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

Laboratory Exercise #6
Introduction to Behavioral Verilog and Logic Synthesis

Lab exercise created and tested by
Abbas Fairouz, He Zhou, Joshua Mashburn, and Sunil P. Khatri

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 ZYBO Z7-10 board on your workbench in order to test those components you described in Verilog.

2 Background

The subsection that follows 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.

```
`timescale 1ns / 1ps
`default_nettype none
/*This module describes a 1-bit wide multiplexer using behavioral constructs *
 *in Verilog HDL.                                                            */

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

    /*declare output and input ports*/
    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;//drive Y with A
        else
            Y = B;//instead drive Y with B
      end

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

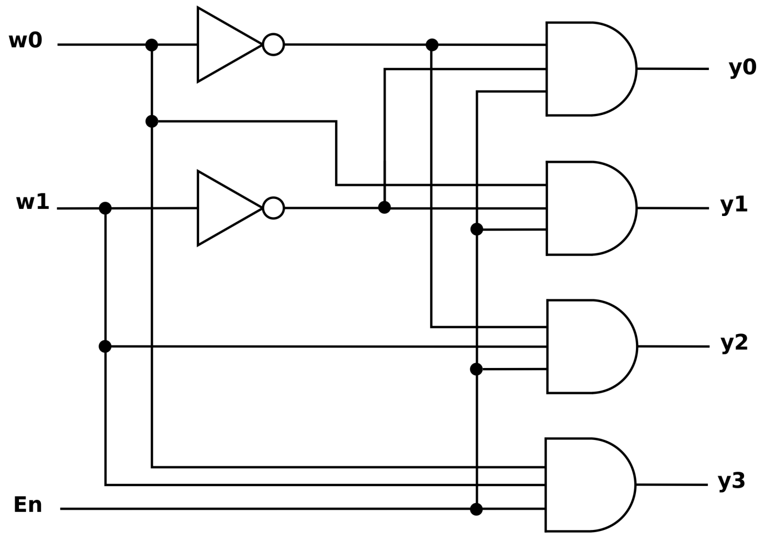


Figure 1: 2:4 Binary Decoder Gate-level Schematic

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

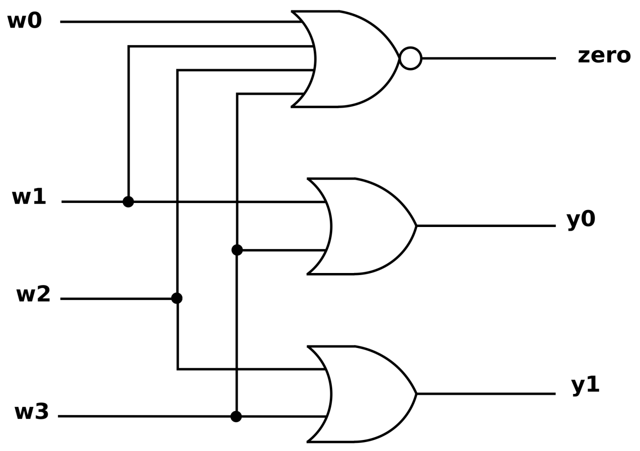


Figure 2: 4:2 Binary Encoder Gate-level Schematic

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-built 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 a part of a larger class of reprogrammable logic devices.

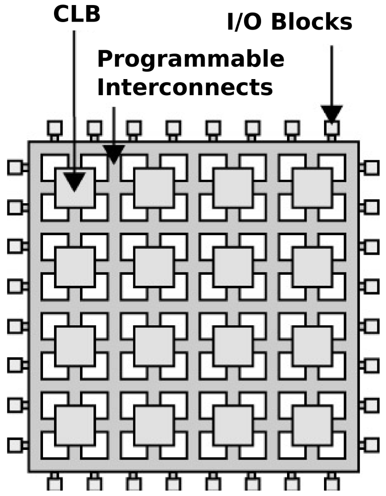


Figure 3: Field Programmable Gate Array

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 Zynq-7000 FPGA manufactured by Xilinx. The ZYBO Z7-10 board manufactured by Digilent is a ready-to-use, entry-level embedded software and digital circuit development board built around the Zynq-7000. Consequently, we will use Xilinx Vivado 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 bitstream 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, delays[1] in Verilog cannot be synthesized. Thus, all constructs in Verilog which can be synthesized in hardware, we call synthesizable Verilog. Constructs that are not a part of this subset are still very useful for creating elaborate testbenches for use during simulation.

2.4 The ZYBO Z7-10

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 prototyping[2], Xilinx provides various boards, which contain an FPGA and a handful of other support components including switches, push-buttons, LEDs for I/O. The board we will be using in lab is the ZYBO Z7-10 Board shown in Figure 4. Take a moment to locate all of the aforementioned components in the picture provided.

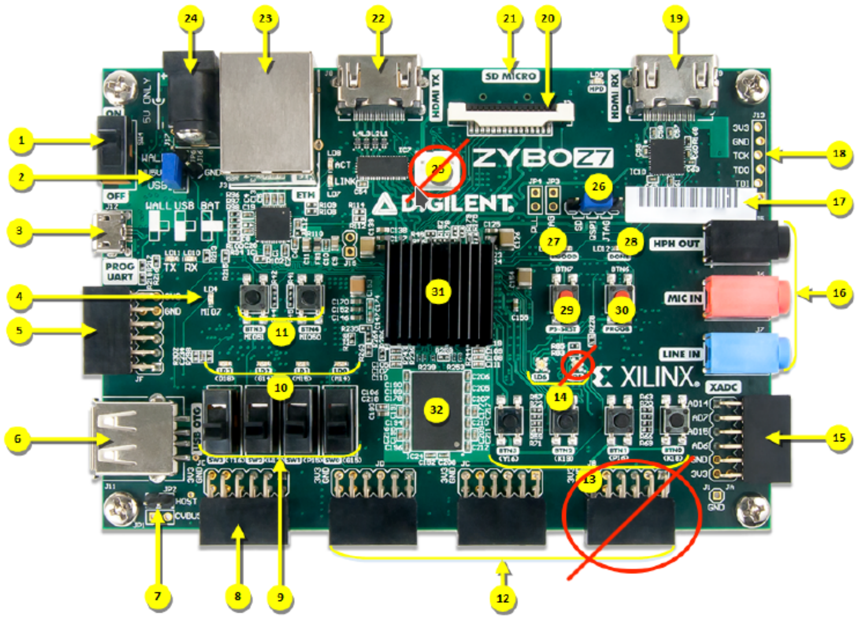


Figure 4: ZYBO board (Z7-20 shown, only minor differences from the Z7-10 we will use)

[1]We will discuss delays in future labs.

[2]A 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.

Callout	*Description*	*Callout*	*Description*	*Callout*	*Description*
1	Power Switch	12	High-speed Pmod ports	23	Ethernet port
2	Power select jumper	13	User buttons	24	External power supply connector
3	USB JTAG/UART port	14	User RGB LEDs	25	Fan connector (5V, three-wire)*
4	MIO User LED	15	XADC Pmod port	26	Programming mode select jumper
5	MIO Pmod port	16	Audio codec ports	27	Power supply good LED
6	USB 2.0 Host/OTG port	17	Unique MAC address label	28	FPGA programming done LED
7	USB Host power enable jumper	18	External JTAG port	29	Processor reset button
8	Standard Pmod port	19	HDMI input port	30	FPGA clear configuration button
9	User switches	20	Pcam MIPI CSI-2 port	31	Zynq-7000
10	User LEDs	21	microSD connector (other side)	32	DDR3L Memory
11	MIO User buttons	22	HDMI output port		* not included on Z7-10

Table 4: Zybo Z7 table callout (Figure 4) - *Zybo Z7-20 pictured

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.

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!

```
/*module interface for the 2:4 decoder*/
module two_four_decoder(
  input wire [1:0] W,
  input wire En,
  output wire [3:0] Y
);

/*module interface for the 4:2 encoder*/
module four_two_encoder(
  input wire [3:0] W,
  output wire [1:0] Y,
  output wire zero
);
```

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 be fed directly into the simple binary encoder discussed above. Figure 5 illustrates this concept. The truth table for the priority encoder that includes these intermediate signals is shown in Table 5. The boolean algebra expressions for the intermediate signals are as follows:

$$\begin{aligned} i_0 &= \overline{w}_3\overline{w}_2\overline{w}_1 w_0 \\ i_1 &= \overline{w}_3\overline{w}_2 w_1 \\ i_2 &= \overline{w}_3 w_2 \\ i_3 &= w_3 \end{aligned}$$

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:

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

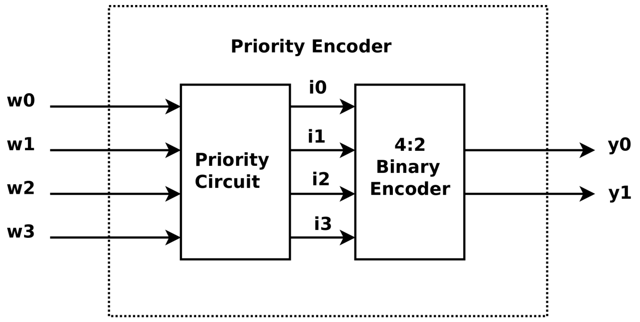


Figure 5: Priority Encoder

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.

Table 5: 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

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') as outputs!

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 Vivado 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 Part 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 Vivado and creating a new project called 'lab6'. 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 lab6 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 Vivado project.

(d) Copy the ‘two_one_mux_tb.v’ file from the course directory into your lab6 directory and add it to your Vivado 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 report.

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 5-11 with the following Verilog code:

```
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
```

(b) Add the new source file to your Vivado project. Copy the ‘four_bit_mux_tb.v’ file from the course directory into your lab6 directory, and add it to your Vivado 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 report.

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.

```
`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
    //select is a 2-bit wire

    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
            2'b00: Y = A; //when S == 2'b00
```

```
                //fill in something here...
                2'b11: Y = D; //when S == 2'b11
            endcase //designates the end of a case statement

endmodule
```

(b) Save the source file and add it to your lab6 project. Similarly, copy the test bench file, 'mux_4bit_4to1_tb.v', into your lab6 directory and add it to your lab6 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 report.

4.2 Experiment Part 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:

   (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.

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

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

    always@( ) //something is missing here... trigger when En or W changes
      begin //not necessary because if is single clause but looks better
        if(En == 1'b1) //can put case within if clause!
            case(W) //selection based on W
                2'b00: Y = 4'b0001; //4'b signifies a 4-bit binary value
                //fill in code here...
                2'b11: Y = 4'b1000; //light up y[3]
            endcase //designates the end of a case statement
        else //if not Enable
            Y = 4'b0000; //disable all outputs
      end

endmodule
```

(b) Add the source file you just created to your Vivado project and simulate the 2:4 binary decoder behavioral model using 'two_four_decoder_tb.v'. Ensure the UUT passes all of the tests and include a screenshot of the simulation waveform and console output in your lab report.

(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
`default_nettype none
/*This module describes a 2:4 decoder using behavioral constructs in Verilog HDL. */

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

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

    /*behavioral portion*/
    always@( ) //something is missing here... trigger when W changes
      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
            //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!
                                //2'bXX means 2-bits of don't cares!
        endcase //designates the end of a case statement
      end

endmodule
```

(d) Simulate the 4:2 binary encoder behavioral model using 'four_two_encoder_tb.v'. Ensure the UUT passes all of the tests and include a screenshot of the simulation waveform and console output in your lab report.

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 **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.

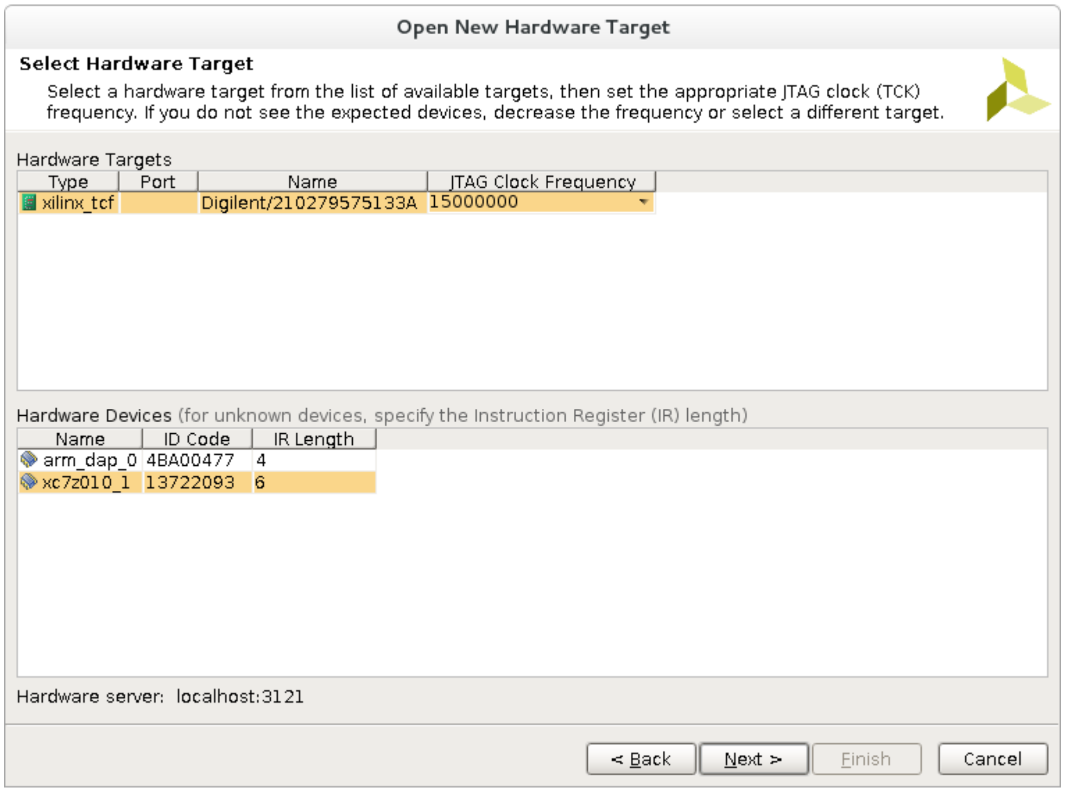
```
casex (W)
    4'b0001: Y = 2'b00; //2'b signifies a 2-bit binary value
    4'b001X: Y = 2'b01; //w[1] is lit up
    //fill in the case where only w[2] is lit up
    4'b1XXX: Y = 2'b11; //w[3] is lit up
    default: Y = 2'bXX; //default covers cases not listed!
endcase
```

4.3 Experiment Part 3

For the final experiment, we will actually put the designs we created in the previous experiment onto the ZYBO Z7-10 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 ZYBO Z7-10 board with the appropriate bit stream.
   (a) Set the **two_four_decoder** module as the 'Top Module' by right-clicking the module name under Design Sources and selecting 'Set As Top' as shown earlier.
   (b) Copy the 'two_four_decoder.xdc' file from the course directory into your lab6 directory. To add it to your Vivado project, follow the same procedure you did to add the source file in the previous lab, except select 'Add or Create Design Constraints' in the Add Sources window. This file is the Xilinx Design Constraints (XDC), which the Xilinx tools use to connect the ports of your design to the pins on the FPGA. The information contained within the XDC file was taken from the documentation that comes with the FPGA board.
   (c) Click on 'Generate Bitstream File' in the Flow Navigator panel within Vivado.
   (d) Once successfully completed, a pop up window will appear indicating 'Bitstream Generation Completed'. Select 'Open Hardware Manager' and press 'OK'. If unsuccessful, then an error has occurred. Use the Console and Messages panels to determine what error occurred and correct your design accordingly.
   (e) Up to this point, you have successfully synthesized and implemented your design. The programming file has also been created. Make sure JP5 on board is set to use 'JTAG' mode for programming. Turn the FPGA board on by flipping the power switch shown in Figure 4. A red LED in the center of the board should illuminate indicating the board is powered on. Once you see the LED light up, click on 'Open New Target'.
   (f) Select 'Open New Target' and an 'Open Hardware Target' window will open. Click 'Next'. In the next window, select 'Local Server' in the Connect to field and click 'Next'. Select 'Digilent'

in Hardware Targets and xc7z010_1 in Hardware Devices as shown in figure below. Click ‘Next’ and select ‘Finish’ to connect to the hardware.



Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

Type	Port	Name	JTAG Clock Frequency
xilinx_tcf		Digilent/210279575133A	15000000

Hardware Devices (for unknown devices, specify the Instruction Register (IR) length)

Name	ID Code	IR Length
arm_dap_0	4BA00477	4
xc7z010_1	13722093	6

Hardware server: localhost:3121

(g) Click on ‘Program Device’ under Hardware Manager and click on xc7z010_1. A window will appear displaying the information about the bitstream and debug files. Leave the default locations and click Program to program the board.

(h) Once programming is done, a green light will appear indicating that the device is programmed successfully.

(i) The FPGA is not the only device that can be programmed on the ZYBO Z7-10 board; however, the Zynq 7000 FPGA is all we will program in this lab, so ensure ‘xc7z010_1’ is selected, not the ‘arm’ device.

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

   (a) Open the XDC file 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.

(c) Demonstrate your progress to the TA once you have found the design to work.

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

   (a) Repeat steps (a) through (e) from 4.3 step 1, with the **four_two_encoder** using the 'four_two_encoder.xdc' file in the course directory.

   (b) Right click on 'two_four_decoder.xdc' and select 'Remove file from project' to remove the xdc file from project. **Do not forget to do this whenever you change designs to implement.** Vivado gives nasty errors when you have multiple XDC files in a project.

   (c) Click on 'Generate Bitstream File' in the Flow Navigator panel within Vivado.

   (d) If you are still in the Hardware Manager screen, instead of selecting 'Open Target' in Vivado, select 'Program Device' and select xc7z010_1.

   (e) Before programming, select 'four_two_encoder.bit' under Bitstream file and click on 'Program'

   (f) For the encoder, the inputs have been mapped to the push-buttons. Open up the corresponding XDC to verify this. Press the push-buttons on the ZYBO 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.

   (g) Now press more than one button and note what is displayed on the LEDs.

4. Finally, use the above steps to load the **priority_encoder** onto the FPGA using the 'priority_encoder.xdc' 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 report.

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 using a breadboard 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 the 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 report.
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 sections 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?