Contents

When clicking on a link in Adobe use alt+arrow left to return to where you started.

Objectives

These are the objectives of the lecture and homework sequence of the course. They have been arranged to parallel the chapter/section sequence in the textbook.

- Digital Design
  - Numbering Systems
    - Positional Numbering Systems
      - Base 10 - Decimal
      - Base 2 - Binary
      - Base 16 - Hexadecimal
    - Between Bases
    - Word Size
    - 2's Complement
  - Representation of Logical Function
    - Elementary Logical Functions
    - Analyzing a word statement for a logic function
    - Creating a truth table description for a logic function
    - Creating a symbolic form for a logic function
    - Creating a circuit diagram for a logic function
    - Creating Hardware Description Language statements for a logic function
    - Conversion between two different representations of a logic function
    - Describing a functions with multiple outputs
    - Timing Diagrams
  - Logic Minimization
    - Karnaugh Maps (Kmaps)
    - Kmaps for circuits with multiple outputs
    - Kmaps to find POSmin
    - Using logic minimization software to describe a logic function .2 Combination Logic Building Blocks
    - Decoder
    - Multiplexers
    - Adders
    - Comparators
    - Wire Logic
    - Combination
      - Arithmetic Statements

- Conditional Statements
- Basic Memory Elements
  - Characteristics
  - Timing
  - Asynchronous set reset
- Sequential Logic Building Blocks
  - Analysis, design and use of a register in a digital design
  - Shift Register
  - Counter
  - Static RAM
  - Register Transfer
- Finite State Machines
  - Hardware organization of a finite state machine
  - State diagram for a finite state machine
  - One's Hot Encoding
  - Design
  - Timing
- Datapath and Control
  - Datapath and Control Architecture
  - Algorithmic Language
  - Design
  - Timing

These are the objectives of the lab sequence in this class broken down into Verilog and FPGA. Verilog objectives are agnostic to the hardware platform. FPGA objectives are specific to the hardware and software platformed used to implement the designs laid out in these labs. This organization should enable instructors to quickly identify units which may need to be adapted to run the lab sequence on different hardware/software platform.

- Verilog
  - Writing concurrent signal assignment statements for a logic function
  - Writing a Verilog statement using primitive logic operations
  - Writing a Verilog statement using an Always/Case statement
  - Creating a Verilog statement that uses vectors
  - Analyzing and designing a Verilog testbench
  - Definition and instantiation of Verilog generic modules
  - Definition o Verilog modules
  - Instantiation of Verilog Modules
  - Definition of Finite State Machines in Verilog
- Hardware and Software Specifics
  - Creating a simulation timing diagram for a module
  - Creating a pin assignment for a module
  - Creating a Do file to automate waveform setup
  - Synthesizing a module on the FPGA development board

Laboratory 1

Introduction to Verilog

1.1 Outcomes and Objective

The outcome of this lab is to introduce you to the Quartus II software, design entry using Verilog and circuit simulation. Through this process you will achieve the following learning objectives.

- Elementary Logical Functions
- Conversion between two different representations of a logic function
- Writing concurrent signal assignment statements for a logic function
- Writing a Verilog statement using primitive logic operations
- Creating a simulation timing diagram for a module

1.2 FPGA: Creating a project in Quartus and running a testbench

In this section you will apply inputs to a 2-input AND gate and observing the output on a timing diagram. Since all this activity will take place in the memory of a computer and not on actual hardware, it is called a simulation. To start this process, you will first have to create a project and add files to it.

1. Select an appropriate working directory for your project. I would recommend selecting your network drive.
   a. Create a new folder *lab1*,
   b. Create another folder within *lab1* called *andgate2*,
   c. Download *andgate2.v* and *andgate2_tb.v* from Canvas,
   d. Save these files in andgate2 directory.
2. Start Quartus II.
   a. If you are prompted by a License Setup choose the free option. You may need to restart Quartus if this happens.
3. Select *File -> New Project Wizard.*
4. In the **Directory, Name, Top-Level Entity** page of the New Project Wizard pop-up:
   a. To the right of the "What is the working directory" box click the ... button,
   b. In the Select Folder pop-up, navigate so you can see the andgate2 directory created

in step 1,
c. Select the andgate2 folder, click Select Folder,
d. In the "What is the name of this project" field type *andgate2*
e. click *Next*.

5. In the **Project Type** page of the New Project Wizard pop-up:

   a. Select the *Empty project* radio button,
   b. click *Next*.

6. In the **Add Files** page of the New Project Wizard pop-up:

   a. Click the ... button to the right of File name,
   b. In the Select File pop-up, navigate to, and select, *andgate2.v* and *andgate2_tb.v*, click Open,
   c. The file should appear in the window below,
   d. Click *Next*

7. In the **Family & Device Settings** page of the New Project Wizard pop-up:

   a. Device family, Family: Cyclone V
   b. Package: FBGA
   c. Pin Count: 672
   d. Speed Grade: 7_H6
   e. Select Specific device selected in 'Available devices' list
   f. From the list of available devices, select: 5CGXFC5C6F27C7
   g. Click Next

8. In the **EDA Tool Settings** page of the New Project Wizard pop-up:

   a. In the Simulation row

      i. Tool Name column: ModelSim-Altera
      ii. Formats column: Verilog HDL

   b. Leave other defaults alone
   c. Click Next

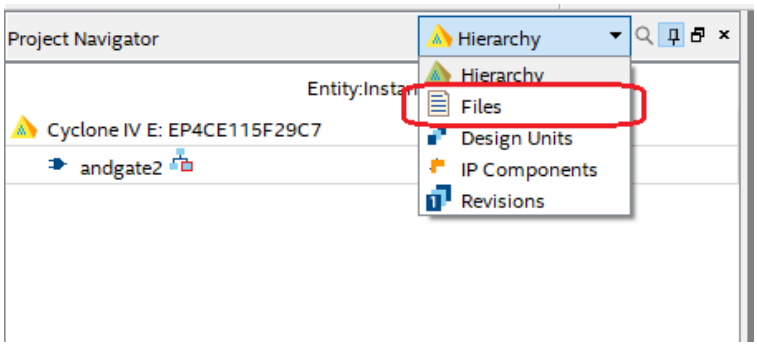
9. In the **Summary** page of the New Project Wizard pop-up:

   a. Review information,
   b. Click Finish.

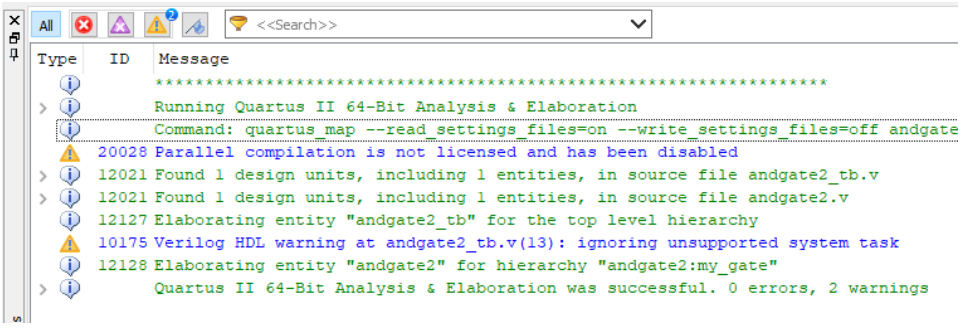
10. Back in the main Quartus II window, Click *Tools -> Options...*
11. In the Options pop-up:

    a. Select *EDA Tool Options* from the Category menu,
    b. If the last row, "ModelSim-Altera" is blank, click on the ... button at right and navigate to the *C:\intelFPGA_lite\18.1\modelsim_ase*, select the *win32aloem* folder, the click Select Folder. Note the software version in these instructions is 18.1 The version installed on your computer may be different. If so, the path should be the same with the exception of the version number.
    c. Click Ok.

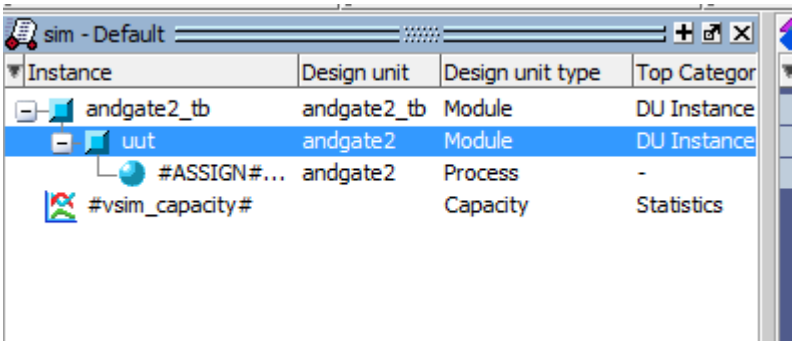
12. Click on the Files tab in the *Project Navigator* pane.



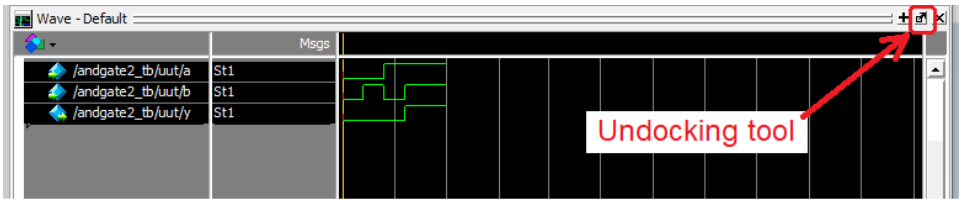
13. Right click on *andgate2_tb* in the *Project Navigator* pane and select Set as Top-Level entity.
14. Double click on *andgate2*.
15. If you added the Verilog file in the correct directory and included it in the project, a Verilog file should pop up on the right.
16. In the main Quartus II window, click on *Processing -> Start -> Start Analysis & Elaboration.* This may take some time, so be patient.
17. If you did everything correctly you should
    a. Notice that andgate_tb is the new top-level entity in the Hierarchy pane. Expand the andgate2_tb by clicking on the ">" arrow to see the entities inside it.
    b. You should see the following messages in the console area, the bottom pane.



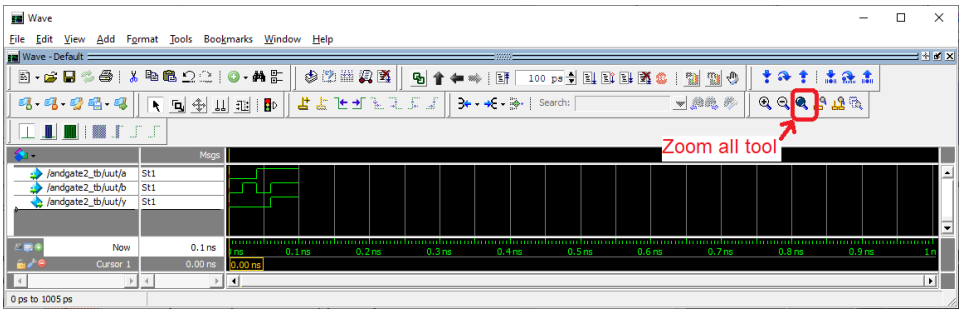
18. In the main Quartus II window, click *Tools -> Run Simulation Tool -> RTL Simulation.* The ModelSim program will launch. This may take a few moments, be patient. If you get a pop-up Nativelink Error window, then go back, check and fix the directory in step 11.
19. In ModelSim, click Simulate -> Start Simulation
20. In the Start Simulation pop-up, expand the *work* library by clicking on the "+" at left. click on *andgate2_tb* and click *Ok.*
21. In the sim pane, right mouse click on uut and select *Add Wave.*



22. Choose *Simulate -> Run -> Run 100*. You should see inputs and output from andgate2. If you see only a small green portion of the waveform on the left margin of the timing diagram, you will need to zoom in on the waveform as follows. First click somewhere in the timing diagram (area under "Undocking tool" in the image below) and then click on the "Zoom all tool" shown in following image.
23. Save this waveform as an image as follows:
    a. Undock the Wave pane by clicking the undocking tool icon.



    b. Resize the undocked Wave window vertically by grabbing its top edge and dragging down. Make the window tall enough to fit all the waves with a little room to spare.



    c. Click the Zoom all tool to file the available horizontal space with the waveform.
    d. Click File -> Export -> Image

    If this does not work, you can take a screen shot of the window by pressing Alt-Print Screen. The "Alt" captures the currently active window into the graphics buffer.

    e. Navigate to your project directory, provide a File name, then click Save
    f. Exit Modelsim using File -> Quit. Do not save wave commands.
24. Back in Quartus, close your current project using File -> Close Project. Save if needed.

1.3 FPGA: Symbolic to Verilog , Timing Diagram, Truth Table

Write Verilog code to realize the function *f02 = a' + bc'* Note that this symbolic expression is written using the notation used in class. This is not a valid Verilog expression.

1. Create **a new project** folder within your *lab1* directory called *function02.*
2. Download *function02.v* and *function02_tb.v* from Canvas to the project directory.
3. Create a project for these two files using the steps above.
4. Modify the line of code that starts with *assign* to realize the function *f02* shown above.
5. Modify *function02_tb.v* so that *f02* is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0 and going to 1,1,1.
6. Perform simulation using the given testbench as described in previous steps. You will need to "run 100" twice as the simulation is over 100ns long.
7. Save this waveform as an image as done in the previous section. If the waveform is missing, you can add it back in using View -> Waveform.
8. From the information in the timing diagram, produce a truth table for *f02.* Remember that a truth table is an enumeration of every possible input and the associated output. Please look at Chapter 2 in the textbook for some examples if you are unclear about how to setup a truth table.

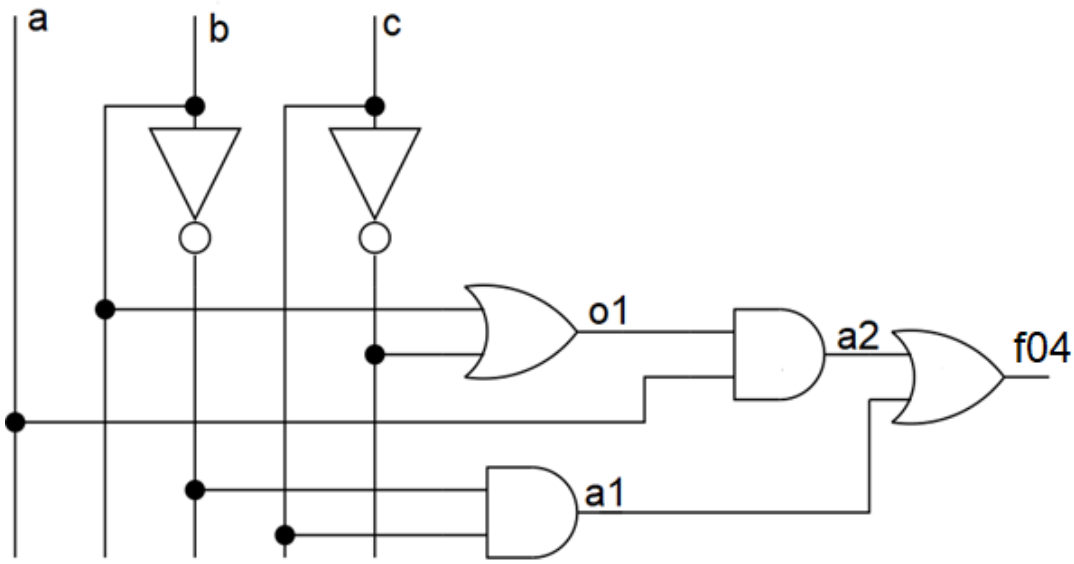
1.4 FPGA: Verilog to Symbolic, Truth Table, Circuit Diagram

The Verilog code in the file function03 contains a complete circuit for *f03.* You will use the Quartus tools to get a timing diagram for the function and, by looking at the Verilog code, determine the symbolic form and circuit diagram.

1. Create **a new project** folder within your *lab1* directory called *function03.*
2. Download *function03.v* and *function03*_tb.v from Canvas to the project directory.
3. Create a project for these two files using the steps above.
4. Modify *function03_tb.v* so that *f03* is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0 and going to 1,1,1.
5. Perform simulation using this test bench as described in previous steps. You will need to "run 100" twice as the simulation is over 100ns long.
6. Save this waveform as an image, but with the following changes.
   a. Resize the area containing the names of the signals by grabbing the right vertical bar of the name area and moving it right.
   b. Re-order the waves so that f03 is lowest. Do this by grabbing the name "/function03_tb/uut/f03 and moving it below all the other signals.
   c. Color the intermediate signals (p1, p2, p4, p7) yellow by right clicking on them, selecting properties. In the View tab of the Wave Properties pop-up, click the Colors... button for Wave Color and choose Yellow, click Close, then OK.
   d. Change the color of *f03* to red.
7. From the information in the timing diagram, produce a truth table.
8. From the information in *function03.v* draw the circuit diagram for *f03.*
9. From the information in *function03.v* write down the symbolic form for *f03.*

1.5 FPGA: Circuit Diagram to Verilog, Symbolic, Truth Table

Write Verilog code to realize the function *f04* shown in the circuit diagram below.



1. Create **a new project** folder within your *lab1* directory called *function04*.
2. Download *function04.v* and *function04*_tb.v from Canvas to the project directory.
3. Create a project for these two files using the steps above.
4. Modify *function04.v* by writing an assignment statement for each of *o1*, *a1*, *a2*, and *f04*.
5. Modify *function04_tb.v* so that *f04* is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0 and going to 1,1,1.
6. Perform simulation using this test bench as described in previous steps. You will need to "run 100" twice as the simulation is over 100ns long.
7. Save this waveform as an image as done in a previous section. Color intermediate signals (01, a1, a2) yellow and output red.
8. From the information in the timing diagram, produce a truth table.

1.6 Turn in

Make a record of your response to numbered items below and turn them in a single copy as your team's solution on Canvas using the instructions posted there. Include the names of both team members at the top of your solutions. Use complete English sentences to introduce what each of the following listed items (below) is and how it was derived.

FPGA: Creating a project in Quartus and running a testbench

- Step 23 Timing diagram of AND gate

FPGA: Symbolic to Verilog , Timing Diagram, Truth Table

- Step 4 Verilog code for *f02*
- Step 7 Timing diagram of *f02*
- Step 8 Truth table of *f02*

FPGA: Verilog to Symbolic, Truth Table, Circuit Diagram

- Step 6 Timing diagram of *f03*
- Step 7 Truth table of *f03*
- Step 8 Circuit Diagram of *f03*
- Step 9 Symbolic form of *f03*

FPGA: Circuit Diagram to Verilog, Symbolic, Truth Table

- Step 4 Just the 4 Verilog assign statement for *o1*, *a1*, *a2*, and *f04*.
- Step 7 Timing diagram of *f04*
- Step 8 Truth table of *f04*

Laboratory 2

Hexadecimal to Seven-Segment Converter

2.1 Outcomes and Objectives

The outcome of this lab is to instantiate a hexadecimal to seven segment converter on the FPGA development board. Through this process you will achieve the following learning objectives.

- Creating a truth table description for a logic function
- Describing a functions with multiple outputs
- Analyzing a word statement for a logic function
- Creating a Verilog statement that uses vectors
- Writing a Verilog statement using an Always/Case statement
- Creating a pin assignment for a module

2.2 Verilog: Vectors

A vector is a collection of bits that are realted to one another in some way. For example, the individual bits of a 4-bit number could be represented as a vector. There are three things that you will need to know about vectors in order to complete today's lab (and future labs), combining bits into a vector, defining a vector, and accessing the bits of a vector. These operations are illustrated in Figure 2.1.

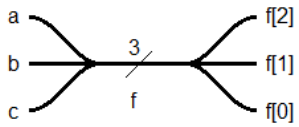


Figure 2.1: An illustration showing three bits combined into a vector, f, and then accessing the individual bits of f.

The operations shown in Figure 2.1 are similar to the operations in Listing 2.1. In this

module, the line of code assign `f = {a,b,c};` combines the individual signals a, b and c into a 3-bit vector f. The left-most signal in the parenthesis list, *a*, becomes the MSB of the vector and the right-most signal, *c*, becomes LSB. Combining signals is more commonly called concatenation. You can concatenate any arrangement of signals as long as the number of bits comes out the same as the signal on the left-hand-side of the = sign.

Listing 2.1: Verilog code which illustrates vector manipulations and declarations.

```
module unimportantModuleName();

    wire a, b, c, x;                    // Just some plain old wires
    wire [2:0] f, g, h;                 // 3-bit vectors

    assign f = {a,b,c};                 // Concatenate bits to vector
    assign g = {f[0], f[2:1]};          // re-arrange bits

    assign x = (f[0] & f[1]) ^ f[2];    // vectors are made of bits
    assign h = 3b'010;                  // A constant vector to h
```

The statement `wire [2:0] f;` defines the vector f as having 3 bits. The numbers in the square brackets are the indices of the most and least significant bits of the vector. We will always index our vectors starting at 0, so the highest index will always be one less than the number of elements in the vector.

The statement `assign x = (f[0] & f[1]) ^ f[2];` shows how you can access the individual bits of a vector by putting the bit index in square brackets. You can also access sub-vector by putting indices in square brackets separated by a colon, e.g. `f[2:1]`

You can provide a constant value to a vector, an operation we will call hardcoding, using the 3b'010; notation. The first number, 3, defines the number of bits in the vector, b' means that this is a bit vector and the 010 is the 3-bit value.

2.3 Verilog: Always/Case statements

We will use the Verilog *always* statement to implement a function using its truth table. Listing **??** shows an always statement that uses the 3-bit value of a signal x to compute the value of f.

Listing 2.2: A 3-input, 2-output function realized with an always statement. Can you figure out how the output was computed?

```
    wire [2:0] x;
    reg [2:0] f;

    always @(*)
        case (x)
            3'b000: f = 3'b00;
            3'b001: f = 3'b01;
            3'b010: f = 3'b01;
            3'b011: f = 3'b10;
            3'b100: f = 3'b01;
            3'b101: f = 3'b10;
            3'b110: f = 3'b10;
```

```
            3'b111: f = 3'b11;
        endcase
```

For the time being, we will trust that the statement always `@(*)` allows the code between `case` and `endcase` to run continuously and concurrent with any other statements in the module. Yes, this means that all the code between `case` and `endcase` acts like a single assign statement. A case statement uses the argument to case, x as a selector for one of the rows below. Every possible value of x must be present and when that value matches x, the action to the right of the colon is performed. When we use a case statement as shown in Listing 2.2 you must make the output type reg.

All signals are either `wire` or `reg` type. A wire is a signal that has a value provided to it by some active element. This active element might be a gate or the output of a module. If a signal does not have an explicit gate or module driving its value, it needs to be typed reg.

2.4 A Multiple Output Function

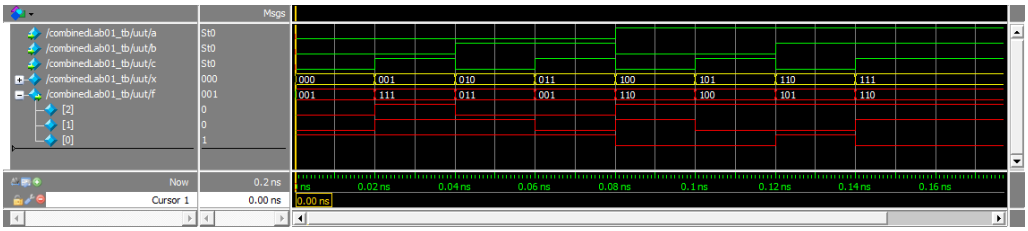
Let's explore vectors and the always statement by combining the three functions created in last weeks assignment into one function.

1. Go back to your Lab 01 solutions and extract the truth tables for function f04, f03, and f02. Put these values into the truth table shown in Table 2.1.

Table 2.1: The Truth Table for the combinedLab01 function. This function has a 3-bit input and 3-bits output.

a	b	C	f04	f03	f02
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

1. Create **a new project** folder within your *lab2* directory called *combinedLab01.*
2. Download *combinedLab01.v* and *combinedLab01*_tb.v from Canvas to the project directory.
3. Create a project for these two files using the steps from last week's lab.
4. Modify *combinedLab01.v* so that *combinedLab01* outputs the values given in Table 2.1.
5. Modify *combinedLab01_tb.v* so that *combinedLab01* is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0 and going to 1,1,1.
6. Perform simulation using with the test bench using the steps from last week's lab.
7. Capture the output waveform from Simulink. It should look something like the following.



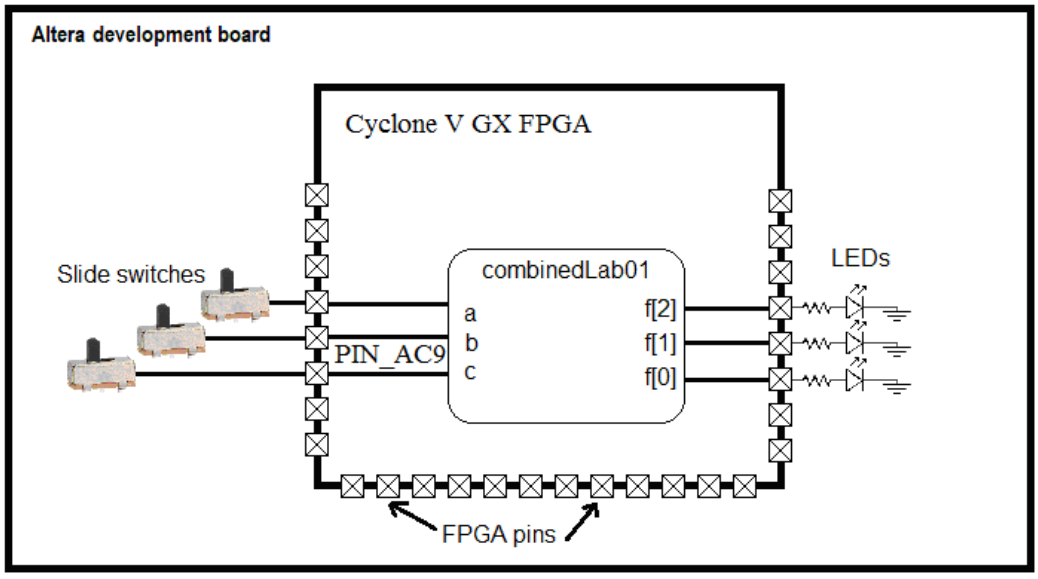
8. From the information in the timing diagram, produce a truth table. Compare the truth table generated from the data in the timing diagram to that you generated in Table 2.1.

2.5 FPGA: Pin-Assignment

The process of converting your Verilog code to a form which you will download onto the development board is called *synthesis*. In order to synthesize your Verilog code, you need to tell the Quartus software which pins of the FPGA are associated with the ports in your top-level Verilog module. In order to perform this assignment, you need to know which pins of the FPGA are associated with useful hardware on the development board. The engineers who created the development board made the assignment of hardware components to FPGA pins when they laid out the printed circuit board. These same engineers documented their decisions in the Cyclone V GX Kit User Manual posted on the class web page.

The Figure 2.2 shows a Verilog module called *combinedLab01* synthesized and downloaded into an Altera FPGA on the development board. Note that ports a, b and c are connected to FPGA pins that are driven by slide switches. Ports f[2], f[1] and f[0] are connected to FPGA pins that drive LEDs. In this way, a user can provide input to the *combinedLab01* module by moving the slide switches and observe the circuit's output on the LEDs.

Figure 2.2: A simple Verilog design synthesized and downloaded onto the development board.



The development board contains an Altera Cyclone V GX FPGA. This FPGA has many pins and they are identified by a lettered group and number. For example, in Figure 2.2 port c of the combinedLab01 module is mapped to pin AC9.

You will need to be able to figure out the remaining pin assignments on your own. To do this open up the C5G User Manual posted on the class Canvas page. Start with Figure 3-9 on page 30 which shows that the slide switches in one of two positions (up or down). In the up position, they assert a logic 1 and down they assert a logic 0. On the next page, 31 of the C5G User Manual look at Table 3-3. This table defines the relationship between the different slide switches and the FPGA pins each is connected to. For example, slide switch SW[0] is connected to PIN_AC9.

Board Reference	*Schematic Signal Name*	*Description*	*I/O Standard*	*Cyclone V GX Pin Number*
SW0	SW0	Slide Switch[0]	1.2-V	PIN_AC9

Figure 3-10 on page 31 of the C5G User Manual shows that the red and green LEDs are active high, meaning that the LED is active (illuminated) when you send it a high signal (logic 1). Consequently, sending the LED a logic 0 turns them off. The pin assignment for the LEDs is given in Table 3-4 on page 32. Note that "R" in "LEDR" means red and "G" stands for green.

Use the information to complete the pin assignment in Table 2.2. We will use this assignment in the next section.

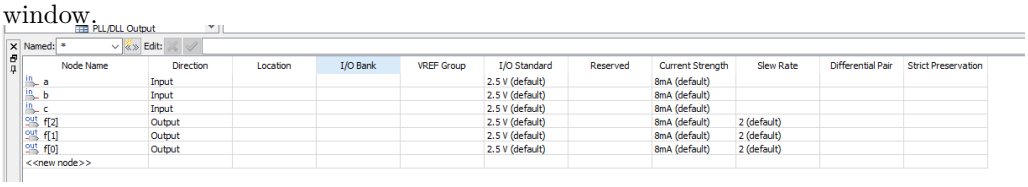
Table 2.2: Pin Assignment Table for combinedLab01.

Port	a	b	c	f[2]	f[1]	f[0]
Signal name	SW[2]	SW[1]	SW[0]	LEDR[2]	LEDR[1]	LEDR[0]
FPGA Pin No.			PIN_AC9			

2.6 FPGA: Synthesizing a Verilog Module

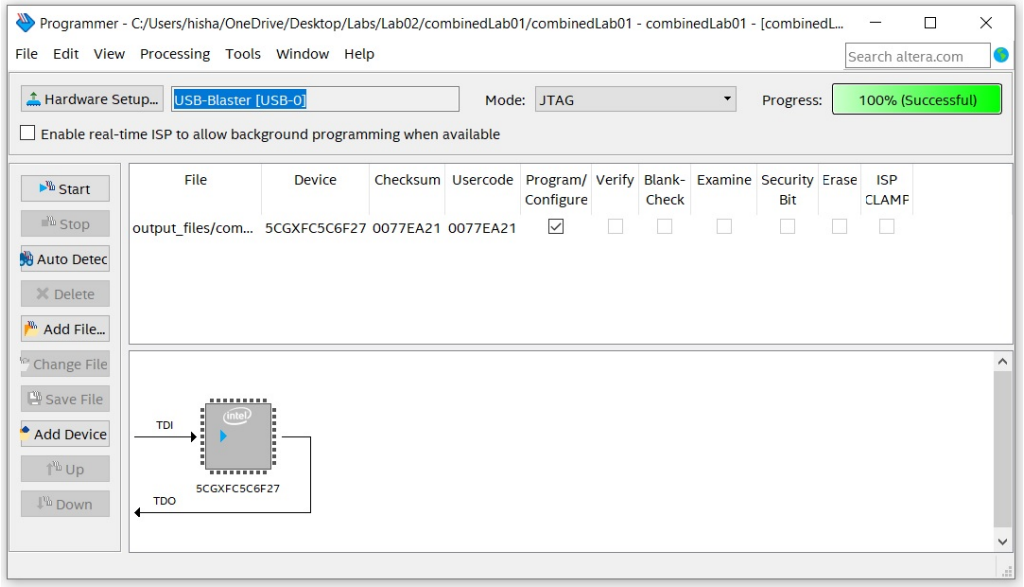
It's time to complete the synthesis process by downloading the combinedLab01 Verilog file along with its pin-assignment to the FPGA. Work through the follow these steps to accomplish this.

1. In Project Navigator pane, select the File tab
2. Right mouse click *combinedLab01.v* and select Set As Top Level Entity.
3. Processing -> Start -> Start Analysis and Elaboration
4. Assignments -> Pin Planner
5. In the Pin Planner pop-up you should see the pin assignment pane at the bottom of the window.



6. Double click in the Location cell for row c
7. Scroll down the list of pins to PIN_AC9
8. Complete the pin assignment for the other 5 inputs and outputs using the information contained in pin assignment table completed earlier.

9. Double check your pin assignments.
10. File -> Close. Note closing your file incorporates this assignment into the project.
11. Back in the Quartus window, Processing -> Start Compilation <Ctrl-L>
12. Tools -> Programmer
13. In the Programmer pop-up window click Add File...
14. In the Select Programming File pop-up, navigate to your project directory, then into the output files folder, the select combinedLab01.sof, click Open. You should see something like the following.



15. Connect the Altera Cyclone V GX FPGA to your computer through the USB port, connect the power supply, and push the red power-on button. Try not to be annoyed by the infernal blinking LEDs.
16. In the Programmer pop-up
    a. Click Hardware Setup....
    b. In the Hardware Setup select USB-Blaster [USB=0] from the Currently selected hardware pull-down
    c. Click Close
17. Back in the Programmer window, the box next to Hardware Setup... should reflect your choice. Click Start,
18. The Development board should stop its infernal blinking and run your program. You may notice that the unused LEDs are dimly illuminated.
19. Move the slide switches up and down to verify that the input/output matches the values in Table 2.1. Use white silk screen printing on the development board to locate slide switches and LEDs you assigned in your pin-assignment.

2.7 Hexadecimal to 7-segment Converter

While working on the previous problem, you probably noticed that the development Board has four 7-segment display. These figure 8 shaped blocks above the slide switches are the devices which light up numbers on some cash registers. We will be using these 7-segment displays for a variety of purposes during the term, so it would be a good idea.

The hexadecimal-to-seven-segment-decoder is a combinational circuit that converts a hexadecimal number to an appropriate code that drives a 7-segment display the corresponding value. BEWARE, the LEDs in the 7-segment displays on the Development Board are active low, asserting a logic 0 on the pin attached to a segment will cause that segment to illuminate.

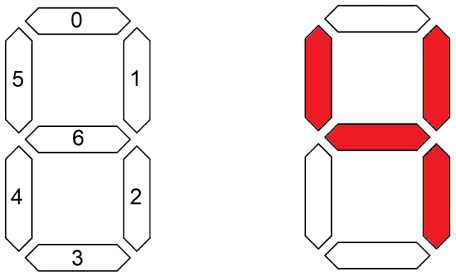


Figure 2.3: Left, the proper numbering of the segments. Right, illuminating segments to form the number 4.

The pattern of segments to be illuminated for each digit is shown in Figure 2.3. For example, to display ‘4 output would be:

```
seg[6]=0   seg[5]=0   seg[4]=1   seg[3]=1   seg[2]=0   seg[1]=0   seg[0]=1
or seg = 7'b0011001
```

Figure 2.4 shows the proper formatting for all the values between 0 – f.

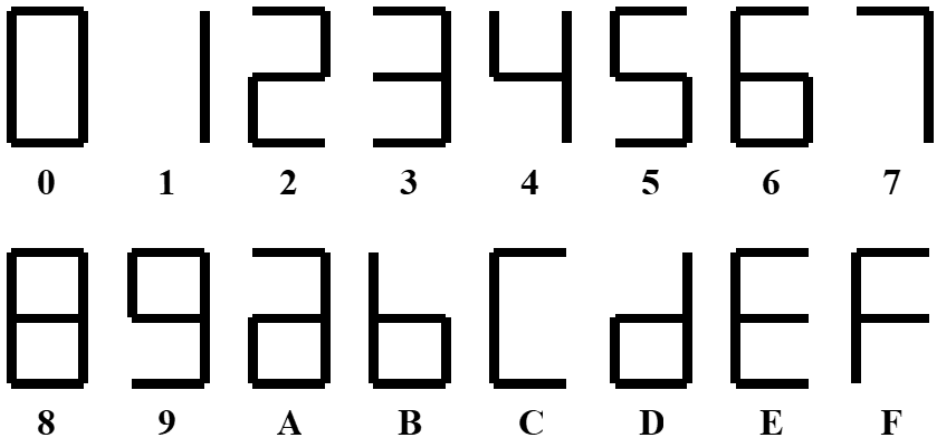


Figure 2.4: The proper arrangement of LEDs to form hexadecimal characters.

Use this information to complete the Table 2.3 to illuminate the active low led segments to generate proper hexadecimal characters.

Table 2.3: Truth table for the hexToSevenSeg component.

x	seg[6]	seg[5]	seg[4]	seg[3]	seg[2]	seg[1]	seg[0]
0000							
0001							
0010							
0011							
0100	0	0	1	1	0	0	1
0101							
0110							
0111							
1000							
1001							
1010							
1011							
1100							
1101							
1110							
1111							

Now that you have a complete desrciption of the input/output behavior of the hexadecimal to seven segment converter. its time to write the Verilog code. You will capture the behavior in Table 2.3 using an always/case statement. Work through the following steps to complete this task.

1. Create **a new project** folder within your *lab2* directory called *hexToSevenSeg.*
2. Download *hexToSevenSeg.v* and *hexToSevenSeg _tb.v* from Canvas to the project directory.
3. Create a project for these two files.
4. Complete the case statement for *hexToSevenSeg.v*

Testbench

With your Verilog code complete, you need to verify your logic before synthesis. While this may seem a waste of time for such a simple design, you are building skills that are essential to debugging the complex designs you will create later in the term. Work through the following steps to complete this task.

1. Modify *hexToSevenSeg _tb.v* so that *hexToSevenSeg* is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0,0 and going to 1,1,1,1.
2. Perform simulation using this test bench as described in previous steps. You will need to "run 100" several times to go through all the inputs.
3. Save this waveform as an image as done in the previous section. If the waveform is missing, you can add it back in using View -> Waveform.
4. Compare From the information in the timing diagram, produce a truth table for in Table 2.3. Fix any errors in the always/case statement before proceeding to synthesis in the next step. *hexToSevenSeg.*

Synthesis

Before you can download your design to the FPGA, you need to assign the input and outputs of the hexToSevenSeg module to FPGA pins. Figure 2.5 shows the slide switches and 7-segment display that will use.

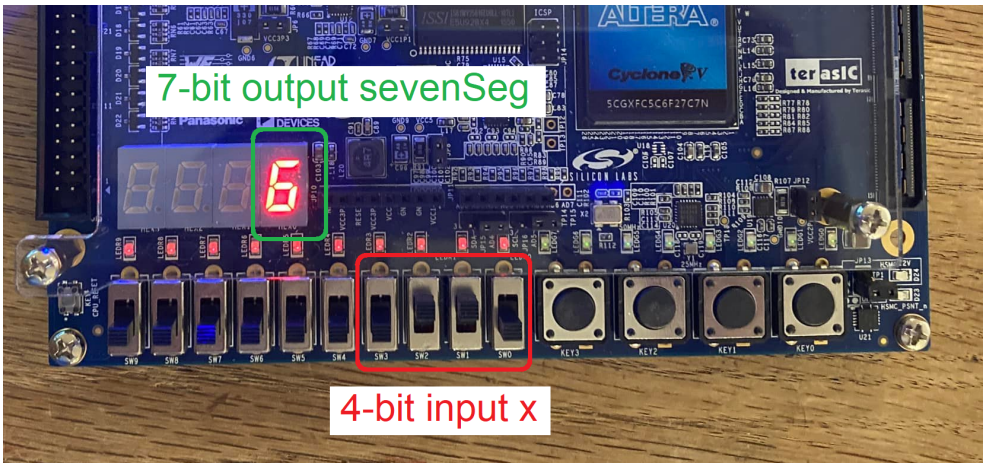


Figure 2.5: The input set to 4'b0110 displaying a 6 on the 7-segment display.

Use the C5G User manual to complete the pin-assignment in Table 2.4.

Table 2.4: Pin-assignment tables for the hexToSevenSeg module.

Port	x[3]	x[2]	x[1]	x[0]
Signal name	SW[3]	SW[2]	SW[1]	SW[0]
FPGA Pin No.				PIN_AC9

Port	sevenSeg[6]	sevenSeg[5]	sevenSeg[4]	sevenSeg[3]	sevenSeg[2]	sevenSeg[1]	
Signal name	HEX0[6]	HEX0[5]	HEX0[4]	HEX0[3]	HEX0[2]	HEX0[1]	HEX0[0]
FPGA Pin No.							

Use the instructions in Section 2.6 to combine the pin assignment with your hexToSevenSeg module. Synthesize your design, bask in the glow of another success as you demonstrate your circuit's functionality to a member of the lab team.

2.8 Turn in

Make a record of your response to numbered items below and turn them in a single copy as your team's solution on Canvas using the instructions posted there. Include the names of both team members at the top of your solutions. Use complete English sentences to introduce what each of the following listed items (below) is and how it was derived.

Combine lab 1

- Truth Table for combinedLab01 function in Table 2.1
- Link Timing diagram for combinedLab01 function
- Pin assignment for combinedLab01 in Table 2.2

Hexadecimal to 7-segment

- Truth Table for hexToSevenSeg function in Table 2.3
- Link Verilog code for hexToSevenSeg function – just the always/case statement
- Link Timing diagram for hexToSevenSeg function
- Pin assignment tables for hexToSevenSeg in Tables 2.4
- Demonstrate working hexadecimal to seven segment module to a member of the lab team.

Laboratory 3

Rock Paper Scissors

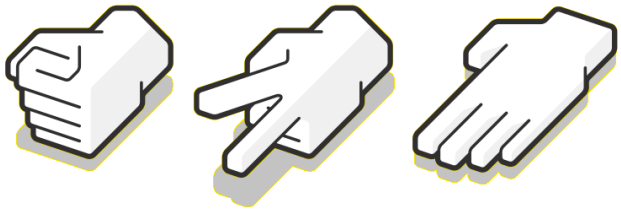
3.1 Outcomes and Objectives

The outcome of this lab is to instantiate a rock, paper scissors games on the FPGA development board. Through this process you will achieve the following learning objectives.

- Analyzing a word statement for a logic function
- Creating a truth table description for a logic function
- Writing concurrent signal assignment statements for a logic function
- Definition o Verilog modules
- Instantiation of Verilog Modules
- Creating a pin assignment for a module
- Synthesizing a module on the FPGA development board

3.2 The Rock Paper Scissors Game

The game of rock, paper, scissors is a two-player game whose goal is to beat the throw of the opposing player. Traditionally, each player throws one of three plays, rock, paper, or scissors by extending their hand in the shape of the object.



The rules of the game state that:

Rock beats scissors Scissors beats paper Paper beats rock

Since prior knowledge of your opponent's throw would provide an unfair advantage, the two players make their throws at the same time. Your goal in this lab is to create a digital

version of rock, paper, scissors on the Altera Cyclone V Board using the inputs and outputs shown in Figure 3.1. Each player will have access to three slide switches and one 7-segment display. Each of the three switches represents one of the three plays and the 7-segment will display the throw when the Play button is pressed. The Win/Lose 7-segment display will show "1" when player 1 wins, "2" when player 2 wins, and "d" when the game is a draw (a tie).

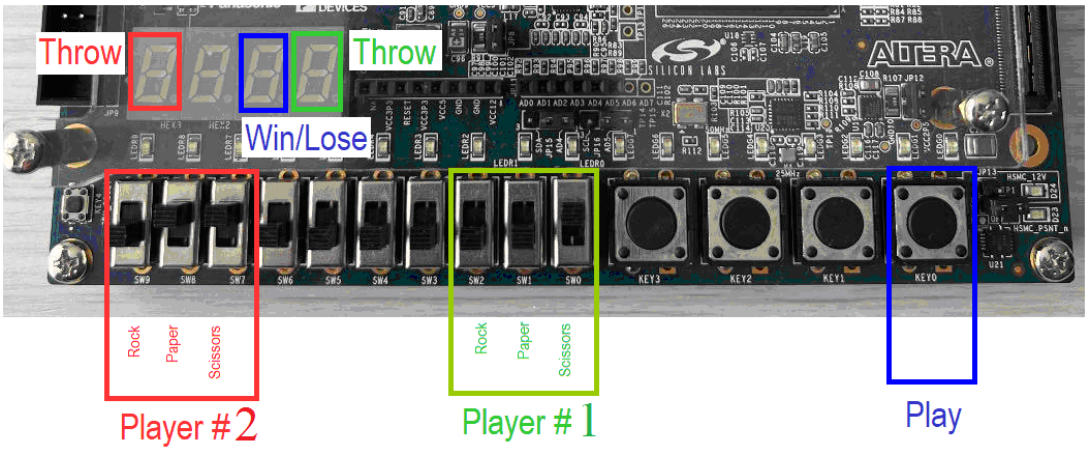


Figure 3.1: The input and output you should use to realize your digital system.

A player will move one of the three slides-switches into the up position to indicate their play. Moving more than one slide-switch, or no slide switch into the up position is in an invalid play. An invalid play always loses to a valid play. If each player throws an invalid play, the game is a draw.

While each player is making their choice of play, their Throw 7-segment display will reflect their choice as shown in Figure 3.2. These patterns are supposed to vaguely resemble the objects.

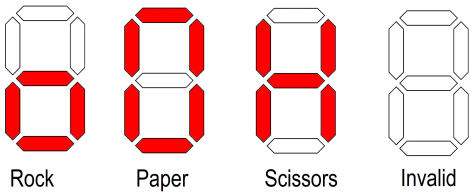


Figure 3.2: Illuminated patters for the different plays.

When the Throw button is pressed, the Win/Lose 7-segment will display "1", "2", or "d" depending on the outcome of the game as shown in Table 3.1. When the Throw button is un-pressed, the Win/Lose 7-segment display is blank.

Table 3.1: The output for every combination of player 1 (P1) and player 2 (P2) throws.

P1 \ P2	Rock	Paper	Scissors	Invalid
Rock	d	2	1	1
Paper	1	d	2	1
Scissors	2	1	d	1
Invalid	2	2	2	d

3.3 System Architecture

The system architecture shown in Figure 3.3 is your guide to building a functioning circuit. As such, we will take a moment to cover some important details of this diagram that will help you write your Verilog code later./

The names outside the FPGA square correspond to the labels in Figure 3.1. Each soft-square (a square with rounded corners) is a Verilog module. Names inside soft-squares, that are adjacent to lines outside the soft-square, are the port names for that module. The instance name and module name of a module are separated by a ":" and usually located along the top edge of the soft-square. Red soft-squares are associated with player 1 and green soft-squares associated with player 2. The names on lines inside the rpsGame soft-square are the signals names you should use in the rpsGame module to connect the 5 modules together. Lines that are slashed with a number denote bit vectors.

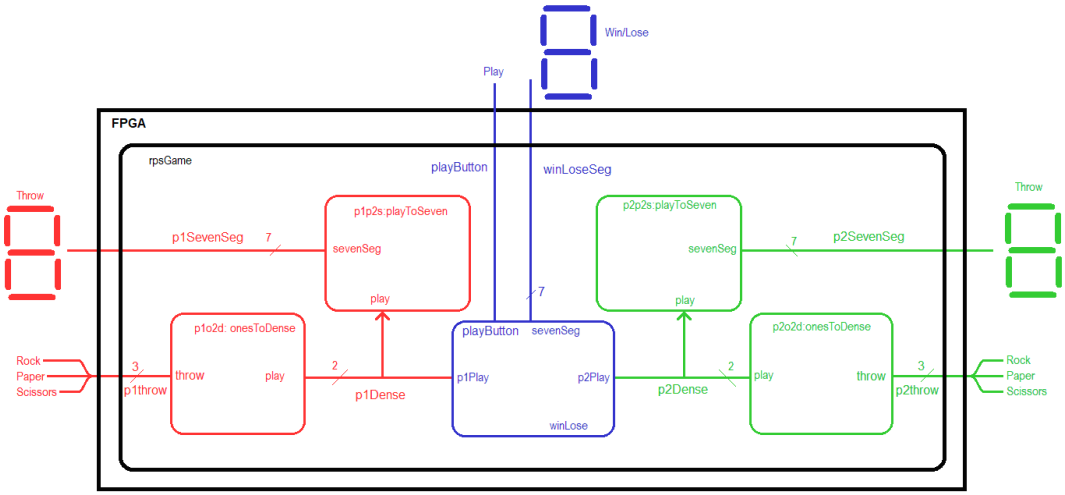


Figure 3.3: System architecture for the rock, paper, scissors system.

3.4 Module: onesToDense

Each player makes their throw selection by placing one of the three slide switches into the up position. As a result of this game mechanic, there are only three valid input combinations for the rock, paper, scissors trio. These are:

- (1,0,0) for when the player moves only the rock slide switch up,
- (0,1,0) for when the player moves only the paper slide switch up,
- (0,0,1) for when the player moves only the scissor slide switch up.

Having a code where only one of the bits is equal logic 1 is called a "ones-hot" code. The "hot" bit being logic 1. A code where every possible combination of bits is assigned a meaning is called a dense code.

This module will convert the input ones-hot code into a dense code. In order to correctly determine the outcome of the game, we need to know when the user has entered an invalid play; the output of this module must be able to represent {rock, paper, scissors, invalid}. You will encode these four combinations in 2-bits as {2'b00, 2b'01, 2'b10, 2'b11} respectively.

In order to write the Verilog code for this module, complete the truth table in Table 3.2 for the onesToDense module.

- r is the state of the rock slide-switch. r=0 slide switch is down. r=1 slide-switch up.
- p is the state of the paper slide-switch. p=0 slide switch is down. p=1 slide-switch up
- s is the state of the scissor slide-switch. s=0 slide switch is down. s=1 slide-switch up
- play = {00} means rock was selected
- play = {01} means paper was selected
- play = {10} means scissor was selected
- play = {11} means invalid selection was made

Table 3.2: The truth table for the onesToDense module.

r	p	s	play	Note
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0	00	Rock
1	0	1		
1	1	0		
1	1	1		

From this truth table, determine the canonical SOP expressions for play[1] and play[0] functions. Do this by writing the canonical SOP expression for the most significant bit of the play output, play[1], in the Table 3.2 truth table while ignoring the LSB. Then proceed to write the canonical SOP expression for the LSB of the play output, play[0], in the Table 3.2 truth table while ignoring the MSB.

play[1] =

play[0] =

Now it's time to write the Verilog code. Incorporate the following into your onesToDense Verilog module:

- Use the module declaration: module onesToDense (throw, play);
- Use vectors for the throw input. The MSB should come from the rock slide-switch and the LSB from the scissors slide-switch.
- Use a vector for the play output.
- Make the input and output port types "wire".
- You may want to break the input vector into its component pieces to correspond to the variable names used in Table 3.2. This will require a wire declaration and two assign statements to give each variable 1-bit from the input vector.

- Use assign statements to realize the AND, OR, NOT logic derived for your canonical SOP expressions.
- Use function04 from lab 1 as a starting point for this module.

3.5 Module: playToSeven

The playToSeven module converts the player throw, represented in the dense coding, to a "graphical" form displayed on the 7-segment display. The symbols for each possible throw are shown in Figure 3.4.

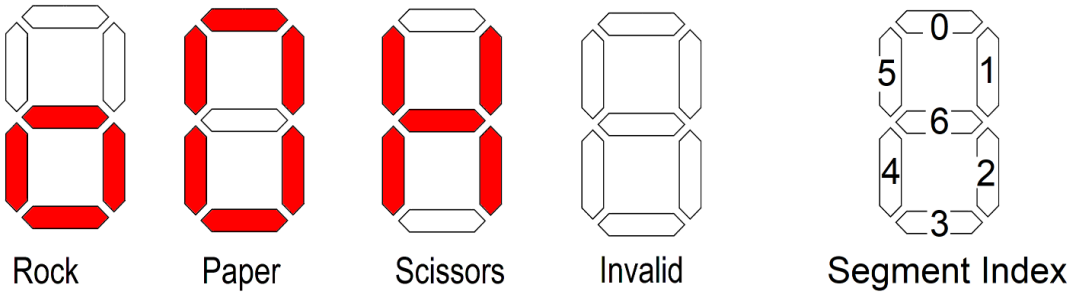


Figure 3.4: 7-segment display patterns for the different throws along with the 7-segment display segment indices.

Use the information in Figure 3.4 to determine the bit patterns needed to generate the four different symbols in Table 3.3. Remember that the LEDs in the segments are active low, meaning that a logic 0 output illuminates an LED segment. In Table 3.3 put a 0 or 1 in each of the numbered column so that each row produces the patterns for its throw. Remember that pPlay codes rock as 00, paper as 01, scissors as 10 and an invalid throw is coded as 11. In the sevenSeg column put the 7-bit code formed by concatenating the bits together. Use proper Verilog syntax to write this 7-bit vector.

Table 3.3: The 7-segment display LEDs to produce the throw patterns.

pPlay	6	5	4	3	2	1	0	sevenSeg	Note
00									Rock
01									Paper
10									Scissors
11								7'b1111111	Invalid

Incorporate the following into your playToSeven Verilog module:

- Use the module declaration: `module playToSeven (pPlay, sevenSeg);`
- Use a vector for the input pPlay and a vector for the sevenSeg output.
- Make the input port type "wire". Make the output port type "reg".
- Use a case statement, embedded in an always statement to realize this module. Enumerate all combinations of the input; do not use a default case
- Use the information in Table 3.3 to assign values to the output.

- Put comments at the end of each case row describing, in words, what the output should look like on the 7-segment display.
- Use the hex2Seven module from lab 2 as the starting point for this module.

3.6 Module: winLose

The winLose module takes the throw from each player (in the dense coding), the push button, and determines what to display on the win/lose 7-segment display. The win/lose 7-segment display is blank when the button is not being pressed, otherwise it will show "1" when player 1 has a winning throw, "2" when player two has the winning throw, "d" when the players have the same throw. The patterns are the same as those you have already created for the hexToSeven module and are shown in Figure 3.5 as a reminder.

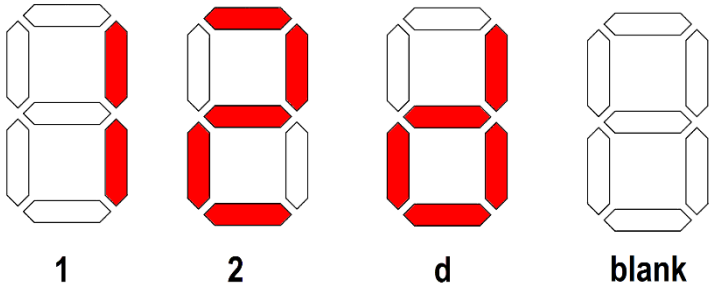


Figure 3.5: The illuminated patterns displayed on the win/lose 7-segment display.

You will use a case statement, based on the information in Table 3.4, to realize this function. In the sevenSeg column put the 7-bit code needed to illuminate the winLose 7-segment display to indicate the outcome of the game. In the Note column put either "1", "2", "Draw", or "Blank" depending on the outcome of the game and button press. Use the bit order given in Figure 3.4.

Table 3.4: Abbreviated truth table for the winLose module.

button	p1Play	p2Play	sevenSeg	Note
0				
0				
0				
0				
0				
0				
0				
0				
0	10 (scissors)	00 (rock)	7'b0100100	2
0				
0				
0				
0				

button	p1Play	p2Play	sevenSeg	Note
0				
0				
0				
1	xx (don't care)	xx (don't care)	7'b1111111	Blank

Incorporate the following into your winLose Verilog module: Use the module and port names given in Figure 3.3.

- Use the module declaration: `module winLose(p1Play, p2Play, playButton, sevenSeg);`
- Use vectors for the p1Play and p2Play inputs.
- Use a vector for the sevenSeg output.
- Make the input port types "wire". Make the output port types "reg".
- You will use a case statement, embedded in an always statement to realize this module.
  - Use brackets to make the vector for the case statement. For example, if button was a 1-bit signal and play was a 2-bit signal, then

```
case ({button, play})
    3'b000:  seg = 7'b1000000;     // display ``0''
    3'b001:  seg = 7'b1111001;     // display ``1''
    3'b010:   seg = 7'b0100100;     // display ``2''
    3'b011:   seg = 7'b0110000;     // display ``3''
    default:  seg = 7'b1111111;     // blank 7-seg
endcase
```

  This code snippet will use the 3-bit value (button as MSB and play as least significant 2-bits) to select one of the rows. Note that the default case handles all the combinations where button is 1.
- Use a default case to handle all the situations where the button is not pressed. The default case catches any unspecified input combinations for the case statement. List the default as the last row in the case list.
- At the end of each "case" row, provide a comment that lists player 1's throw, player 2's throw and the output that is displayed on the 7-segment display. For example, in my program the first case row has a comment that looks like:
  `// P1: Rock P2:Rock Draw`
- Use the hex2Seven module from lab 2 as the starting point for this module.

3.7 Module: rpsGame

The rpsGame module, "glues" together the modules shown in Figure 3.3 and serves as the top-level entity. The Verilog code for this module consists of 5 instantiation statements; one of them is given as the last bullet point item in the list below. For this module, I want you to:

- Use the module declaration:
  `module rpsGame(p1Throw, p1SevenSeg, p2Throw, p2SevenSeg, playButton, winLoseSeg);`
- Make the p1Throw and p2Throw inputs vectors with the MSB coming from the rock slide-switch input and scissors slide-switch as the LSB. You will need to keep this consistent with the pin assignment that you will complete next.
- The `playButton` input is not a vector.
- Use a vector for the `winLoseSeg` output.

- Make the input and output port types “wire”.
- You need to create 2 internal vectors. Look carefully at Figure 3.3 and find wires that begin and end inside the rpsGame module. These are the vectors.
- Name the module instances using the names provided in Figure 3.3.
- When you instantiate a module
  - The first term is the name of the module you are instantiating
  - The second term is the instance name of the module
  - The remaining term is the parenthesis list of signal in and out of the module. The order of the signals in the instantiation must be the same as those in the module declaration. Pay special attention to this!
  - For example, in my program I had an instantiation that looked like:
    `onesToDense p1o2d(p1Throw, p1Dense);`

3.8 Pin-Assignment

We are forgoing a simulation of today’s lab. This requires that you pay especially close attention to the warnings created by the Quartus software. If you are having issues with your design, go through the compilation report and look for unconnected inputs or vector size mismatches.

Use the image of the Development Board in Figure 3.1 and the information in the board User Guide to determine the FPGA pins associated with the input and output devices used by the rpsGame module.

Table 3.5: Pin assignment tables for the Rock Paper Scissor game.

Segment	Player 1 Throw	Player 2 Throw	Win/Lose
seg[6]	PIN_Y18		
seg[5]		PIN_AC23	
seg[4]			
seg[3]			
seg[2]			
seg[1]			
seg[0]			PIN_AA18

	Player 1 Slide Switch	Player 2 Slide Switch
slide[2]		
slide[1]		
slide[0]	PIN_AC9	

Play Button	Key[0]	

Note, each push-button provides a high logic level when it is not pressed, and provides a low logic level when pressed.

3.9 Turn in

You may work in team of at most two. Make a record of your response to the items below and turn them in a single copy as your team's solution on Canvas using the instructions posted there. Include the names of both team members at the top of your solutions. Use complete English sentences to introduce what each of the following listed items (below) is and how it was derived. In addition to this submission, you will be expected to demonstrate your circuit at the beginning of your lab section next week.

onesToDense Module

- Complete Table 3.2 truth table for oneToDense module.
- link Canonical SOP expressions for the play[1] and play[0] functions.
- link Verilog code for the entire. module (courier 8-point font single spaced), leave out header comments.

playToSeven Module

- Complete Table 3.3.
- link Verilog code for the module (courier 8-point font single spaced), leave out header comments.

winLose Module

- Complete Table 3.4 truth table for winLose module.
- link Verilog code for the module (courier 8-point font single spaced), leave out header comments.

rpsGame Module

- link Verilog code for the module (courier 8-point font single spaced), leave out header comments.

Pin Assignment

- Completed pin assignment for 7-segment, slide switches and button in Table 3.5.
- Demonstrate your working design to a lab team member.

Laboratory 4

High Low Guessing Game

4.1 Outcomes and Objectives

The outcome of this lab is to instantiate a guessing game using common logic building blocks. on the FPGA development board. Through this process you will achieve the following learning objectives.

- Analyzing a word statement for a logic function
- Creating a truth table description for a logic function

4.2 The Guessing Game

The guessing game is a two-person game where, one player is the guesser and the other, an honest, secret keeper. The game starts with the secret keeper generating a *secret number* between [**0** and **15**], inclusive. Once the *secret number* is decided, the guesser makes a *guess*, a number in the interval [**0** to **15**] inclusive, and tells this to the secret keeper. The secret keep then replies to the guesser if *guess* is less than, equal to, or greater than the *secret number*. The game continues with repeated guesser/secret keeper exchanges until the guesser correctly identifies the *secret number*.

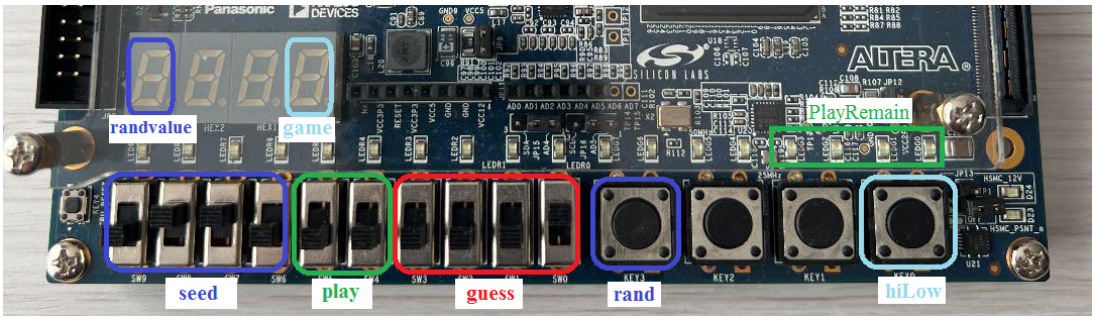


Figure 4.1: The input and output you should use to realize the High/Low Guessing game.

Your goal in this lab is to create a digital version of the guessing game using the development board using the inputs and outputs shown in Figure 4.1. In this case, the FPGA will play the role of secret keeper. You will enter a seed value using the **seed** slide switches. The seed value will be "randomized" into a 4-bit *secret number* using a linear feedback shift generator (more about this later). Pressing the **rand** button reveals the **4**-bit *secret number* as a **1** –digit hexadecimal value on the **randValue** 7-segment displays. Obviously, the guesser should not press the **rand** button during regular game play.

The player will make their guess about the secret number on the **guess** slide switches. This *guess* is compared to the *secret number* and the outcome is displayed on the **game** 7-segment display when the **hiLow** button is pressed. The **game** 7-segment display will show:

- 'H' when *guess* > *secret number*
- 'I' when *guess* = *secret number*
- 'L' when *guess* < *secret number*

A player is only allowed **4** guesses to get the secret number. To keep track of this, every time that the player makes a guess, they increment the binary number on the **play** slide switches. When a slide switch is in the up position, the bit value is 1 and when in the down position, the bit value is 0. This means that the player needs to understand how to count in binary. In order to make keeping track of the number of guesses remaining, the number of illuminated green **playRemaining** LEDs will equal the number of guesses left. For example, if the binary value set on the **play** slide switches equals **2**, then the right-most **2** green LEDs would be illuminated. You should illuminate LEDs starting from the right side and increasing towards the left side.

4.3 System Architecture

Use the system architecture shown in Figure 4.2 as your guide to this design. Please note that lines with the same name in different places are connected together. For example, the signal *randBut* connects the button input to the 2:1 mux in the upper left corner of the FPGA.

4.4 Module: 2:1 Mux

A 2:1 multiplexer, a mux for short, is a basic building block in many digital systems. The 2:1 mux shown in Figure 4.3 routes one of the two N-bit data inputs, $\mathbf{y_0}$ or $\mathbf{y_1}$ to the N-bit output, **F**, depending on the value of a 1-bit select signal, **s**. When $\mathbf{s} = 0$, $\mathbf{F} = \mathbf{y_0}$ and when $\mathbf{s} = 1$, $\mathbf{F} = \mathbf{y_1}$. In other word, **F** equals the **y** input whose subscript equals **s**.

You may notice that the data inputs of the 2:1 muxes in Figure 4.2 have their $\mathbf{y_0}$ and $\mathbf{y_1}$ data inputs denoted as '0' and '1' respectively. This is done to save space and increase clarity in the schematic.

The Verilog code for a 2:1 mux is provided to you on Canvas. When creating instances of the 2:1 mux, you will need to correctly order the signals in the module instantiation. To do this, follow the order shown in the module declaration shown in the top two lines in Listing 4.1.

Listing 4.1: Top, module definition for a 2:1 mux. Bottom, module instantation of a 2:1 mux in Figure 4.2.

```
// Module definition for the 2:1 mux
module genericMux2x1(y1, y0, sel, f);

// Module instantation for a 2:1 mux in the hiLow digital circuit
```

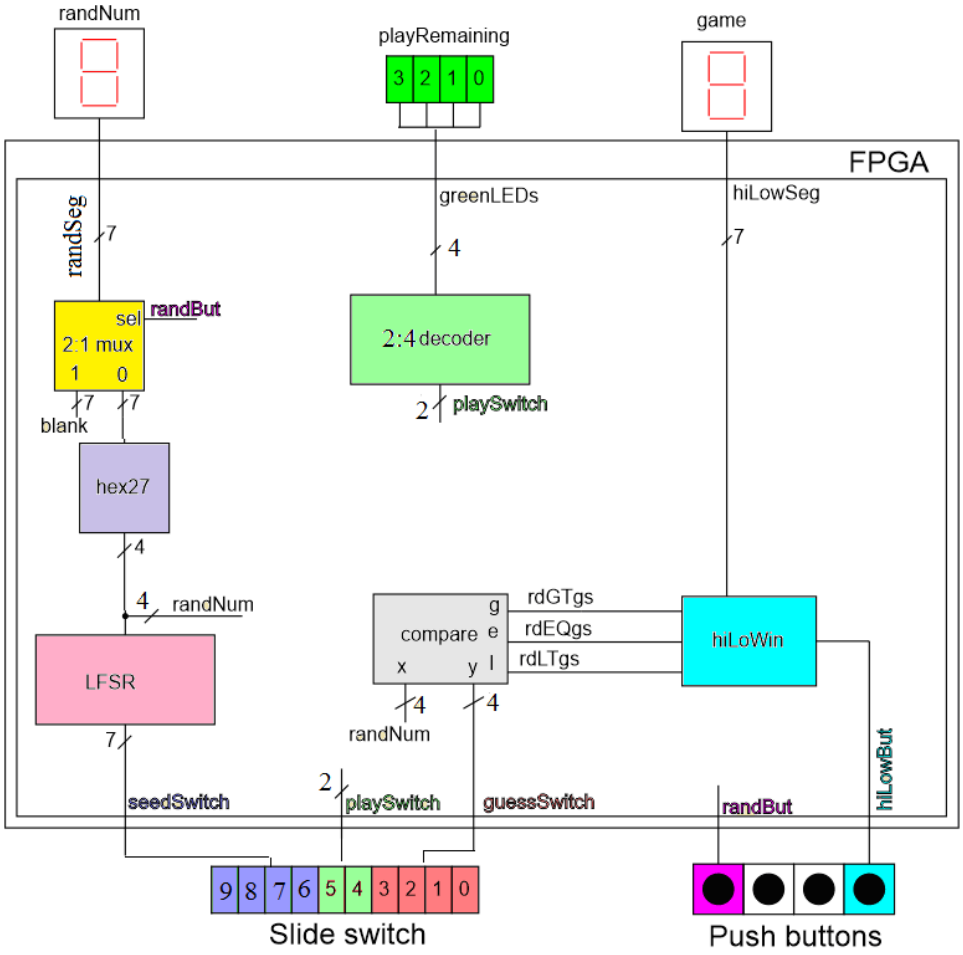


Figure 4.2: System architecture for the guessing game.

```
genericMux2x1 #(7) muxHex(7'b1111111, RandHex, randBut, randSeg);
```

The signal width, **N**, shown in Figure 4.3 is a placeholder for an integer value that describes the width of the *y1, y0, F* signals. You can specify this width when you instantiate a *genericMux2x1* module using the #() specifier immediately after *genericMux2x1* as shown in the bottom line of code in Listing 1.

A component that you can instantiate with different signal widths is called "generic" and often used in the module's name. Often generics have a default value, for the genericMux2x1 it is 8-bit. This is worth mentioning because if you forget to include #(7) in your instantation, the compiler will generate a warning that is easy to overlook and your design will not simulate or synthesize correctly. If you suspect this is occuring in your design, look in the Compilation Report tab -> Analysis & Synthesis folder -> Connectivity Checks folder. Click on the offending module and you will see the following error report.

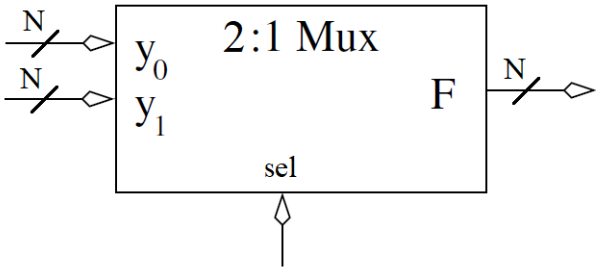


Figure 4.3: The schematic representation of a 2:1 mux.

Compilation Report - hiLow

Port Connectivity Checks: "genericMux2x1:muxMsbHex"

	Port	Type	Severity	Details
1	y1	Input	Info	Stuck at VCC
2	f	Output	Warning	Output or bidir port (8 bits) is wider than the port expression (7 bits) it drives; bit(s) "f[7..7]" have no fanouts
3	y1	Input	Warning	Input port expression (7 bits) is smaller than the input port (8 bits) it drives. Extra input bit(s) "y1[7..7]" will be connected to GND.
4	y0	Input	Warning	Input port expression (7 bits) is smaller than the input port (8 bits) it drives. Extra input bit(s) "y0[7..7]" will be connected to GND.

Figure 4.4: Forgetting the generic specifier in a 2:1 Mux will generate this report.

4.5 Module: Compare

A N-bit comparator is a basic building block in many digital systems. The N-bit comparator shown in Figure 4.5 checks the relative magnitude of the two N-bit inputs **x** and **y** and sets one of the three outputs equal to 1, one's-hot output, depending on their relation to each other.

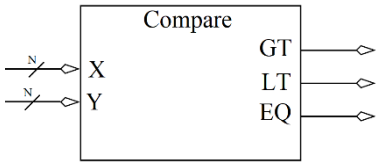


Figure 4.5: A schematic representation of a N-bit comparator.

The relationship between the inputs and outputs is given in the following list. Note that the order of the inputs is important as **X** is always on the left side of the relational operator.

- **GT** = 1 when **X** > **Y** else **GT** = 0
- **EQ** = 1 when **X** == **Y** else **EQ** = 0
- **LT** = 1 when **X** < **Y** else **LT** = 0

The Verilog code for the N-bit comparator is available on Canvas. When creating instances of the comparator, you will need to correctly order the signals in the module instantiation. To do this, follow the order shown in the module definition shown in the top two lines in Listing reflisting:comparatorVerilog.

Listing 4.2: Top, the module definition for the comparator. Bottom, module instantiation of a comparator in Figure 4.5. Remove the component instantiation line break in your code.

```
// Module definition for the comparator
```

```
module genericComparator(x, y, gt, eq, lt);

// Module instantiation for a compataror in the hiLow digital circuit
genericComparator #(4) randVsGuess(randNum, guessSwitch, \
                    randGTguess, randEQguess, randLTguess);
```

Like the mux, the comparator is a generic module. This means that you need to specify the width of the **X** and **Y** vectors using the #() specifier. The same warnings about vector size mismatch applies to comparators.

4.6 Module: hexToSevenSeg

You should use the hexToSevenSeg module you developed in a previous lab. Note, the name of this module was shortened in Figure 4.2 to hex27 in order to save space and make the schematic more readable.

4.7 Module: 2:4 Decoder

The module labeled 2:4 decoder interprets the 2-bit input s_1, s_0 as a 2-bit binary number that we will call **s**. All the y outputs whose subscript is less than or equal to **s** will have an output of 1. All the y outputs whose subscript is greater than **s** will have an output of 0.

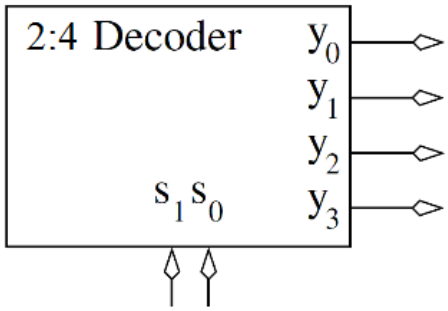


Figure 4.6: A 2:4 decoder.

The first few rows for the truth table for the 2:4 decoder are shown in Table 4.1.

Table 4.1: Partial truth table for the 2:4 decoder.

s_1	s_0	y_3	y_2	y_1	y_0
0	0	1	1	1	1
0	1	0	1	1	1
1	0	0	0	1	1

While this implementation may look odd, it converts the user's selection on the **play** slide-switches to show the correct number of plays remaining for the user on the LEDs.

You should implement the 2:4 decoder in the hiLow module using an always/case statement similar to the one used to implement your hexToSevenSeg. You should put this Verilog code

in the hiLow module as a (large) concurrent statement. **This means that you should not have a separate Verilog file for the 2:4 decoder.** Remember that the output type from an always/case statement must have the "reg" qualifier, not "wire".

4.8 Module: hiLowWin

The hiLowWin functionality converts the output from the comparator into the illuminated patterns shown in Figure 4.7 when the **hiLow** button is pressed. The "I" from "wIn" is needed because you cannot make a "W" on a 7-segment display.

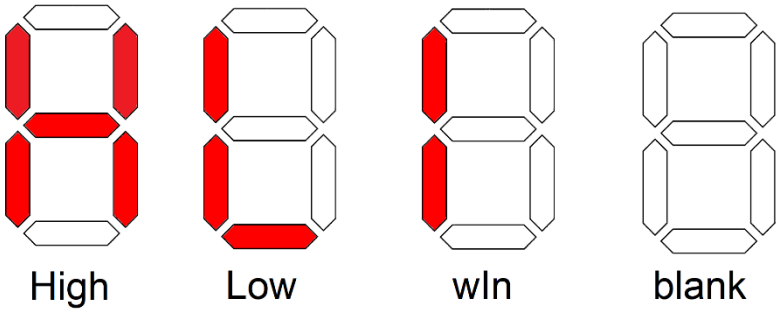


Figure 4.7: The illuminated patterns to inform the guesser about the magnitude of their guess.

You should implement hiLowWin inside the hiLow module using an always/case statement similar to the one used to implement your hexToSevenSeg. You will need to create a vector out of the 4-separate inputs using the parenthesis operator as shown in Listing 4.3. Note that the code shown Listing 4.3 is incomplete.

Listing 4.3: Starter code for the hiLowWin module.

```
always @(*)
        case ({hotColdBut, hotWire, warmWire, coldWire})
                4'b0001: hotColdSeg = 7'bxxxxxxx;
                default: hotColdSeg = 7'bxxxxxxx;
        endcase
```

You should put this Verilog code in the hiLow module as one of the many concurrent statement. This means that you should not have a separate Verilog file for the hiLowWin. Remember that the output type from an always/case statement must have the "reg" qualifier, not "wire".

4.9 Module: LFSR

A linear feedback shift register (LFSR) is a digital circuit that generates a pseudo-random sequence of numbers starting from a seed value. Since we do not yet have storage devices in our class, we will implement a LFSR that performs a single iteration of the randomization step as shown in Figure 4.8.

Figure 4.8 shows the input bits $I_2 \ldots I_0$ being shifted one bit to the left on their way to the outputs $O_3 \ldots O_1$. The output O_0 is formed by computing $I_2 \hat{} I_0$ where "^" is the xor operation.

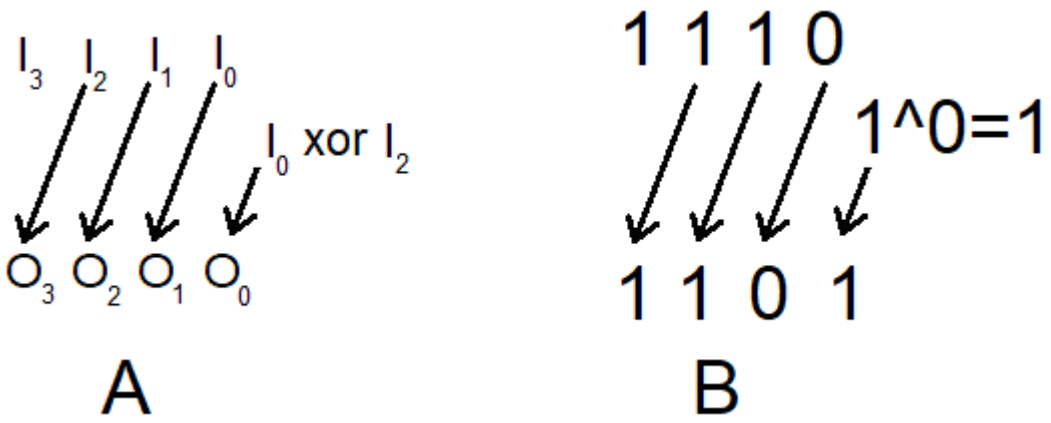


Figure 4.8: A) A schematic illustration of a 4-bit LFSR operation. B) The input 4b'1110 produced the output 4b'1101.

Let's use Table 4.2 to understand what happens if the input in Figure 4.8 was 7'b1110, which, when interpreted as a decimal number, is 14. The upper 3-bits of output are formed by shifting the input left by one bit. The least significant bit of the output is formed by computing 1 ^ 0 which equals 1. The resulting output is 4'b1101, when interpreted as a decimal number, equals 13. Fill in the next blank row of Table 2 using decimal 13 as an input. Repeat for the last row of the table.

Table 4.2: The first iteration of the LFSR shown in Figure 4.8 when started at decimal 14.

O_3	O_2	O_1	O_0	decimal
1	1	1	0	14
1	1	0	1	13

If you continued the output from the shift operation performed in Table 4.2 you would eventually find a decimal number that repeats because there are only 16 different combinations of 4-bits. Call this repeat number the nexus. The length of the sequence of numbers a nexus back to itself is the length of the sequence. The length of the sequence generated by the operation in Figure 4.8 is 15. This means that if Table 2 had 15 rows and you filled them all in, you would get 14 on the 14^{th} row. Can you figure out what number is excluded from the sequence?

For the lfsr module, you need to:

- Use the module declaration:
  ```
  module lfsr(Seed, outputRand);
  ```
- Make the input and outputs vectors with wire type.
- Use 4 assign statements to give each bit of output a value.
- Complete the testbench for the lfsr module. Create timing diagram that asserts the four inputs listed in Table 4.2 waiting #20 between inputs. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Switch radix

to unsigned decimal for input and output (right click on signal name in wave pane and select radix -> unsigned).

4.10 Module: hiLow

The hiLow module stitches together all the modules and contains all the signals shown in Figure 4.2. The module declaration is provided below to assist your pin assignment.

```
module hiLow(seedSwitch, playSwitch, guessSwitch, randBut, hiLowBut,
   randSeg, greenLEDs, hiLowSeg);
```

To complete this module, you will need to instantiate all the modules in Figure 4.2. To provide guidance on this process let's focus on the 2:1 mux from Figure 4.9. This 2:1 mux is repsoduced in Figure 4.9.

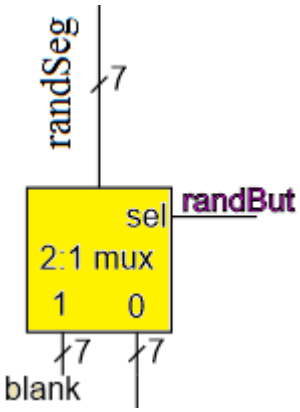


Figure 4.9: A small piece of hardware from Figure 4.2.

Let's write the Verilog code to instantiate the 2:1 mux. The first step that you need to take is to give EVERY signal in the system architecture a name or constant value. With respect to Figure 4.9, the output of the 2:1 mux is already named **randSeg** and the select line is named **randBut**. The data input y1 will have a constant value **7b'1111111**, needed to produce a blank 7-segment display. The input y0 is the output from a hexToSevenSeg module, a signal named **RandHex**.

The second step is to know the order of the parameters in the 2:1 mux module declaration. This was given earlier as:

```
module genericMux2x1(y1, y0, sel, f);
```

The third step is instantiating the 2:1 mux in Verilog. To do this:

- Define the width of the data input and data output of the mux (7-bits),
- Give the 2:1 mux instance a descriptive and unique name. For example, **muxHex**,
- Put the system architecture signals in their corresponding locations in the module

```
genericMux2x1 #(7) muxHex(7\textquotesingle b1111111, RandHex, randBut, randSeg)
```

Once you get the hang of it, you are just translating the system architecture of Figure 4.2 into words.

For the hiLow module, you should:

- Use the module declaration:

```
module hiLow(seedSwitch, playSwitch, guessSwitch, randBut, hiLowBut,
  randSeg, greenLEDs, hiLowSeg);
```

- Make the `seedSwitch, playSwitch, seedSwitch` inputs vectors with the left switch the MSB. You will need to keep this consistent with the pin assignment that you will compete next.
- The `randBut, hiLowBut` inputs are not vectors.
- Use a vector for the `randSeg` output with wire type.
- Use a vector for the `greenLEDs, hiLowSeg` output with reg type.
- My module had 3 internal vectors (wire type) and 3 internal one bit signals (shown in the system architecture).

4.11 Testbench

Run the testbench for the hiLow module provided on Canvas. Produce a timing diagram with the following characteristics. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Order the traces from top to bottom as shown in the following table.

signal	radix	color trace
t_seedSwitch	unsigned	green
t_guessSwitch	unsigned	green
t_playSwitch	unsigned	green
t_randBut	default	green
t_hiLowBut	default	green
LFSR output	unsigned	yellow
t_randSeg	hex	red
t_hiLowSeg	hex	red
t_greenLEDs	default	red

4.12 Pin-Assignment

Use the image of the development board in Figure 4.1 and the information in the board User Guide to determine the FPGA pins associated with the input and output devices used by the hiLow module.

Table 4.3: Pin-Assignment for the High Low Guessing Game.

Segment	randSeg	hiLowSeg
seg[6]	PIN_AC22	PIN_Y18
seg[5]		
seg[4]		
seg[3]		
seg[2]		
seg[1]		
seg[0]		

	seedSwitch	playSwitch	guessSwitch
slide[3]	PIN_AE19	N/A	
slide[2]		N/A	
slide[1]			
slide[0]			

randBut	Key[3]	
hiLowBut	Key[0]	

G[3]	G[2]	G[1]	G[0]

Complete the pin-assignment in Quartus, compile your design and download to the FGPA development boards. If you are having difficulty getting your circuit to work correctly, please refer to Section 4.14 for some useful debugging tips.

Once you get your design working, demonstrate it to a member of the lab team.

4.13 Turn in

You may work in teams of at most two. Make a record of your response to the items below and turn them in a single copy as your team's solution on Canvas using the instructions posted there. Include the names of both team members at the top of your solutions. Use complete English sentences to introduce what each of the following listed items (below) is and how it was derived. In addition to this submission, you will be expected to demonstrate your circuit at the beginning of your lab section next week.

Module: LFSR

- Link Verilog code for the body of the module (courier 8-point font single spaced), leave out header comments.
- A completed Table 4.2.
- Link Complete the testbench for the lfsr module. Create timing diagram that asserts the four inputs listed in Table 4.2 waiting #20 between inputs. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Switch radix to unsigned decimal for input and output (right click on signal name in wave pane and select radix -> unsigned).

Module: hiLow

- Verilog code for the body of the hiLow module (courier 8-point font single spaced), leave out header comments.
- Run the testbench for the hiLow module provided on Canvas. Produce a timing diagram according to this table. Zoom to fill the available horizontal space with the waveform.

Pin-Assignment

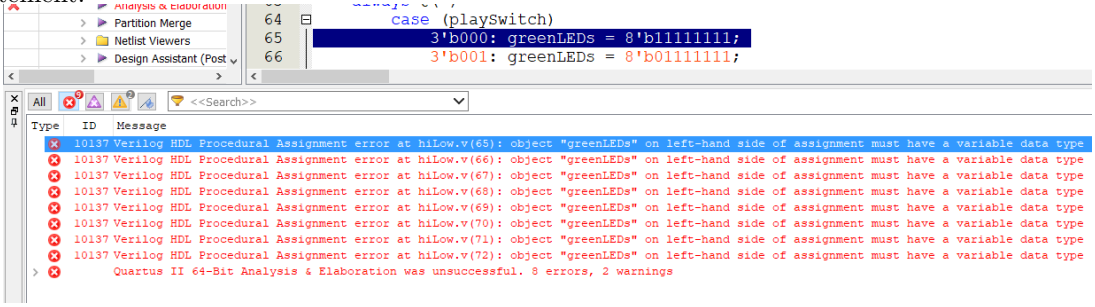
- Completed pin assignment table for all the signals in the hiLow module in Table 4.3.

4.14 Debugging Tips

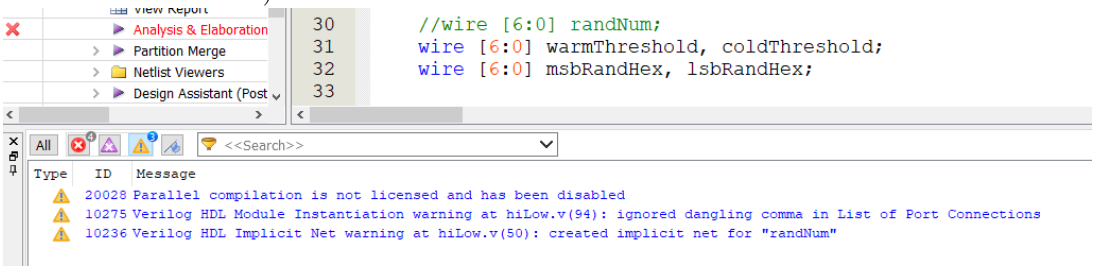
This laboratory typically generates a variety of new errors that you have not seen before. This section on useful debugging techniques will help you more effectively interpert the compilers output to locate errors in your code. The following is an example story of someone debugging their code...

After you put together all the components, you can run Start Analysis & Elaboration. It may take you a while to find all your errors. Try clicking on the Error icon (red x) or Warning icon (yellow triangle) in the console area, to eliminate a lot of the clutter.

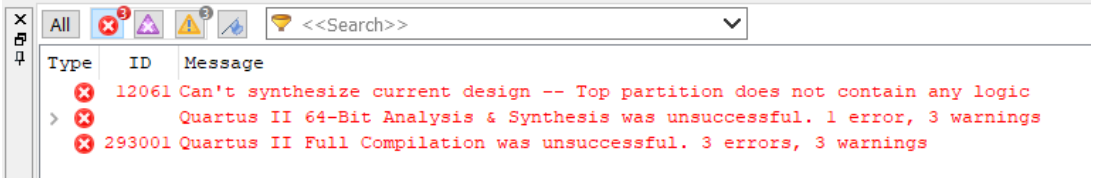
Th error below is a result of defining the output `wire {[}7:0{]} greenLEDs`. the output should have `reg{[}7:0{]} greenLEDs;` because greenLEDs is the output of an always/case statement.



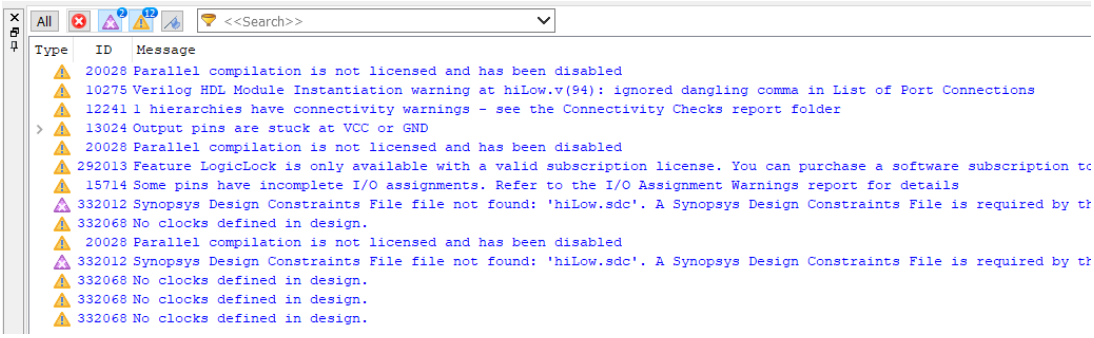
The following error is the result of forgetting to include the declaration of randNum. The top warning always appears and the second is a result of an unused output on the adder (more about this in the next lab).



Teh next error shows what happen when you accidentally leave a testbench as the top-level entity when attempting to synthesize.



These are all the Critical Warnings and Warnings that you will see on your final, working version. You should NOT attempt to fix these “errors”.



The Connectivity Checks folder from the Compilation Report will help you find weird connection problems that you may have inadvertently created in your design.