advantage of Verilog is that it is easier to find a mistake and correct it compared to a breadboard circuit which can prove to be very hard to find which wire is not in the correct position or which part is physically broken. The advantage of a breadboard over Verilog is that Verilog is only a simulation, breadboaring is building an actual circuit.

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?**

   Structual abstraction is used to describe a netlist as a text alternative to a diagram. Dataflow abstraction can use Boolean equations or conditional operators similiar to a truth table.
   Dataflow is useful for test bench, structual is useful for building IC's.
   An example of structual include: "not G1 (N3,C0), G2 (N5,N2), G3 (N6,N3); nand G4 (N1,A0,B0);"
   An example of dataflow include: "assign N2 = (A0 — B0); assign C1 = ((N1 & C0) — N2);"[1]

Student Feedback

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

I was able to finish on time. I didn't like how there was not much info in **Experiment 3**

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

   Experiment 3 was not very clear. Initializing previously created Modules was not explained very well. It was just shown in some given code in **Experiment 2**

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

   Explain Module calling better.

References

[1] Charles Kime & Thomas Kaminski *Logic and Computer Design Fundamentals http://www.cs.bilkent.edu.tr/ will/courses/CS223/Verilog/LCDF3$_V$erilog$_C$h$_4$.pdf*