

Laboratory Exercise #7

Introduction to Behavioral Verilog and Logic Synthesis

ECEN 248: Introduction to Digital Design
Department of Electrical and Computer Engineering
Texas A&M University



1 Introduction

In the previous lab, we looked at two different approaches to describing digital circuits in Verilog HDL, namely structural and dataflow modeling. For this week's laboratory assignment, we will introduce an even higher level of abstraction available in Verilog HDL, commonly referred to as behavioral modeling. For the first lab experiment, you will recreate the multiplexers described in the last lab using behavioral Verilog. For the second experiment, you will use behavioral Verilog to describe the binary encoders and decoders talked about in lecture. In addition to behavioral modeling, this laboratory assignment will introduce logic synthesis, the process of translating HDL code into implementable digital logic. Experiment 3 will guide you through the process of synthesizing the decoders and encoders simulated in experiment 2. Furthermore, you will programming the Spartan 3E board on your workbench in order to test those components you described in Verilog.

2 Background

The subsection that follow will provide you with the background information necessary to complete the experiments in lab this week. Please read through this section and use it to complete the pre-lab assignment prior to your lab session.

2.1 Behavioral Verilog Modeling

Higher levels of abstraction within an HDL improve the productivity of a hardware developer by allowing him or her to simply describe the intended behavior of the digital circuit rather than the flow of digital data through logic gates. Compared to gate-level descriptions at the structural or dataflow level, behavioral modeling utilizes many of the constructs available in higher level programming languages to describe the algorithm the circuit designer would like to implement in hardware. One thing to keep in mind, however, is that most HDLs (including Verilog) are still concurrent languages, which means that behavioral statements you describe run in parallel. After all, it is hardware that you are describing!

Behavioral modeling in Verilog makes use of two very different structured procedure statements, **initial** and **always**. Both of these statements use the **begin** and **end** keywords to group behavioral statements into a single block. For example, code in between a **begin** and **end** pair immediately following an **initial** statement constitutes an **initial** block. Similarly, an **always** block includes the grouped code immediately following an **always** statement. An **initial** block executes only once starting at the beginning of simulation, whereas an **always** block repeats whenever a given trigger condition is met. The code below describes a 1-bit, 2:1 MUX using behavioral Verilog. Please take a moment to examine the code.

```

1 'timescale 1ns / 1ps
  'default_nettype none
3 /*This module describes a 1-bit wide multiplexer using behavioral constructs *
   *in Verilog HDL.*/
5
6 module two_one_mux(Y, A, B, S); //define the module name and its interface
7
8   /*declare output and input ports*/
9   output reg Y; //declare output of type reg since it will be modified in
   //an always block!
11  input wire A, B, S; //declare inputs of type wire
   //wires can drive regs in behavioral statements
13
14  always@(A or B or S) //always block which triggers whenever A, B, or S changes
15    begin //block constructs together
   /*1'b0 represents a 1-bit binary value of 0*/
17    if(S == 1'b0) //double equals are used for comparisons
      Y = A; //drive Y with A
19    else
      Y = B; //instead drive Y with B
21    end
23 endmodule //designate end of module

```

Behavioral modeling introduces a new type of net called a **reg**. Behavioral statements are not able to modify nets of type **wire** but can modify nets of type **reg**. In other words, you should never see a **wire** on the left-hand side of behavioral statement. However, regs can be driven by wires, which means you may see a **wire** on the right-hand side of a behavioral statement. Likewise, regs can drive wires within **assign** statements or other regs within behavioral statements. In the example above, the behavior of the multiplexer is easy to interpret. If the 1-bit wire, *S*, is equal to '0', then the output, *Y*, is driven by *A*. Otherwise, *Y* is driven by *B*.

2.2 Decoders and Encoders

Binary encoders and decoders can be used to transform the way digital data is represented. For example, a 2:4 binary decoder converts a 2-bit binary number into a 4-bit *one-hot* encoded output such that only one of the four output bits is active at one time. Table 1 illustrates the truth table for a 2:4 binary decoder with an active HIGH enable signal, *En*. Take a moment to examine the truth table and verify that the output signals are in fact *one-hot* encoded. The gate-level schematic for such a decoder is depicted in Figure 1. Examine the schematic and convince yourself that it implements the logic described in Table 1.

For an example use case of a decoder, consider a situation in which a device has four LEDs which indicate mutually exclusive events. In this circumstance, no more than one LED should be light at any point in time. We can then use a 2-bit binary number to indicate which LED should be light and feed that binary

Table 1: 2:4 Binary Decoder Truth Table

En	w_1	w_0	y_3	y_2	y_1	y_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
0	X	X	0	0	0	0

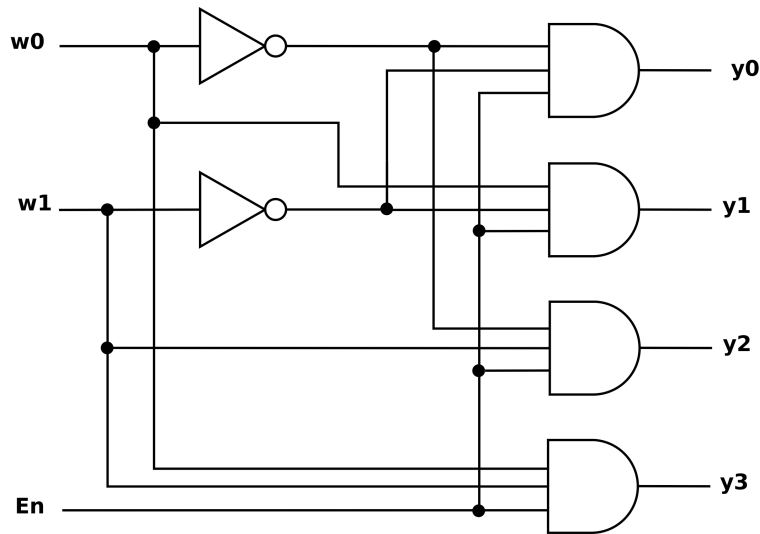


Figure 1: 2:4 Binary Decoder Gate-level Schematic

number into a decoder to power four LEDs. Now that we understand what a binary decoder does, we can surmise what a binary encoder might do. As you may have figured, a binary encoder will transform a *one-hot* encoding back into a binary number. The truth table for a 4:2 binary encoder can be found in Table 2. Notice that unexpected inputs where more than one input signal is HIGH are not shown. Also, a *zero* signal has been provided which indicates when no input signal is HIGH. The corresponding gate-level schematic is shown in Figure 2.

A binary encoder similar to the one discussed above could be used to encode buttons on a keypad assuming only one button is expected to be pressed at any point in time. For example, if we have four buttons as input, the 4:2 binary encoder would convert the four digital signals coming from the buttons into a 2-bit binary number representing which button is being pressed. The *zero* signal would indicate that no buttons are being pressed when asserted.

Table 2: 4:2 Binary Encoder Truth Table

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

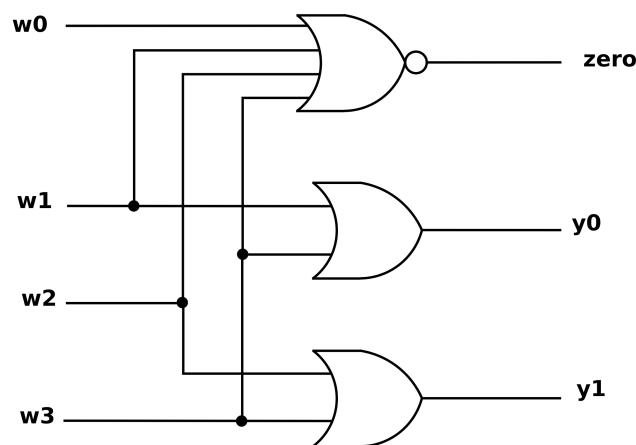


Figure 2: 4:2 Binary Encoder Gate-level Schematic

Table 3: 4:2 Priority Encoder Truth Table

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

This type of encoder works well when no more than one button is pressed at a time, but what happens when that is not the case? Figure 2 was drawn assuming those inputs would not exist; however, we can use it to determine exactly what the output would be in all 16 button combinations. A pre-lab exercise will ask you to do just that! If we must also handle these extraneous cases, we may want to utilize a priority encoder. Table 3 shows a truth table of a priority encoder which assigns priorities to each input bit. Input bits to the left have a higher priority than input bits to the right.

2.3 FPGAs and Logic Synthesis

Logic simulation greatly simplifies the design process by allowing a designer to simulate the operation of his or her HDL code without the need for bread-boarding the digital circuit. Simulation works well until the design phase is far enough along that your circuit needs to interact with other pre-build components within a larger system, or your customer is demanding an early demonstration of your product to ensure their investment is sound. At that point, what is the next step? Well one very good solution would be to employ a Field Programmable Gate Array (FPGA), which is apart of a larger class of re-programmable logic devices.

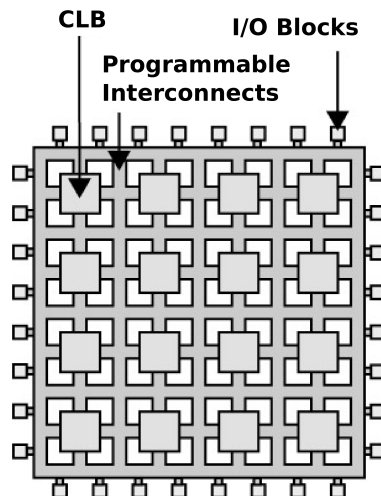


Figure 3: Field Programmable Gate Array

An FPGA contains an array of Configurable Logic Blocks (CLBs) surrounded by programmable interconnects. The hierarchy of programmable interconnects allows logic blocks to be interconnected as needed, somewhat like an on-chip programmable breadboard. In addition to re-programmable logic, FPGAs typically contain hardware macros such as memory blocks, arithmetic units, and even entire microprocessors. The re-programmability of the FPGA means that it can be programmed in the field (i.e. after being purchased). The FPGA that we will use in lab is a Spartan 3E FPGA manufactured by Xilinx. Consequently, we will use Xilinx ISE to perform logic synthesis, which will convert our HDL code into a low level netlist of FPGA primitives (i.e. CLBs). Once synthesis is complete, we will move into the implementation phase of the design, which will place the primitives in the netlist onto the FPGA and route the necessary interconnects. The output of the implementation phase is a bit stream file which we will use to program the FPGA.

Before moving on, it is important for us to talk about a subset of Verilog known as synthesizable Verilog. Not all of the Verilog constructs you have seen so far are implementable. For example, **initial** blocks cannot

be synthesized. Likewise, delays¹ in Verilog cannot be synthesized. Thus, we consider all constructs in Verilog, which can be synthesized, to be synthesizable Verilog. Constructs that are not apart of this subset are still very useful for creating elaborate test benches for use during simulation.

2.4 The Xilinx Spartan 3E Board

An FPGA by itself is not particularly useful because it must be soldered onto a circuit board, which will provide adequate power and ground in addition to a steady clock signal. For prototyping², Xilinx provides various evaluation boards, which contain an FPGA and a handful of other support components including switches, push-buttons, LEDs, and Liquid Crystal Displays (LCDs) for I/O. The evaluation board we will be using in lab is the Spartan 3E Board shown in Figure 4. Take a moment to locate all of the aforementioned components in the pictured provided.

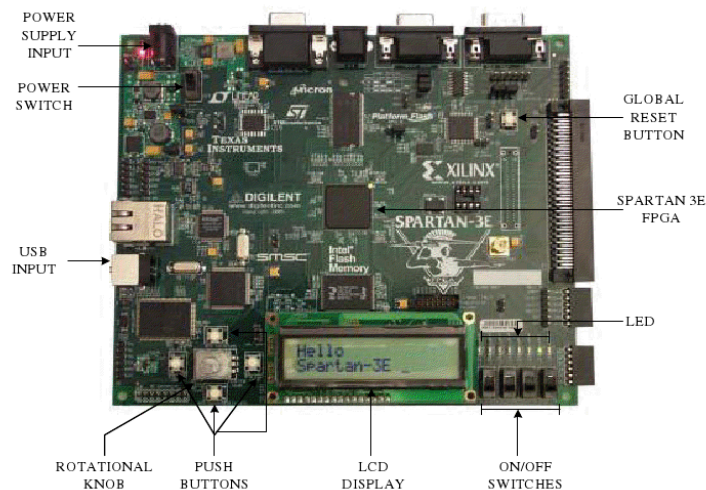


Figure 4: The Spartan 3E Board

3 Pre-lab

The intention of the pre-lab is to prepare you for the upcoming lab assignment. Please complete the pre-lab prior to attending your lab session.

¹We will discuss delays in future labs.

²A prototype is a model built to test a concept. Prototyping refers to the creation of such a model and is done quite often in industry sometimes prior to even finding a customer.

3.1 Binary Decoder and Encoder

In this week's lab assignment, we would like to compare the use of behavioral modeling with that of structural and dataflow modeling. To do so, you will be asked to come to lab with a Verilog model of the 2:4 binary decoder, the 4:2 binary encoder, and the 4:2 priority encoder. For the first two on the list, the background section explains all you need to know to describe them in Verilog. Use the module interfaces found below to get started. You will need to describe one of them (it does not matter which one) using structural Verilog and the built-in gate-level primitives, while for the other one, you will need to use gate-level dataflow Verilog. If these terms do not make sense, please consult the previous lab assignments. Be sure to comment your code thoroughly!

```

1  /*module interface for the 2:4 decoder*/
   module two_four_decoder(
3    input wire  [1:0] W,
      input wire  En,
5    output wire [3:0] Y
   );
7
   /*module interface for the 4:2 encoder*/
9  module four_two_encoder(
      input wire [3:0] W,
11   output wire [1:0] Y,
      output wire zero
13 );

```

3.2 Priority Encoder

For the priority encoder, it is more convenient to describe a set of intermediate signals which essentially set the priority. Those signals can then be feed directly into a simple binary encoder discussed above. Figure 5 illustrates this concept. The truth table for the priority encoder modified to include these intermediate signals is shown in Table 4. The boolean algebra expressions for the intermediate signals are as follows:

$$i_0 = \overline{w_3}\overline{w_2}\overline{w_1}w_0$$

$$i_1 = \overline{w_3}\overline{w_2}w_1$$

$$i_2 = \overline{w_3}w_2$$

$$i_3 = w_3$$

Using the above expressions as intermediate signals, describe the priority encoder in Verilog. You may use either structural or dataflow Verilog and the following module interface:


```

1 module priority_encoder(
    input wire [3:0] W,
3   output wire [1:0] Y,
    output wire zero
5   );

```

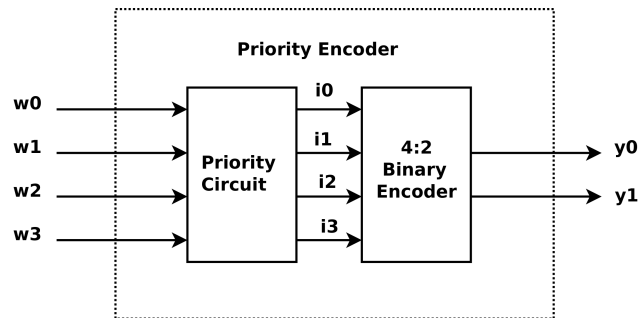


Figure 5: Priority Encoder

Table 4: 4:2 Priority Encoder Truth Table

w_3	w_2	w_1	w_0	i_3	i_2	i_1	i_0	y_1	y_0	$zero$
0	0	0	0	0	0	0	0	X	X	1
0	0	0	1	0	0	0	1	0	0	0
0	0	1	X	0	0	1	0	0	1	0
0	1	X	X	0	1	0	0	1	0	0
1	X	X	X	1	0	0	0	1	1	0

3.3 Pre-lab Deliverables

Please include the following items in your pre-lab write-up in addition to the items mentioned in the *Policies and Procedures* document.

1. Verilog code with comments for the 2:4 binary decoder, the 4:2 binary encoder, and the 4:2 priority encoder. Do not use *behavioral* Verilog for these descriptions! Use the structural and dataflow concepts introduced in the previous lab.
2. The complete truth table for the gate-level schematic shown in Figure 2. This truth table should not include “don’t cares” (i.e. ‘X’)!

3. A brief comparison of the the behavioral implementation of a multiplexer described in the background section with the multiplexer you described in the previous lab using structural and dataflow.

4 Lab Procedure

For the following laboratory experiments, you will be expected to complete procedures that were introduced in the previous lab such as creating a new ISE project, creating a new source file, etc. Please reference the previous lab manual for step-by-step guidance if you do not remember how to perform an action listed below.

4.1 Experiment 1

For this experiment, we would like to get a feel for describing hardware in behavioral Verilog. To do so, we will start by simulating the multiplexer provided to you in the background section. We will then ask you to extend that multiplexer to a 4-bit wide, 2:1 multiplexer and finally to a 4-bit wide, 4:1 multiplexer. The steps below will guide you through the entire process.

1. Describe a 1-bit, 2:1 multiplexer in behavioral Verilog and simulate its operation.
 - (a) Begin by opening ISE and creating a new project called “lab7”. We will use this same project throughout the lab assignment.
 - (b) Create a new source file and save it as “two_one_mux_behavioral.v” in your lab7 directory.
 - (c) Type the behavioral code provided in the background section of this manual into the file you just created. Save the file and then add it to your new ISE project.
 - (d) Copy the “two_one_mux_tb.v” file from the course directory into your lab7 directory and add it to your ISE project.
 - (e) Simulate the test bench and ensure the UUT passes all of the tests. Include a screenshot of the simulation waveform and console output in your lab write-up.
2. Describe a 4-bit, 2:1 multiplexer in behavioral Verilog and simulate its operation.
 - (a) A 4-bit, 2:1 multiplexer can be easily described by simply making the output port, Y , and the input ports, A and B , 4-bits wide. Save the “two_one_mux_behavioral.v” file as “four_bit_mux_behavioral.v” and replace lines 5 through 11 with the following Verilog code:

```

1 module four_bit_mux(Y, A, B, S);
2
3     /*declare output and input ports*/
4     output reg [3:0] Y; //output is a 4-bit wide reg
5     input wire [3:0] A, B; //A and B are 4-bit wide wires
6     input wire S; //select is still 1 bit wide
7
8
9
10
11

```

- (b) Add the new source file to your ISE project. Copy the “four_bit_mux_tb.v” file from the course directory into your lab7 directory, and add it to your ISE project as well.
 - (c) Simulate the test bench and ensure the UUT passes all of the tests. Include a screenshot of the simulation waveform and console output in your lab write-up.
3. The **if-else** clause in the 2:1 multiplexer code works quite well; however, the **case** statement can be more succinct when the number of inputs is greater than two. Describe a 4-bit, 4:1 multiplexer in behavioral Verilog and simulate its operation.
- (a) Create a new Verilog source file, “mux_4bit_4to1.v”, and use the following Verilog code as a template to describe the multiplexer.

```

1  'timescale 1ns / 1ps
2  'default_nettype none
   /* This module describes a 4-bit, 4:1 multiplexer using behavioral constructs */
4  /* in Verilog HDL. */
   module mux_4bit_4to1(Y, A, B, C, D, S);
6
   /* declare output and input ports */
8   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
10  // select is a 2-bit wire

12  always@(*) // new Verilog trick!!! * means trigger when anything changes
   // notice that we did not use begin and end because the case
   // statement is considered one clause
   case(S) // selection based on S
16     2'b00: Y = A; // when S == 2'b00
        // fill in something here...
18     2'b11: Y = D; // when S == 2'b11
   endcase // designates the end of a case statement
20
endmodule

```

- (b) Save the source file and add it to your lab7 project. Similarly, copy the test bench file, “mux_4bit_4to1_tb.v”, into your lab7 directory and add it to your lab7 project.
- (c) Simulate the test bench and ensure the UUT passes all of the tests. Include a screenshot of the simulation waveform and console output in your lab write-up.

4.2 Experiment 2

For the next experiment, we would like to give you exposure to binary encoders and decoders discussed in lecture, while reinforcing the behavioral Verilog concept.

1. Use behavioral Verilog to describe the 2:4 binary decoder and the 4:2 binary encoder. Simulate the **two_four_decoder** and **four_two_encoder** modules you design in ISE using the “two_four_decoder_tb.v” and “four_two_encoder_tb.v” test bench files, respectively.

- (a) Create a new source file called “two_four_decoder.v” and describe the 2:4 binary decoder in behavioral Verilog using the code below as a starting point.

```

1  `timescale 1ns / 1ps
   `default_nettype none
3  /*This module describes a 2:4 decoder using behavioral constructs in Verilog HDL. */

5  /*module interface for the 2:4 decoder*/
   module two_four_decoder(
7      input wire [1:0] W,
      input wire En,
9      //Y should be a 4-bit output of type reg
   );
11
      always@( ) //something is missing here... trigger when En or W changes
13      begin //not necessary because if is single clause but looks better
          if (En == 1'b1) //can put case within if clause!
15              case(W) //selection based on W
                  2'b00: Y = 4'b0001; //4'b signifies a 4-bit binary value
17                  //fill in code here...
                  2'b11: Y = 4'b1000; //light up y[3]
19              endcase //designates the end of a case statement
          else //if not Enable
21              Y = 4'b0000; //disable all outputs
      end
23
   endmodule

```

- (b) Add the source file you just created to your ISE project and simulate the 2:4 binary decoder behavioral model with the 2:4 decoder test bench used to simulate the decoder you described in the pre-lab. Ensure the UUT passes all of the tests and include a screenshot of the simulation waveform and console output in your lab write-up.
- (c) Create a new source file, “four_two_encoder.v”, and use the code below to describe a the 4:2 binary encoder in behavioral Verilog.

```

   `timescale 1ns / 1ps
2  `default_nettype none
   /*This module describes a 2:4 decoder using behavioral constructs in Verilog HDL. */
4
   /*module interface for the 4:2 encoder*/
6  module four_two_encoder(
      input wire [3:0] W,
8      output wire zero,
      //Y should be a 2-bit output of type reg

```

```

10 );

12     /*can mix levels of abstraction!*/
    assign zero = (W == 4'b0000); //a zero test! notice the use of == rather than =
14
    /*behavioral portion*/
16     always@( ) //something is missing here... trigger when W changes
        begin //not necessary because case is single clause but looks better
18             case(W) //selection based on W
                4'b0001: Y = 2'b00; //2'b signifies a 2-bit binary value
20                4'b0010: Y = 2'b01; //w[1] is lit up
                //fill in the case where only w[2] is lit up
22                4'b1000: Y = 2'b11; //w[3] is lit up
                default: Y = 2'bXX; //default covers cases not listed!
24                //2'bXX means 2-bits of don't cares!
            endcase //designates the end of a case statement
26        end

28 endmodule

```

- (d) Simulate the 4:2 binary encoder behavioral model with the appropriate test bench. Ensure the UUT passes all of the tests and include a screenshot of the simulation waveform and console output in your lab write-up.

2. Describe a Verilog model for the priority encoder discussed in the background section of this manual.

- (a) Compare the **case** statement for the 2:4 binary encoder with Table 2. Notice the similarity between the **case** statement and the truth table of the encoder. Essentially, the **case** statement allows the designer to describe a circuit's truth table directly in Verilog. Furthermore, we can use a **casex** to include don't cares on the left side of a truth table. These constructs make it easy to describe a priority encoder. Use the code below as a starting point. Save the source file as "priority_encoder.v" and simulate it with the "priority_encoder_tb.v" test bench.

```

    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
        //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

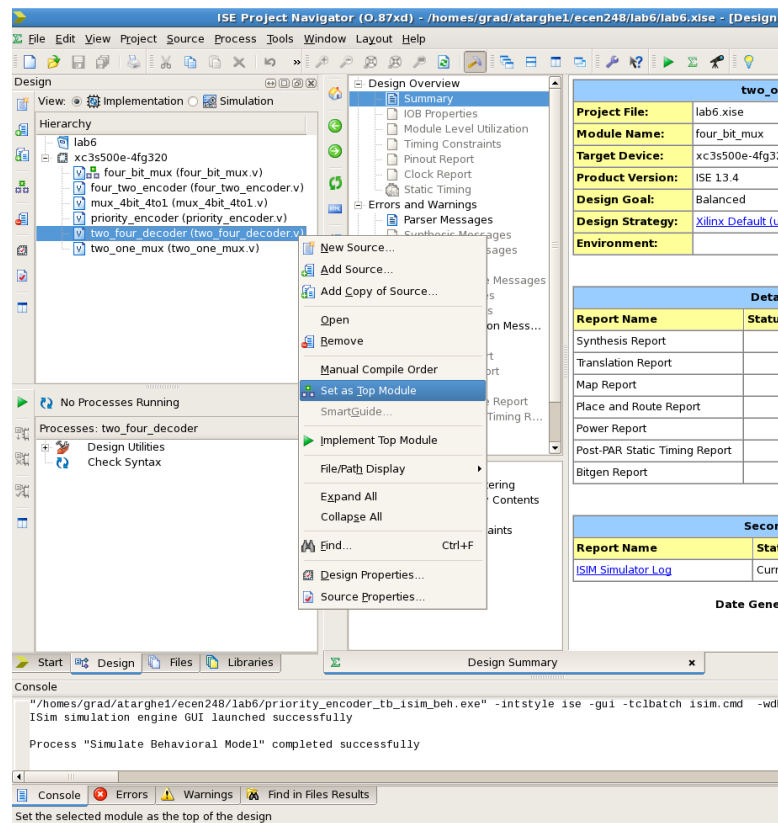
```

4.3 Experiment 3

For the final experiment, we will actually put the designs we created in the previous experiment onto the Spartan 3E board. We will make use of the switches, push-buttons, and LEDs built into the evaluation board

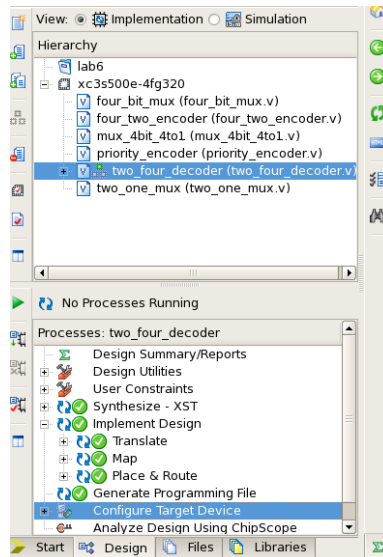
to verify the functionality of those components. Please note that we have already performed a behavioral simulation on those modules so we should have a high level of confidence that they will work properly.

1. Synthesize and Implement the **two_four_decoder** module. Then, program the Spartan 3E board with the appropriate bit stream.
 - (a) From the Design window, change the view from “Simulation” to “Implementation.”
 - (b) Set the **two_four_decoder** module as the “Top Module” by right-clicking the module name and selecting “Set As Top Module.” See screenshot below.

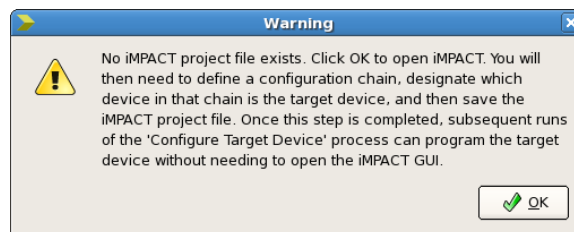


- (c) Copy the “two_four_decoder.ucf” file from the course directory into your lab7 directory and add it to your ISE project. This file is the User Constraint File (UCF), which the Xilinx tools use to connect the ports of your design to the pins on the FPGA. The information contained within the UCF was taken from the documentation that comes with the FPGA board.
- (d) Double-click on “Generate Programming File” in the Processes window within ISE.
- (e) Once complete, you should see green checks next to “Implement Design” and “Generate Programming File” within the Process window as seen below. If your screen has red X’s, then an

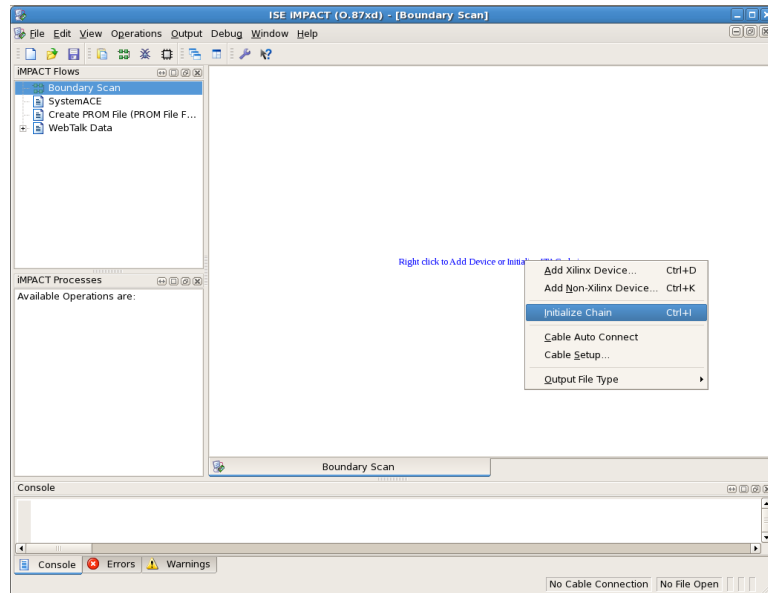
error has occurred. Use the console window to determine what error occurred and correct your design accordingly.



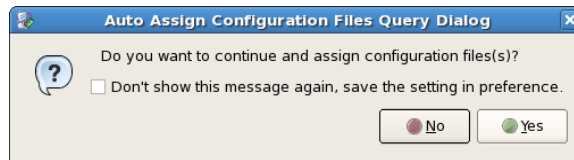
- (f) Up to this point, you have successfully synthesized and implemented your design. The programming file has also been created. Turn the FPGA board on by flipping the power switch shown in Figure 4. A red LED in the top-left corner of the board should illuminate indicating the board is powered on. Once you see the LED light up, Double-click on “Configure Target Device” to begin the programming phase.
- (g) Immediately after selecting “Configure Target Device” a warning box will appear (shown below) stating that no iMPACT project file exists. That is just fine so select “OK” in the dialog box.



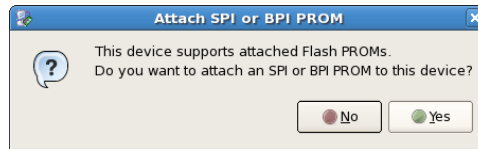
- (h) iMPACT, the ISE software tool that we will use to program our FPGA, will open up. Double-click on “Boundary Scan” and right-click on the blue lettering in the center of the screen. Then select “Initialize Chain.” See screenshot below.



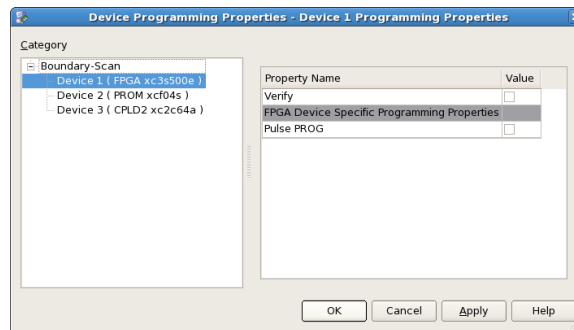
- (i) Reprogrammable devices can be chained together using a common communication standard known as JTAG. An explanation of JTAG is far beyond the scope of this lab; however you should know that the chain that is being initialized is that of JTAG. iMPACT will now scan for devices on JTAG chain. There are three devices connected to the JTAG chain. After the JTAG chain has been scanned, a dialog box will appear asking if we would like to assign configuration files to the devices on the chain (see below). We would like to do just that so select ‘Yes.’



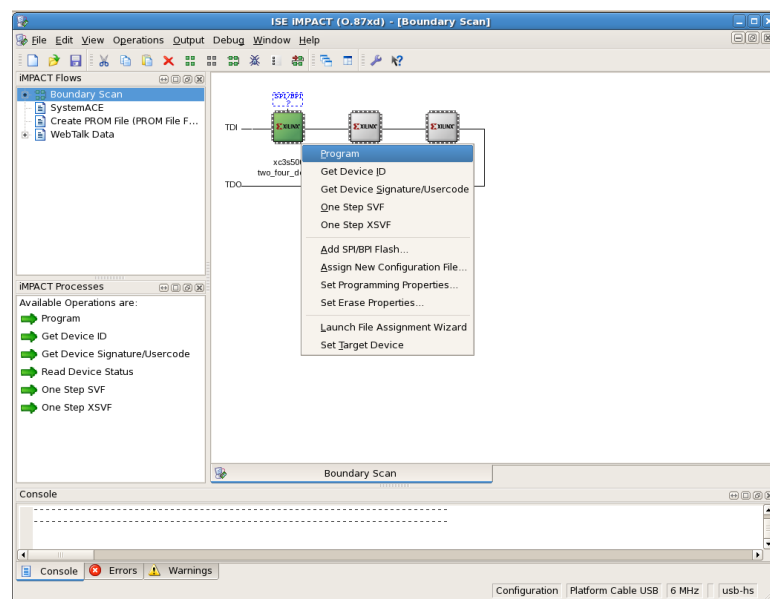
- (j) The FPGA is not the only device that can be programmed on the Spartan 3E board; however, the Spartan 3E FPGA is all we will program in this lab. The first device on the chain will be the FPGA. A browser window will appear such that you can select a configuration file for the FPGA. Select “two_four_decode.bit,” which was generated during the implementation phase earlier. Then hit “Open” to proceed.
- (k) After selecting the configuration file for the FPGA, a dialog box shown below will appear asking if we would like to attach an SPI or BPI PROM to the FPGA. These are advance features available on the FPGA board so we will not use them. Select “No” to continue.



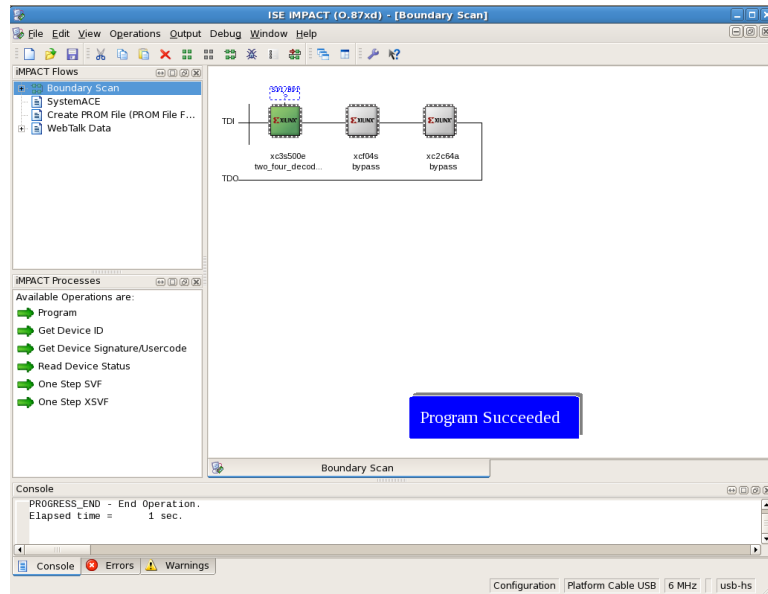
- (l) For the next two devices, select “Bypass” when prompted to select a programming file. We will not program these devices in this lab.
- (m) Finally, a “Device Programming Properties” box will appear. We plan to leave all settings as default so hit “OK” to proceed.



- (n) Now within the iMPACT window, right-click on the FPGA part and select “Program” (see below).



- (o) If the FPGA was programmed successfully, you should see “Program Succeeded” at the bottom of the iMPACT window as seen below.



2. If you have reached this point in the lab manual, you have successfully program an FPGA! Now it is time to see if our simple 2:4 binary decoder works!

- (a) Open the UCF and examine the contents. Notice that the **En** is mapped to switch 2, while bits 1 and 0 of **W** are mapped to switch 1 and switch 0, respectively. The output bus, **Y**, has been mapped to four of the LEDs.
- (b) Flip these switches to change the status of the LEDs. Try all possible input combinations and ensure the design is working properly. Create a truth table with **SW2**, **SW1**, and **SW0** as inputs and **LED3**, **LED2**, **LED1**, and **LED0** as outputs. Demonstrate your progress to the TA once you have found the design to work.

Note: Leave iMPACT open as we will use it for the remainder of the lab assignment.

3. Now program the FPGA with 4:2 binary encoder you simulated earlier.

- (a) Repeat steps (a) through (e) above with the **four_two_encoder** using the “four_two_encoder.ucf” file in the course directory.
- (b) Instead of selecting “Configure Target Device” in ISE, return to iMPACT and right-click on the FPGA (xc3s500e). Select “Assign New Configuration File...” Then choose “four_two_encoder.bit.”
- (c) Now repeat steps (k) through (o) to finish programming the board.

- (d) For the encoder, the inputs have been mapped to the push-buttons. Open up the corresponding UCF to verify this. Press the push-buttons on the Spartan 3E board and note that the LEDs display a binary code based on the particular button being pressed. Likewise, ensure the *zero* signal is working properly.
 - (e) Now press more than one button and note what is display on the LEDs.
4. Finally, use the above steps to load the **priority_encoder** onto the FPGA using the “priority_encoder.ucf” file. The same button mapping exists for the priority encoder as the binary encoder so that you can compare the two. Test out the priority encoder as you did the binary encoder and note the difference. Demonstrate your progress to the TA once you have found the design to work.

5 Post-lab Deliverables

Please include the following items in your post-lab write-up in addition to the deliverables mentioned in the *Policies and Procedures* document.

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!
2. Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.
3. Provide a comparison between 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.
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?

6 Important Student Feedback

The last part of lab requests your feedback. We are continually trying to improve the laboratory exercises to enhance your learning experience, and we are unable to do so without your feedback. Please include the following post-lab deliverables in your lab write-up.

Note: If you have any other comments regarding the lab that you wish to bring to your instructor’s attention, please feel free to include them as well.

1. What did you like most about the lab assignment and why? What did you like least about it and why?
2. Were there any section of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?
3. What suggestions do you have to improve the overall lab assignment?