Objective

In the last lab, two different methods to create digital circuits in Verilog was introduced. One was structural method and the other was dataflow modeling. In this lab, a 3^{rd} method will be used, behavioral modeling.

This lab consisted of three separate experiments. For **Experiment 1**, the multiplexers created last lab will be converted into behavioral Verilog. In **Experiment 2**, students will describe, using behavioral Verilog, binary encoders and decoders. Finally, **Experiment 3** will introduce logic synthesis, as well as translating HDL code into implementable digital logic. The code written in Experiment 2 will be synthesized and programmed onto a Spartan 3E in order to further verify that the encoder and decoder described is in fact correct.

Design

Experiment 1

In the first experiment, the student was able to practice their newly formed knowledge of behavioral Verilog by converting preiovosly created structaul code into behavioral. There were three separate conventions, one for a *1- Bit 2:1 MUX*, another for a *4-Bit 2:1 MUX*, and finally *4- Bit 4:1 MUX*. The following is the code wirtten for each convertion.

Code Block 1: 1-Bit 2:1 MUX Behavioral

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////
3  // Module Name:          two_one_mux_behavioral
4  // Author: Joseph Martinsen
5  //
6  //////////////////////////////////////////////////////
7
8  module two_one_mux(Y, A, B, S);
9      /*output and input ports*/
```

```
10      output reg Y;
11
12      input wire A;
13      input wire B;
14      input wire S;
15
16      // 2:1 Behavioral Mux
17      always@(A or B or S)
18          begin
19              if(S == 1'b0)
20                  Y = A;
21              else
22                  Y = B;
23          end
24  endmodule
```

Code Block 2: 4-Bit 2:1 MUX Behavioral

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Module Name: four_bit_mux_behavioral
4   // Author: Joseph Martinsen
5   //
6   //////////////////////////////////////////////////////////////////////////////////
7   module four_bit_mux(Y, A, B, S);
8       /*output and input ports*/
9       output reg [3:0]Y;
10
11      input wire [3:0]A;
12      input wire [3:0]B;
13      input wire S;
14
```

```
15      // Behavioral logic for 4-bit 2:1 Mux
16      always@(A or B or S)
17              begin
18                      if(S == 1'b0)
19                              Y = A;
20                      else
21                              Y = B;
22              end
23  endmodule
```

Code Block 3: 4-Bit 4:1 MUX Behavioral

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////
3   // Module Name:            mux_4bit_4to1
4   // Author: Joseph Martinsen
5   //
6   //////////////////////////////////////////////////////
7
8   module mux_4bit_4to1(Y, A, B, C, D, S);
9       // Input/output Ports
10      output reg [3:0] Y;
11      input wire [3:0] A,B,C,D;
12      input wire [1:0] S;
13
14      // Logic for 4-bit 4:1 Mux
15      always@(*)
16              case(S)
17                      2'b00: Y = A;
18                      2'b01: Y = B;
19                      2'b10: Y = C;
20                      2'b11: Y = D;
```

```
21              endcase
22
23  endmodule
```

Experiment 2

In the second experiment, the student was able to move into creating new behavioral code for the first time. A 2:4 decoder, 4:2 encoder, and a priority encoder were all implemented. All of which had not been mentioned in lecture.

Code Block 4: 2:4 Decoder

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////
3   // Module Name:            two_four_decoder
4   // Author: Joseph Martinsen
5   //
6   //////////////////////////////////////////////////////////////////
7
8   module two_four_decoder(
9       // Input/Output Ports
10              input wire [1:0] W,
11              input wire En,
12              output reg [3:0] Y
13             );
14
15      // Behavioral logic for 2:4 Decoder
16       always@ (En or W)
17              begin
18                      if(En == 1'b1)
19                              case(W)
20                                2'b00: Y = 4'b0001;
21                                2'b01: Y = 4'b0010;
```

```
                                2'b10: Y = 4'b0100;
                                2'b11: Y = 4'b1000;
                            endcase
                    else
                            Y = 4'b0000;
            end
endmodule
```

Code Block 5: 4:2 Encoder

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Module Name:          four_two_encoder
// Author: Joseph Martinsen
//
//////////////////////////////////////////////////////////////////////////////////
module four_two_encoder(
    // Input and output ports
    input wire [3:0] W,
    output wire zero,
    output reg [1:0] Y
    );

    // Assign case for no W
    assign zero = (W == 4'b0000);

    // Behavioral logci for 4:2 Encoder
    always@(W)
        begin
            case(W)
                4'b0001: Y = 2'b00;
                4'b0010: Y = 2'b01;
```

```
23                              4'b0100: Y = 2'b10;
24                              4'b1000: Y = 2'b11;
25                              default: Y = 2'bXX;
26                      endcase
27              end
28
29  endmodule
```

Code Block 6: Priority Encoder

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////
3   // Module Name:          priority_encoder
4   // Author: Joseph Martinsen
5   //
6   //////////////////////////////////////////////////////
7   module priority_encoder(input wire [3:0] W, // Input/Ouput prots
8       output wire zero,
9       output reg [1:0] Y
10      );
11
12      // Assign value when W has no change
13      assign zero = (W == 4'b0000);
14
15      // Behavioral logic for Priority Encoder
16      always@(W)
17              begin
18                      casex(W)
19                              4'b0001: Y = 2'b00;
20                              4'b001X: Y = 2'b01;
21                              4'b01XX: Y = 2'b10;
22                              4'b1XXX: Y = 2'b11;
```

```
23                                  default: Y = 2'bXX;
24                          endcase
25                  end
26
27  endmodule
```

Experiment 3

In the final experiment, no new code was written but the code written in experiment 2 was implemented onto the Spartan 3E board in lab. The components on the board allows the student to confirm that the code created actually functions. The buttons and LEDs on the board were used for input and output.

Results

Experiment 1

In the first experiment, the three MUX designs were tested against the appropriate test benches. The following are the results of these tests. All the tests passed in all three cases.

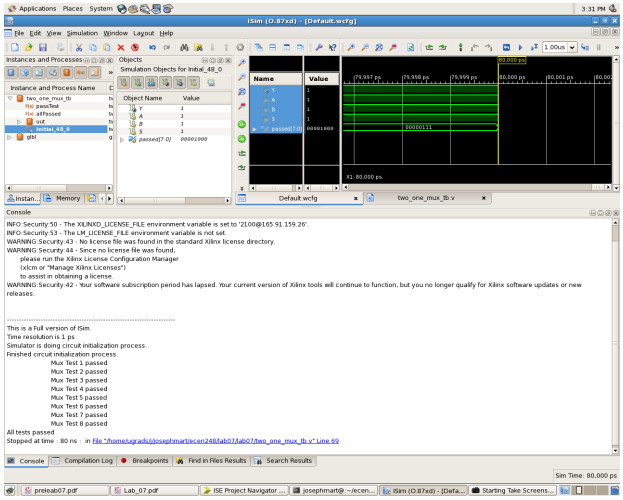

Figure 1: *1-Bit 2:1 Mux Behavioral Tests*

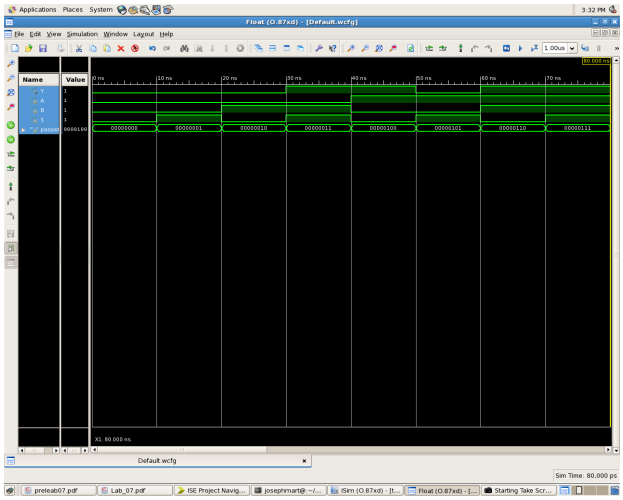

Figure 2: *1-Bit 2:1 Mux Behavioral Graph*

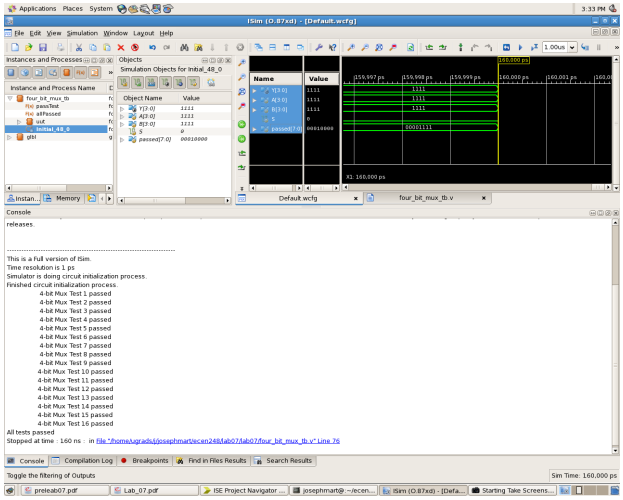

Figure 3: *4-Bit 2:1 MUX Behavioral Tests*

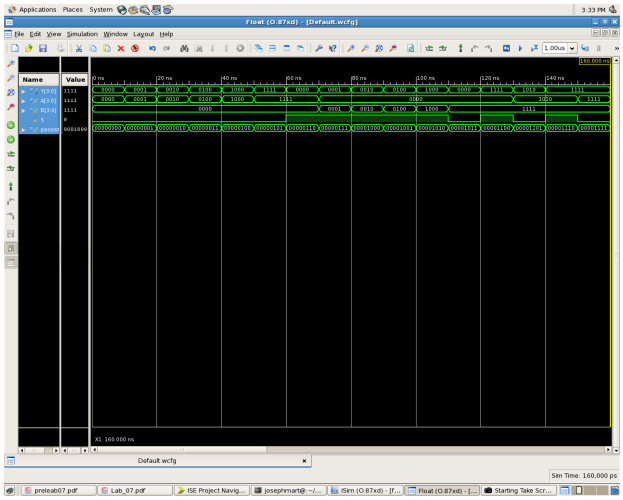

Figure 4: *4-Bit 2:1 MUX Behavioral Graph*

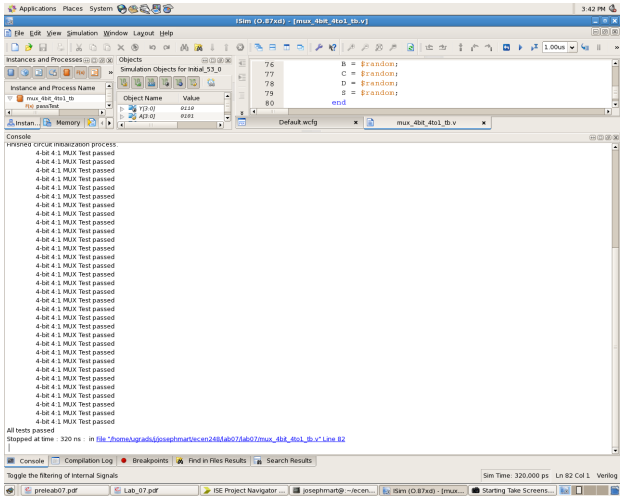

Figure 5: *4-Bit 4:1 MUX Behavioral Tests*

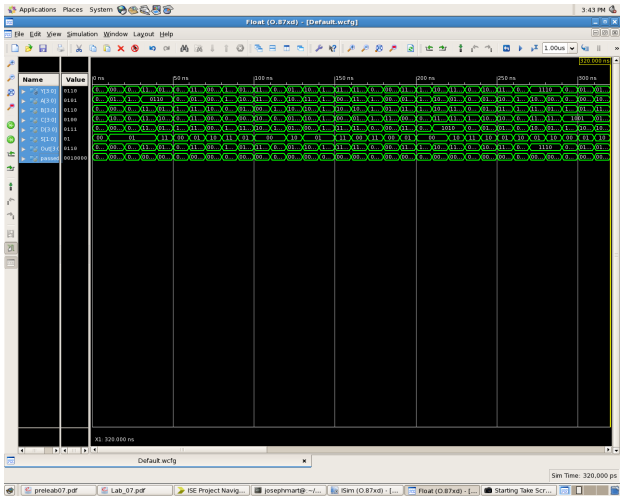

Figure 6: *4-Bit 4:1 MUX Behavioral Graph*

Experiment 2

In the second experiment, the 2:4 decoder, 4:2 encoder, and the priority encoder were all tested against the appropriate test benches. The following are the results of these tests. All the tests passed in all three cases.

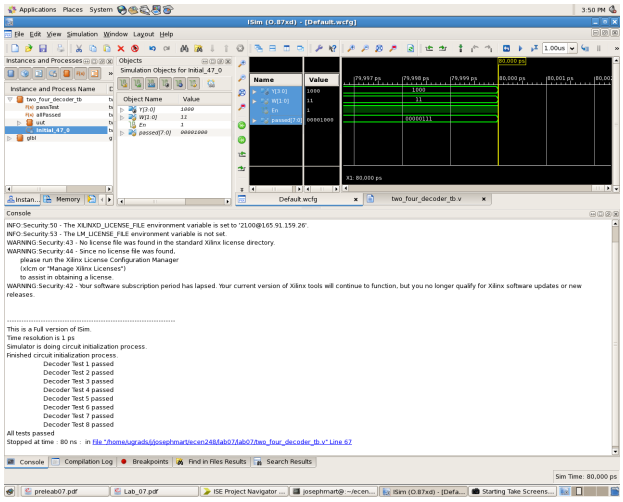

Figure 7: *2:4 Decoder Tests*

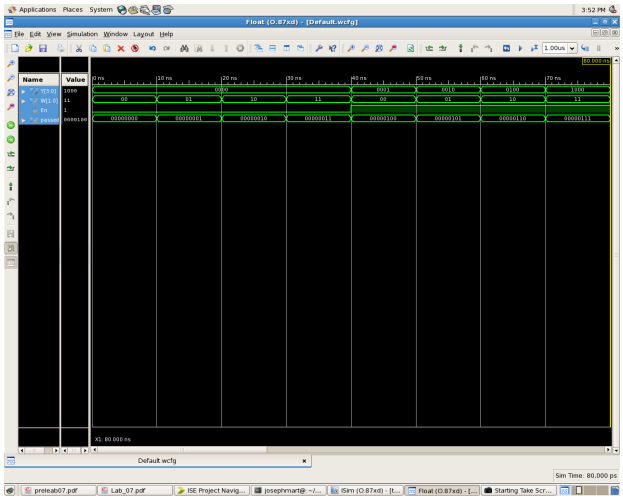

Figure 8: *2:4 Decoder Graph*

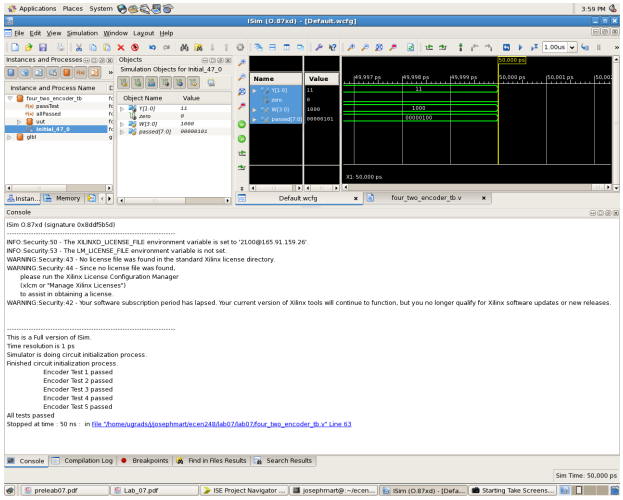

Figure 9: *4:2 Encoder Tests*

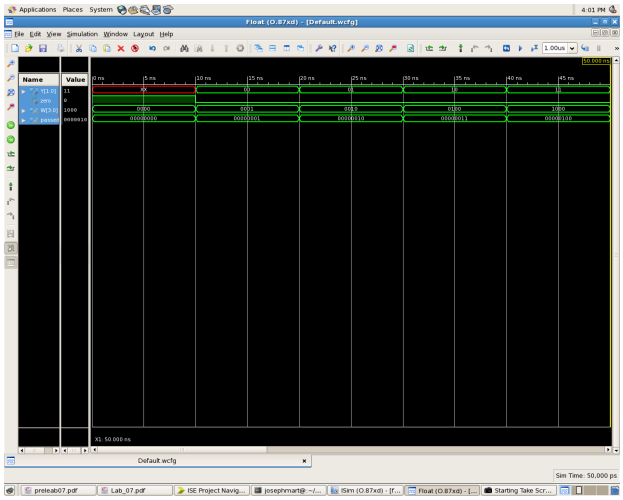

Figure 10: *4:2 Encoder Graph*

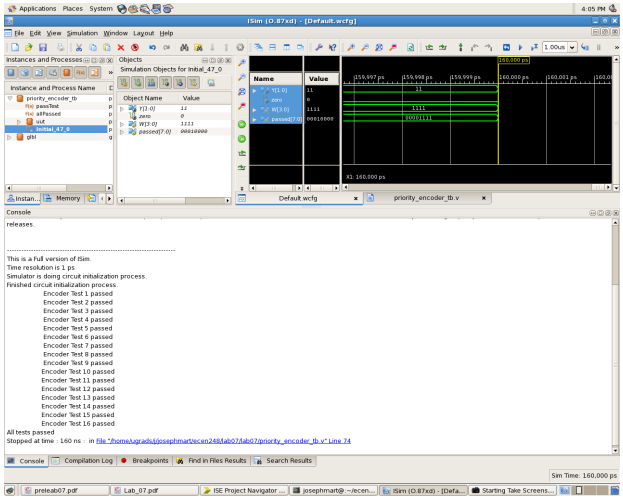

Figure 11: *Priority Encoder Tests*

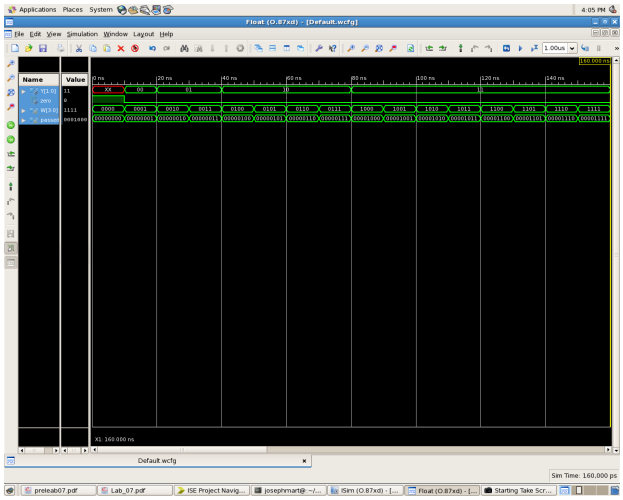

Figure 12: *Priority Encoder Graph*

Experiment 3

In the final part of the lab, the design code was implemented onto the board. Buttons on the board were used for input. LEDs were used for output. The results were then compared the the proper results similiar to the test bench code.

Conclutions

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 Report

2. **Include screen shots of all wave forms captured during simulation in addition to the test bench console output for each test bench simulation.**

   In Report

3. **Provide a comparison between behavioral Verilog used in this weeks lab and the structural and dataflow Verilog used in last weeks lab. What**

might be the advantages and disadvantages of each.

Code Block 7 is an example of dataflow Verilog. *Code Block 8* is an example of behavioral Verilog. Dataflow is useful when logic gates are wanted to be used in the code design. Behavioral Verilog is used when one wants to use code similar to other coding languages and implement if statements

Code Block 7: Add Sub

```
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
```

Code Block 8: 1-Bit 2:1 MUX Behavioral Code

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////
3   // Module Name:          two_one_mux_behavioral
4   // Author: Joseph Martinsen
5   //
6   //////////////////////////////////////////////////////
7
8   module two_one_mux(Y, A, B, S);
9       /*output and input ports*/
10      output reg Y;
11
12      input wire A;
13      input wire B;
14      input wire S;
15
16      // 2:1 Behavioral Mux
17      always@(A or B or S)
18          begin
19              if(S == 1'b0)
```

```
20                                  Y = A;
21                          else
22                                  Y = B;
23              end
24  endmodule
```

4. **Compare the process of bread-boarding digital circuit to implementing a digital circuit on an FPGA. State some advantages and disadvantages of each. Which process do you prefer?**

   The biggest and most obvious difference is that one is done on a computer (Verilog) and then implemented onto a FPGA the other is physically built (breadboarding). The advantage of Verilog and Implementation allows for higher level circuits to be built onto smaller areas. The advantage of a breadboard over FPGA is that it is easier to make changes to components because the size is much larger.

Student Feedback

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

   I enjoyed implementing my design onto the Spartan board. I did not enjoy having to wait for another person to finish their lab so I could use their computer because my Spartan board was not functioning.

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?**

   This lab was very straight forward

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

   Better equipment!!