ECEN 248 - Lab Report

Lab Number: 6

Lab Title: Introduction to Behavioral Verilog and Logic Synthesis

Section Number: 513

Student's Name: Abhishek Bhattacharyya

Student's UIN: 731002289

Date: 3/22/2023

TA: Hesam Mazaheri

Objectives:

The main objective of this lab is to use Verilog for digital circuit simulation on an actual Zybo FPGA board. While the previous lab focused on structural and data flow modeling for describing digital circuits in Verilog HDL, this lab will implement behavioral modeling on real hardware. Behavioral modeling is a higher level of abstraction within Verilog. This lab covers how to simulate the behavior of digital circuits including a 2:1 MUX, 4 bits 2:1 MUX, 4 bits 4:1 MUX, 2:4 binary decoders, 4:2 binary encoders, and the priority encoder. This lab also looks back at Lab 5 to compare the previous code created using only structure and data flow modeling with the behavioral code used in this lab. Finally, this lab also introduces Logic Synthesis, the process of converting HDL code into digital logic used to program the ZYBO Z7-10 board. This board is equipped with a Field Programmable Gate Array (FPGA) which is programmed using the code to test the behavior of the circuit in the real world.

Design:

Lab 5 served to help familiarize students with setting up an environment in Vivado and how the software works. Using this knowledge, a new project was started in Vivado for Lab 6 where the code for each module was completed, along with running each testbench and programming the Zybo board using the provided constraints files.

To start, first open the terminal and run the two commands provided in Lab 5 to start Vivado. Then, create a new project in Vivado named 'lab6' to house all the modules. Now, Vivado is ready to start coding each of the modules for this lab. The first module (2:1 MUX) was provided in the lab manual, and the provided design was tested after adding the design and simulation sources. There were some syntax errors and missing code, but once these were fixed, the simulation ran successfully and screenshots were collected of the console output and waveform window. The above steps (import starter code and simulation testbench, complete the code, and run simulations until all tests passed) were repeated with the rest of the modules to complete functioning code for the 4-bit 2:1 MUX, 4-bit 4:1 MUX, 2:4 binary decoder, 4:2 binary encoder, and the priority encoder. Comments were also added to the source code of the design as required by the lab procedure.

Once all the code has been completed and tested successfully, it is time to test the binary encoder and decoder as well as the priority encoder on the ZYBO board. Following the instructions in the Lab 6 Manual, synthesize the first code file (2:4 decoder) using the synthesis tool and generate a bitstream file using the design constraint files which were provided. Then, connect the board to the computer. Ensure that the JP5 jumper is set to JTAG mode. Then, open the hardware manager, select the board in the options, and continue. Finally, wait for the board to be programmed and then test the board’s functionality for intended behavior. These steps were repeated for the rest of the files (4:2 encoder, priority encoder). During the testing phase, a truth table was made for the 2:4 decoder by observing the outputs generated by each input (flipping the switches) displayed in Table 1. This completes the lab.

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

module two_one_mux(Y, A, B, S); // define the module name and its interface

    output reg Y;   // declare output of type reg since it will be modified in an always block
    input wire A, B, S;     // declare inputs of type wire, wires can drive regs in behavioral statements

    always@(A or B or S)    // always block which triggers whenever A, B, or S changes
        begin   //block constructs together
        //1'b0 represents a 1-bit binary value of 0
        if(S == 1'b0)   //double equals are used for comparisons
            Y = A;
        else
            Y = B;      // instead drive Y with B
    end

endmodule   // designate end of module
```

Figure 1: *Code w/ comments for 2:1 MUX*

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

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

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

     always@(A or B or S)    // always block which triggers whenever A, B, or S changes
        begin   //block constructs together
        //1'b0 represents a 1-bit binary value of 0
        if(S == 1'b0)   //double equals are used for comparisons
            Y = A;
        else
            Y = B;      // instead drive Y with B
    end

endmodule   // designate end of module
```

Figure 2: *Code w/ comments for 4-bit, 2:1 MUX*

```
`timescale 1ns / 1ps
`default_nettype none
/* This module describes a 4-bit, 4:1 multiplexer
using behavioral constructs in Verilog HDL*/
module mux_4bit_4to1(Y, A, B, C, D, S);

    //declare output and input ports
    output reg [3:0] Y; //output is a 4-bit wide reg
    input wire [3:0] A, B, C, D;    //4-bit wide input wires
    input wire [1:0] S;             // select is a 2-bit wide wire


    always@(*) begin  //new Verulog trick - * means triggr when anything changes
        //notice we did not use the begin and end bcause
        // the cas statement is considered in one clause
        case(S) // selection based on S
            2'b00: Y = A;   //when S == 2'b00, connect to mux wire A
            2'b01: Y = B;   //when S == 2'b01, connect to mux wire B
            2'b10: Y = C;   //when S == 2'b10, connect to mux wire C
            2'b11: Y = D;   //when S == s'b11, connect to mux wire D
        endcase     // designates the end of a case statement
    end
endmodule
```

Figure 3: *Code w/ comments for 4-bit, 4:1 MUX*

```
`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_four_decoder_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 [3:0] 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 [1:0] W;
   reg En;

   reg [7:0] passed;

   // Outputs
   wire [3:0] Y;

   // Instantiate the Unit Under Test (UUT)
   two_four_decoder uut (
       .Y(Y),
       .W(W),
       .En(En)
   );

   initial begin
       // Initialize Inputs
       W = 0;
       En = 0;

       passed = 0;

       // Add stimulus here
       /*set input (i.e. stimulus)*/
       /*wait 10 time units*/
       /*check output against known answer*/
       {En, W} = {3'b000}; #10; passTest(Y, 4'b0000, "Decoder Test 1", passed);
       {En, W} = {3'b001}; #10; passTest(Y, 4'b0000, "Decoder Test 2", passed);
       {En, W} = {3'b010}; #10; passTest(Y, 4'b0000, "Decoder Test 3", passed);
       {En, W} = {3'b011}; #10; passTest(Y, 4'b0000, "Decoder Test 4", passed);
       {En, W} = {3'b100}; #10; passTest(Y, 4'b0001, "Decoder Test 5", passed);
       {En, W} = {3'b101}; #10; passTest(Y, 4'b0010, "Decoder Test 6", passed);
       {En, W} = {3'b110}; #10; passTest(Y, 4'b0100, "Decoder Test 7", passed);
       {En, W} = {3'b111}; #10; passTest(Y, 4'b1000, "Decoder Test 8", passed);
       allPassed(passed, 8);//did all the tests pass???
       $stop;//that's all folks!
   end

endmodule
```

Figure 4: *Code w/ comments for 2:4 Decoder*

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

/* module interface for the 4:2 encoder */
module four_two_encoder (
  input wire [3:0] W,
  output reg [1:0] Y, // Y should be a 2-bit output of type reg
  output wire zero
);

/* can mix levels of abstraction! */
assign zero = (W == 4'b0000); // a zero test! notice the use of == rather than =

/* behavioral portion */
always @(*)
begin // not necessary because case is single clause but looks better
  case (W) // selection based on W
    4'b0001: Y = 2'b00; // 2'b signifies a 2-bit binary value
    4'b0010: Y = 2'b01; // w[1] is lit up
    4'b0100: Y = 2'b10;         // fill in the case where only w[2] is lit up
    4'b1000: Y = 2'b11; // w[3] is lit up
    default: Y = 2'bXX; // default covers cases not listed!
  endcase // designates the end of a case statement
end

endmodule
```

Figure 5: *Code w/ comments for 4:2 Encoder*

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

//module describing a priority encoder
module priority_encoder (
  input wire [3:0] W,
  output reg [1:0] Y,
  output wire zero //define inputs and outputs, especially for zero
);

assign zero = (W == 4'b0000); // a zero test! notice the use of == rather than =

always @(*) // trigger when W changes
begin
  casex (W)
    4'b0001: Y = 2'b00; // 2'b signifies a 2-bit binary value
    4'b001X: Y = 2'b01; // w[1] is lit up
    4'b01XX: Y = 2'b10;       // fill in the case where only w[2] is lit up
    4'b1XXX: Y = 2'b11; // w[3] is lit up
    default: Y = 2'bXX; // default covers cases not listed!
  endcase
end

endmodule
```

Figure 6: *Code w/ comments for Priority Encoder*

Results:

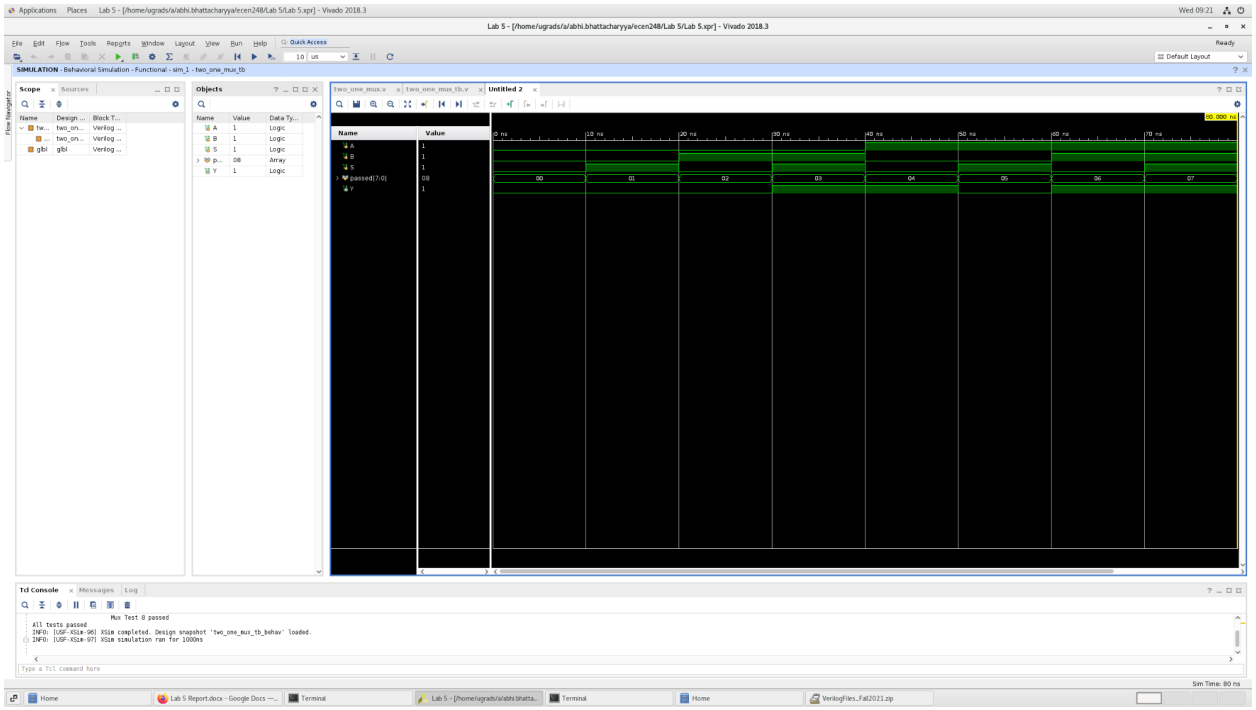


Figure 7: *2:1 MUX waveform and console output*

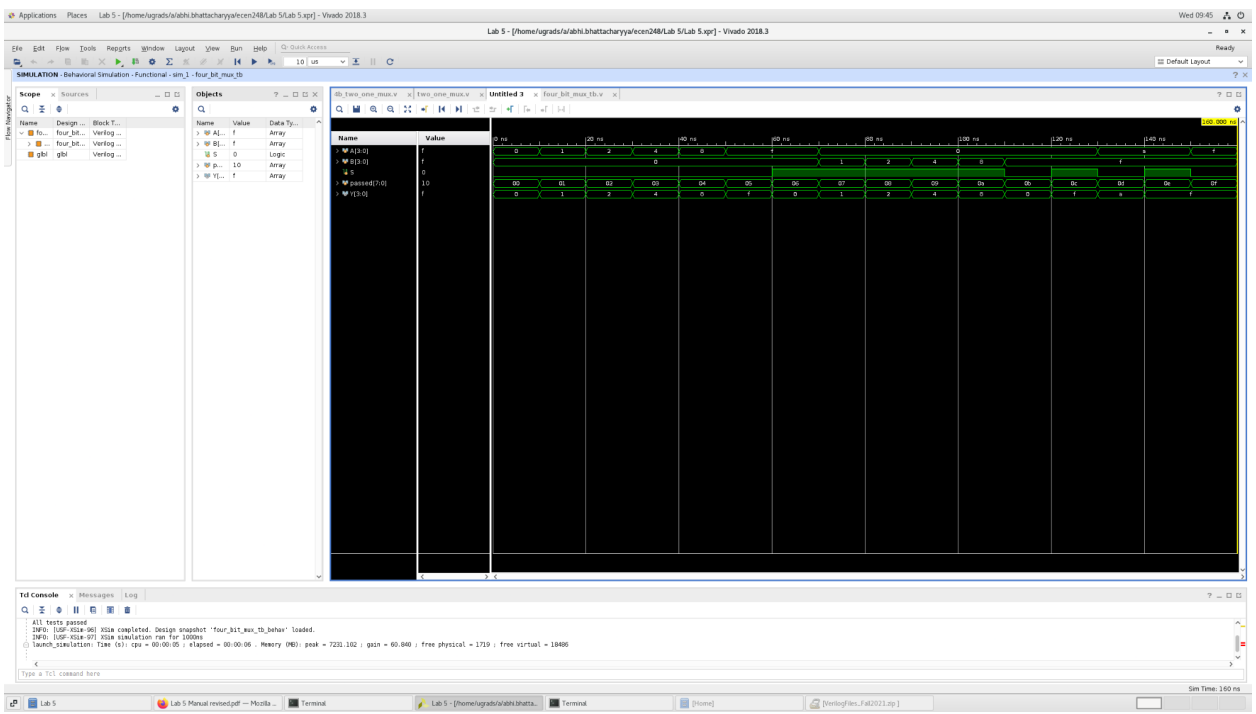


Figure 8: *4-bit, 2:1 MUX waveform and console output*

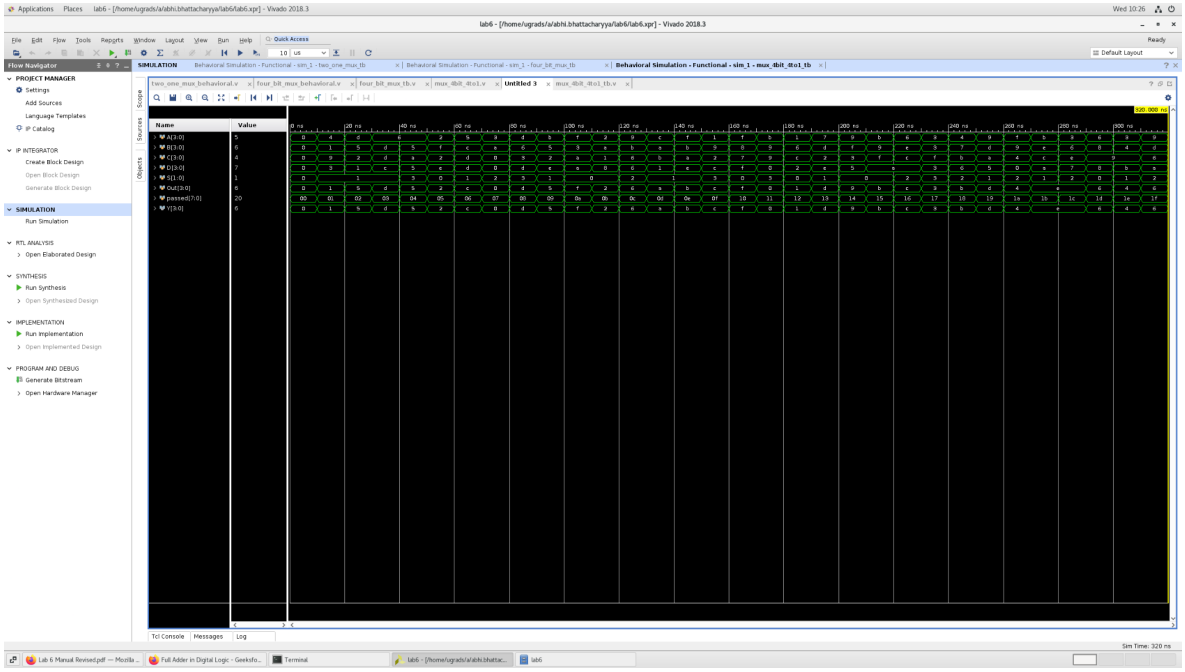


Figure 9: *4-bit, 4:1 MUX waveform and console output*

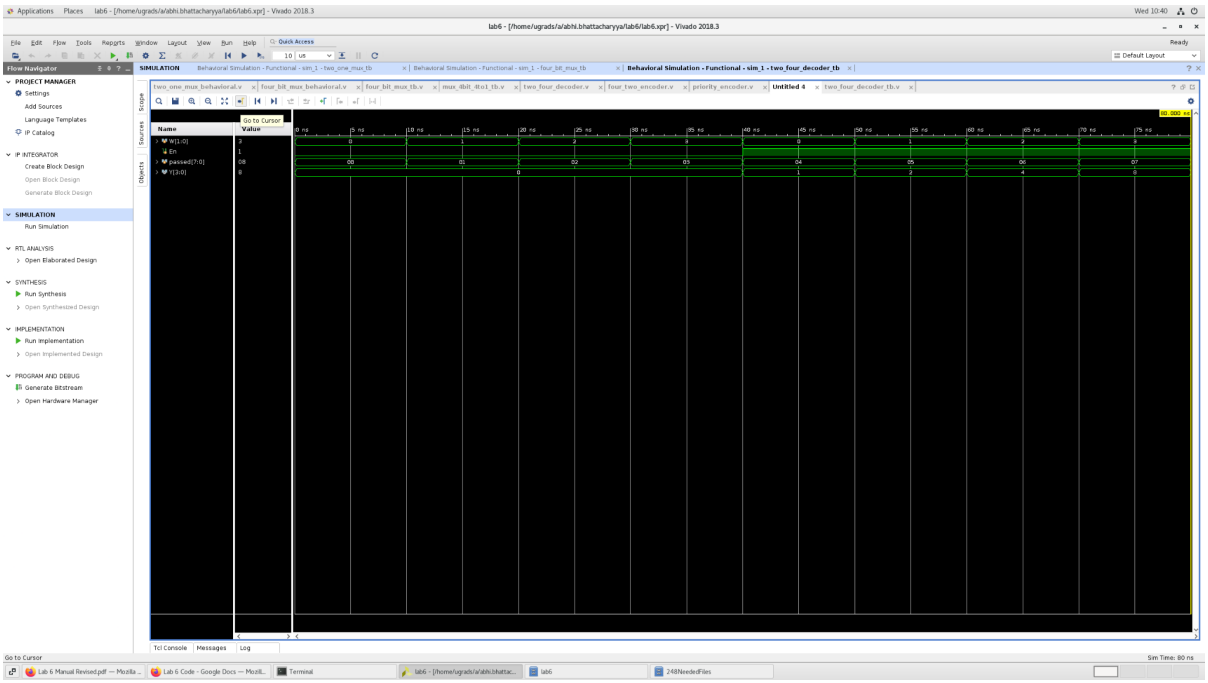


Figure 10: *2:4 Decoder waveform and console output*

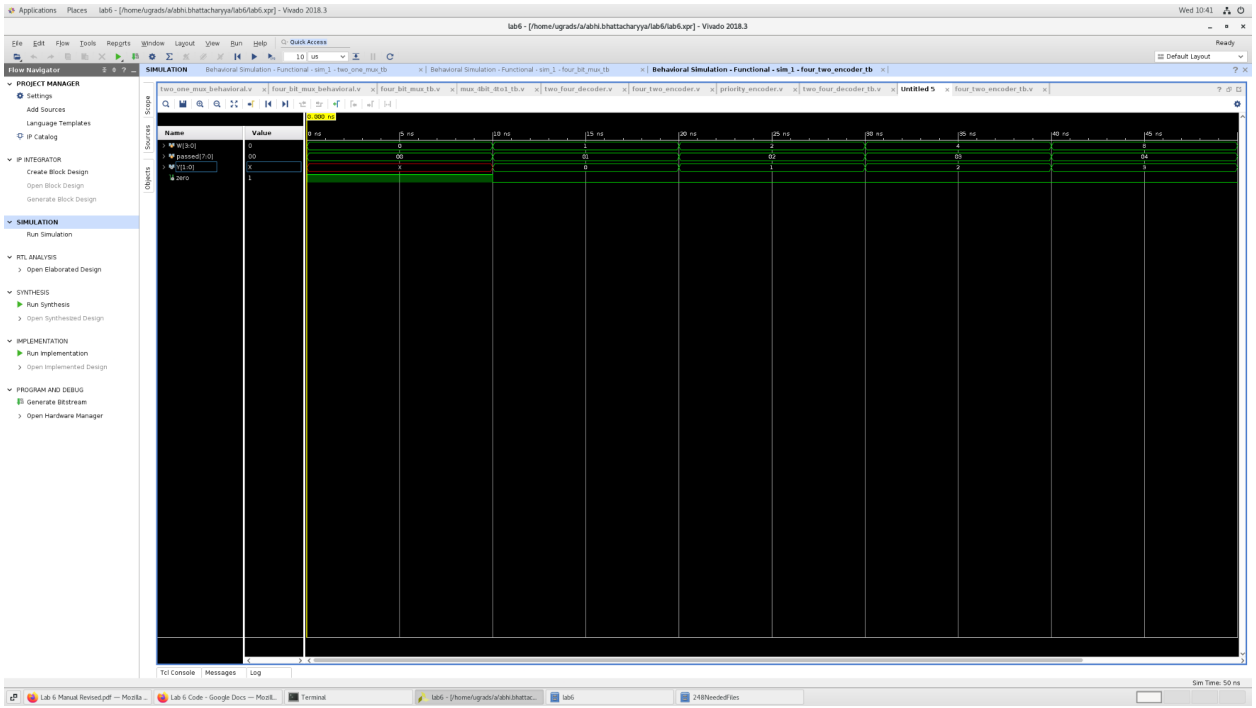


Figure 11: *4:2 Encoder waveform and console output*

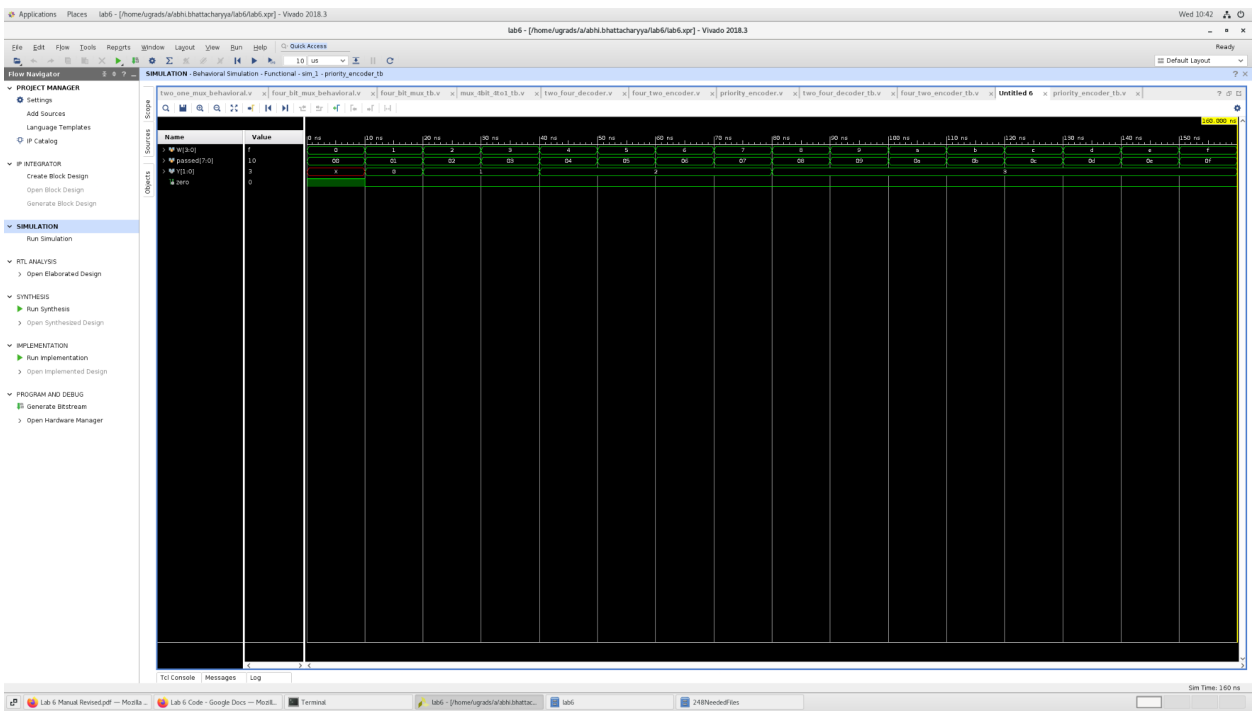


Figure 12: *Priority Encoder waveform and console output*

Table 1: Truth Table for Observed Results, 2:4 Binary Decoder

SW2 (En)	SW1 (w_1)	SW0 (w_0)	LED3 (y_3)	LED2 (y_2)	LED1 (y_1)	LED0 (y_0)
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Lab 6 Demo:

After testing the 2:4 binary decoder on the Zybo board, the 4:2 Binary Encoder and the Priority Encoder were also tested. The results and analysis for these are included here rather than in the **Results** section as the Lab 6 Demo specifically calls for these in the procedure. Below are the expected results (from the lab manual) and actual results (from the board) for the 4:2 Binary Encoder and the Priority Encoder, listed in Table 2.1/2.2 and Table 3.1/3.2 respectively.

Table 2.1: Truth Table for Expected (from the lab manual) Results, 4:2 Binary Encoder

w_3	w_2	w_1	w_0	y_1	y_0	$zero$
0	0	0	0	X	X	1
0	0	0	1	0	0	0
0	0	1	0	0	1	0
0	1	0	0	1	0	0
1	0	0	0	1	1	0

Table 2.2: Truth Table for Actual (from the board) Results, 4:2 Binary Encoder

BTN3 (w_3)	BTN2 (w_2)	BTN1 (w_1)	BTN0 (w_0)	LED2 (Zero)	LED1 (y_1)	LED0 (y_0)
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	1	0
1	0	0	0	0	1	1

Shown above, the actual results for the 4:2 binary encoder were identical to the expected from the lab manual. One interesting behavior was observed for the 4:2 binary encoder: if multiple buttons were held down, the LEDs would change. For example, holding Button 0 and Button 1 lights up LED0. This differs from the Priority Encoder - holding multiple buttons on the priority encoder did not change the LEDs. When holding down button 1 and button 2 (button 2 was pressed first), LED1 was lit, which is consistent with the expected behavior for just pressing button 2. This behavior matches what was expected in the lab manual (shown by the priority encoder having X's, or "don't care" values), meaning the ZYBO board was functioning properly.

Provided below are pictures of two cases of the Zybo board's results, with the inputs and outputs documented. Also listed is the extraneous case of holding down two buttons mentioned in the example above (holding Button 0 and Button 1 lights LED0).

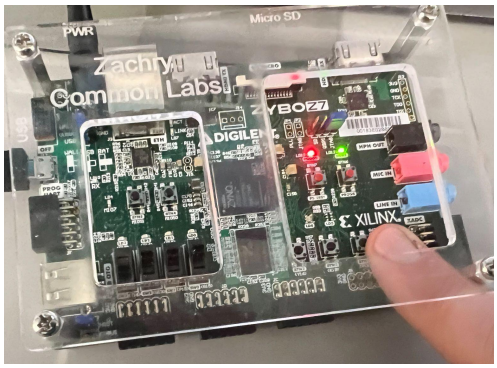
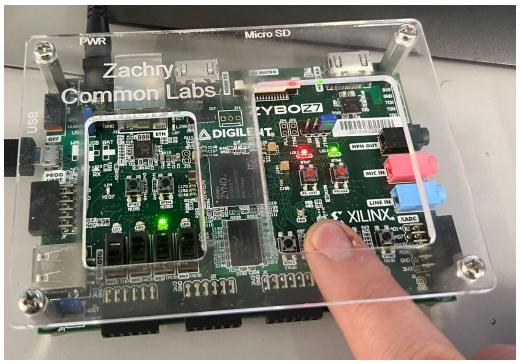
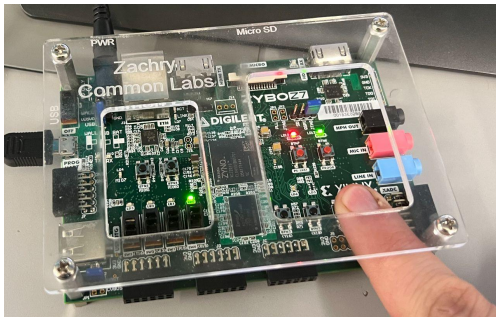
 Case 2: BTN3 = 0, BTN2 = 0, BTN1 = 1, BTN0 = 0, LED2 = 0, LED1 = 0, LED0 = 0	 Case 4: BTN3 = 0, BTN2 = 1, BTN1 = 0, BTN0 = 0, LED2 = 0, LED1 = 1, LED0 = 0
 Edge Case: BTN3 = 0, BTN2 = 0, BTN1 = 1, BTN0 = 1, LED2 = 0, LED1 = 0, LED0 = 1	

Table 3.1: Truth Table for Expected (from the lab manual) Results, Priority Encoder

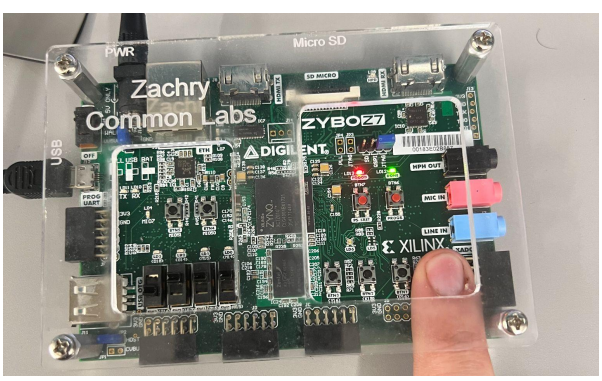
w_3	w_2	w_1	w_0	y_1	y_0	$zero$
0	0	0	0	X	X	1
0	0	0	1	0	0	0
0	0	1	X	0	1	0
0	1	X	X	1	0	0
1	X	X	X	1	1	0

Table 3.2: Truth Table for Actual (from the board) Results, Priority Encoder

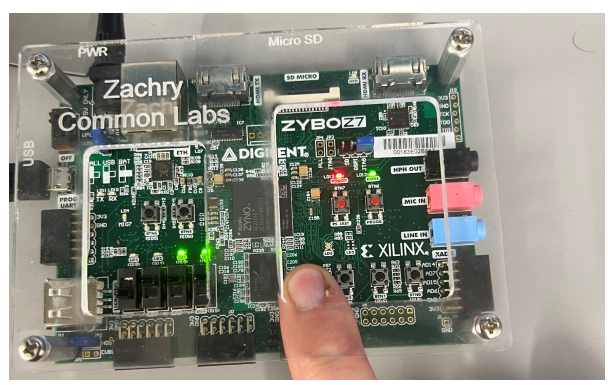
BTN3 (w_3)	BTN2 (w_2)	BTN1 (w_1)	BTN0 (w_0)	LED2 (Zero)	LED1 (y_1)	LED0 (y_0)
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	1	X	0	0	1
0	1	X	X	0	1	0
1	X	X	X	0	1	1

Shown above, the actual results for the priority encoder were identical to the expected from the lab manual. This includes the “don’t care” cases - it was observed that no matter how many buttons were pressed, the encoder prioritized the one that was pressed down first.

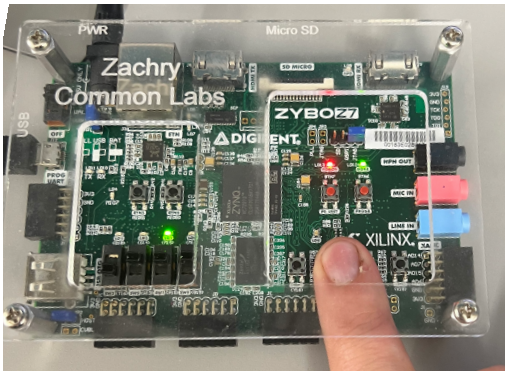
Provided below are pictures of two cases of the Zybo board’s results, with the inputs and outputs documented. Also listed is the extraneous case of holding down two buttons mentioned in the example above (holding Button 1 and Button 2 with Button 2 pressed first, LED1 stays lit).



Case 2: BTN3 = 0, BTN2 = 0, BTN1 = 0,



Case 5: BTN3 = 1, BTN2 = 0, BTN1 = 0,

BTN0 = 1, LED2 = 0, LED1 = 0, LED0 = 0	BTN0 = 0, LED2 = 0, LED1 = 1, LED0 = 1
 Edge Case: BTN3 = 0, BTN2 = 1, BTN1 = 1, BTN0 = 0, LED2 = 0, LED1 = 1, LED0 = 0 (Button BTN2 was pressed first).	

Conclusion:

In conclusion, this lab successfully introduced the knowledge of running and setting up Vivado to create a digital simulation using behavioral modeling. Behavioral modeling is a higher level of abstraction than the structural and data flow modeling we used in the last lab. After coding each of the modules in Vivado, each module was tested using the provided testbench files and debugged until each passed with no errors. Then, once all tests passed, the code was programmed on the Zybo boards and tested on real hardware.

In this lab, we learned how to code behavioral logic in Verilog and how to implement different levels of abstraction in Verilog code. Additionally, we learned how to correctly program the FPGA on the Zybo board using the modules that we coded in Vivado. With the Zybo FPGA board, it is possible to test your digital logic in seconds, as opposed to hours of wiring on a breadboard. This lab also illustrated the differences between a 4:2 Binary Encoder and a Priority Encoder, which is somewhat self-explanatory: The priority encoder prioritizes the first input and does not change when other erroneous inputs are introduced at the same time.

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. Provide a comparison between the behavioral Verilog used in this week's lab and the structural and dataflow Verilog used in last week's lab. What might be the advantages and disadvantages of each?

The lowest level of abstraction is Structural Verilog. The main advantage is that it is closer to plain English and easier to understand. For example, the words "and" and "or" are visible so you can clearly see which logic gates are used on each line of code. The drawback is that it is difficult to create complex circuits such as the ones used in this lab. This type of Verilog is best used for small, basic circuit designs with a limited number of gates.

The next level of abstraction is Dataflow Verilog. The advantage is that you can define the data flow of the circuit, instead of defining each gate one by one. This streamlines design as you have to think through the circuit you are coding and understand how the data flows through it. The disadvantage of using this type of Verilog is that you need to know the gate-level design of the circuit since we need to know which gates are in the circuit in order to figure out which operators are needed within a line of code.

The highest level of abstraction we have learned so far is Behavioral Verilog. The main advantage is you do not need gate-level design of the circuit, just the algorithm of the circuit - how it acts given a specific set of inputs. Another advantage is that you can code circuits using "Always" statements, "If Else" statements, Case statements, and various other types of statements, which are commonplace in software programming and familiar to those in the computer industry. The downside of using behavioral Verilog is the data flow and structure of the circuit are not as clearly defined as they are in the lower levels of abstraction, which means many comments are needed to explain what each line of code does.

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

Implementing a circuit on a breadboard is a lot of manual work, precise positioning, and constant checking/debugging. In comparison, implementing a circuit on an FPGA once the code is written is as simple as 2-3 clicks. While writing the code is not easy, depending on the nature and complexity of the project, each approach has its merits. The main advantage of using an FPGA is the ease of use and reproducibility, while the main disadvantage is integration, which is breadboarding's main advantage - an FPGA cannot simulate all different manners of specialized ICs (such as proprietary hardware) which a project might need. For this reason, I still prefer breadboard design as in my work I am usually designing systems based on an already existing board that I need to interface with and cannot simulate on an FPGA.

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 the LEDs on the board light up after I successfully programmed it was very satisfying.

Similar to the last lab, I least enjoyed the tedium of having to go through code and complete small parts. Also, the debugging help provided in the lab is minimal. There should really be more documentation provided other than "if you see something complete unsuccessfully, debug the error". I got especially stuck on this while trying to program by Zybo board.

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?

I had an error where my Zybo board could not be programmed as it was missing design constraints. It turned out I needed to set the 2:4 decoder (what I was programming) as top in the design section, not the simulation section. This distinction wasn't detailed in the lab manual and should be included.

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

I think we just need more assistance to ensure the lab is completed during the assigned lab time and doesn't have to bleed over into time outside of the lab, as it is very difficult to troubleshoot an error without the help of a TA present.