Contents

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

Laboratory 1

Introduction to Verilog

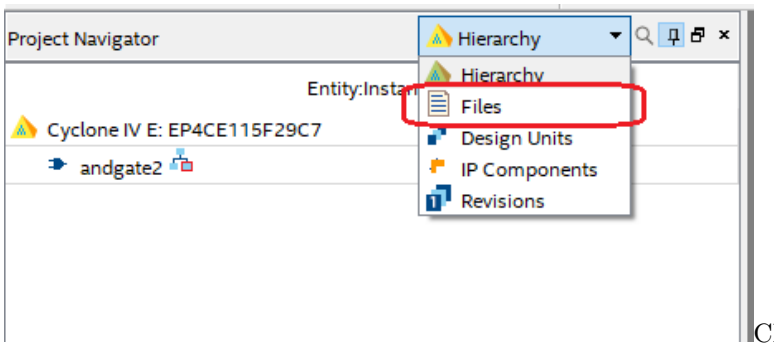
1.1 Objective

The objective of this lab is to introduce you to the Quartus II software, design entry using Verilog and circuit simulation.

1.2 Part 1: Setting up a project in Quartus and running a testbench

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 18.1 (64-bit).
    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,
    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*.

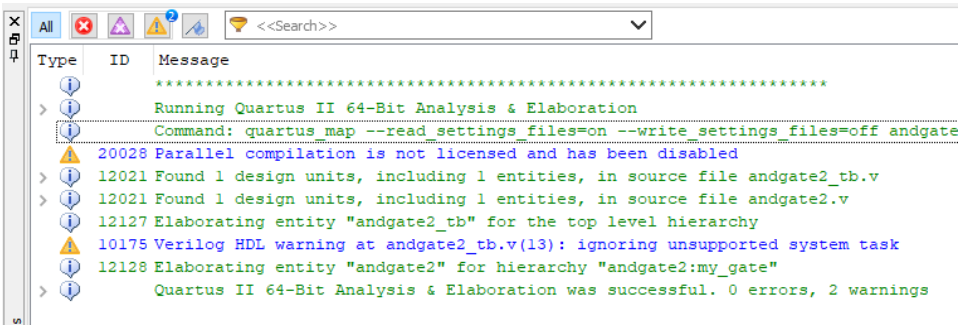
15. If you added the Verilog file in the correct directory and included it in the project, a Verilog file should pop up on the right.

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.

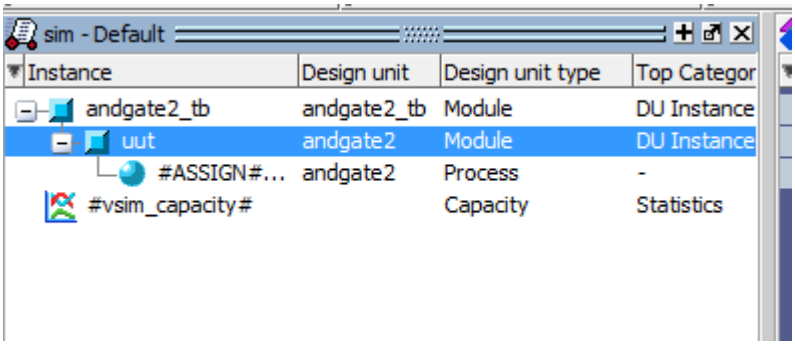


18. In the main Quartus II window, click *Tools -> Run Simulation Tool -> RTL Simulation.* The ModelSim program will launch. This may take a few moments, be patient. If you get a pop-up Nativelink Error window, then go back and check and fix the directory in step 11.

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.*

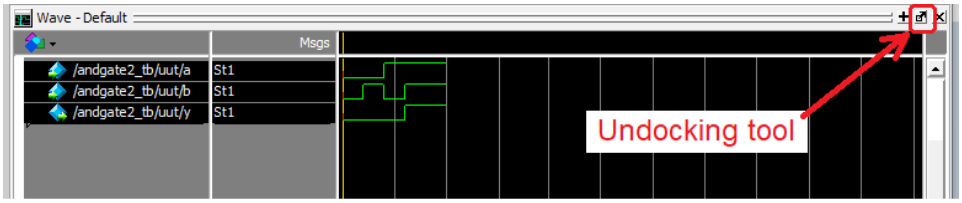
21. In the sim pane, right mouse click on uut and select *Add Wave.*



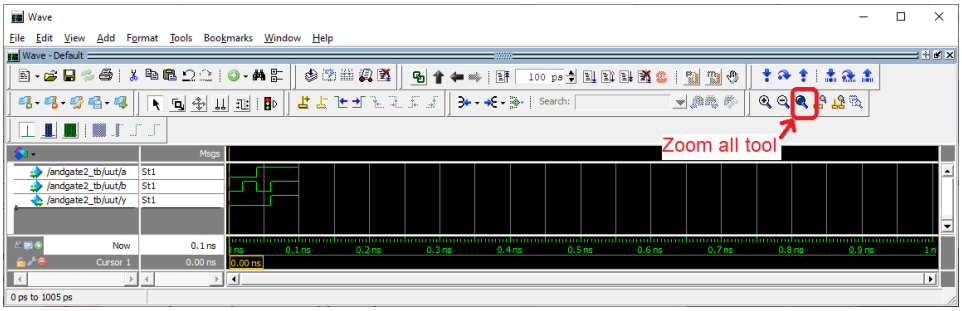
22. Choose *Simulate -> Run -> Run 100*. You should see inputs and output from andgate2. If you see only a small green portion of the waveform on the left margin of the timing diagram, you will need to zoom in on the waveform as follows. First click somewhere in the timing diagram (area under "Undocking 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 Part 2: 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 Part 3: 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.

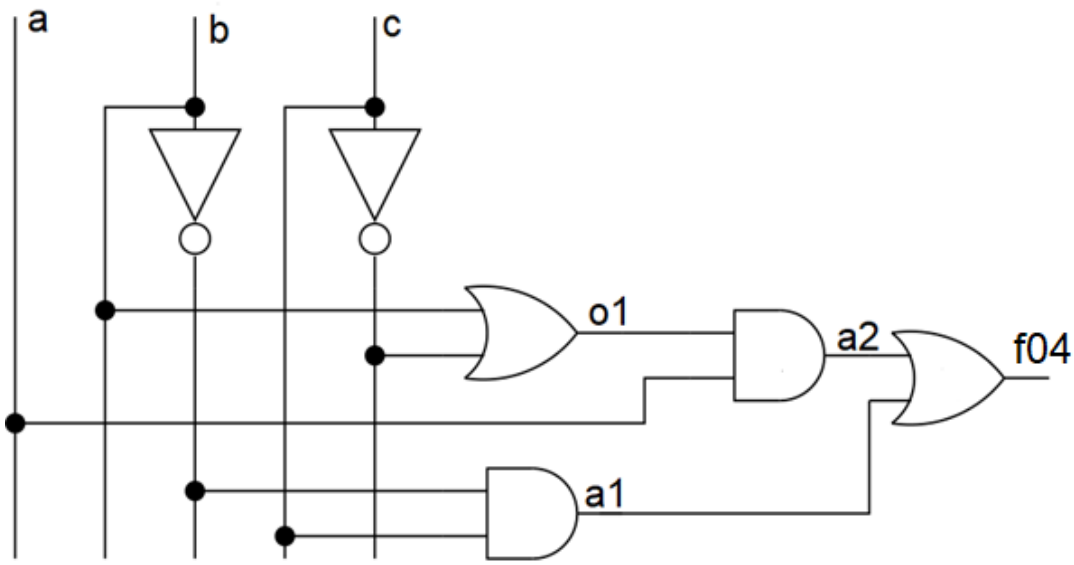
7. From the information in the timing diagram, produce a truth table.

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 Part 4: 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.

Part 1: Step 23 Timing diagram of AND gate
Part 2: Step 4 Verilog code for *f02*
Part 2: Step 7 Timing diagram of *f02*
Part 2: Step 8 Truth table of *f02*
Part 3: Step 6 Timing diagram of *f03*
Part 3: Step 7 Truth table of *f03*
Part 3: Step 8 Circuit Diagram of *f03*
Part 3: Step 9 Symbolic form of *f03*
Part 4: Step 4 Just the 4 Verilog assign statement for *o1*, *a1*, *a2*, and *f04*.
Part 4: Step 7 Timing diagram of *f04*
Part 4: Step 8 Truth table of *f04*

Laboratory 2

Hexadecimal to Seven-Segment Converter

2.1 Objective

The objective of this lab is to become familiar with the always statement used to implement truth tables, how to combine bits into vectors and how to download synthesized code onto the development board.

Today's laboratory will require to learn about two new Verilog concepts, The Always statement and Vectors. Let's look at the easier of these two, Vectors, first.

Vectors

A vector is simply a collection of bits, very similar to an array in a regular programming language. You might use a vector to represent a 3-bit binary number that you want to perform an operation on. 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 1.

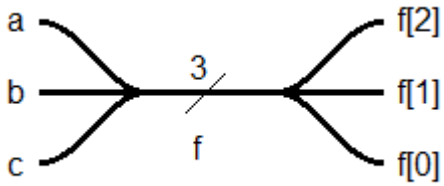


Figure 1: A schematic illustration of combining bits into a vector, f, and then accessing the individual bits of f.

Let's explore these ideas with the code snipped show in Figure 2. In this code snipped, the line of code assign f = {a,b,c}; combines the individual signals a, b and c into a 3-bit vector. The left-most signal in the parenthesis list becomes the MSB of the vector and the right-most becomes LSB. In other words, a is the MSB and c the 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.

The statement wire [2:0] f; is how you define a vector. 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. While I am not sure what the Verilog programmer was going for in this statement, you can access the individual bit of a vector by putting the index of that bit in square brackets. You can also access sub-vector by putting indices in square brackets separated by a colon.

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

Always

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

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 (in this 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 Figure 3 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.

2.2 Part 1: Combine lab 1 functions.

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 below.

Table 2.1: Table 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.

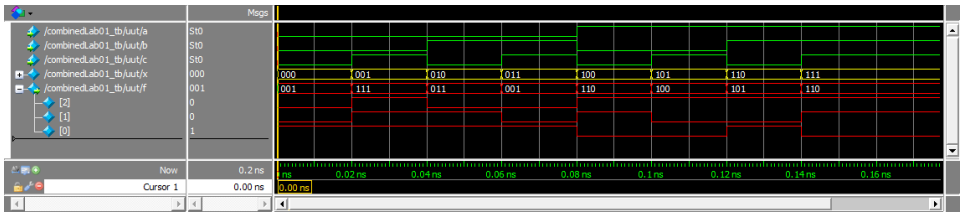
3. Create a project for these two files using the steps from last week's lab. For your convenience, these are given at the end of this document in the section entitled **Creating a Project**.

4. Modify *combinedLab01.v* so that *combinedLab01* outputs the values given in Table 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 this test bench using the steps from last week's lab. For your convenience, these are given at the end of this document in the section entitled **Performing a Simulation**.

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 similar (exactly?) the same as Figure 4.

Bridging the divide between logical and physical

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 1 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 to 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.

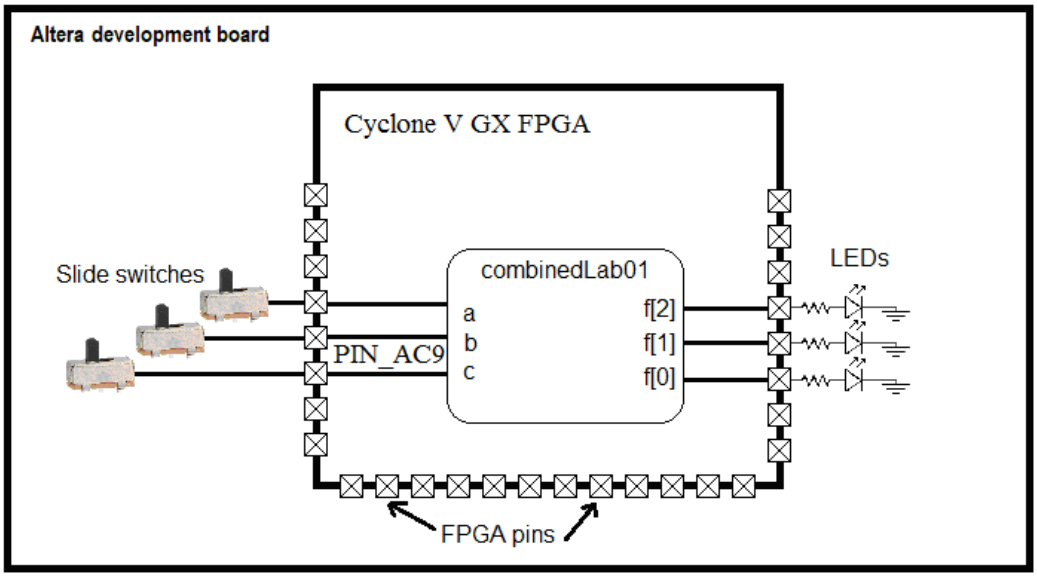


Figure 5: A simple Verilog design synthezied and downloaded onto the development board.

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 1 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 User Manual posted on the class Canvas page. Go to page **32** of the User Manual and find Table 3-3. It shows that slide switch SW[0] is connected to PIN_AC9.

Table 4-1 Pin Assignments for Slide Switches

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

The LEDs shown in figure 4-9 in the User Manual are active high, meaning that the LED is active (illuminates) when you send it a high signal (logic 1). Clearly, sending the LED a logic 0 turns the LED off. You can place the slide switches in one of two positions (up or down). In the up position, they assert a logic 1 on their input pin. Down causes the slide switch to assert a logic 0 on its input pin.

Use the information to complete the following table. We will call this the "pin assignment table" and use it in the next section.

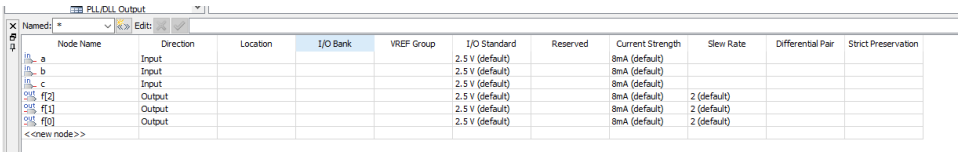
Table 2.2: Table 2: Pin Assignment Table for combinedLab01.

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			

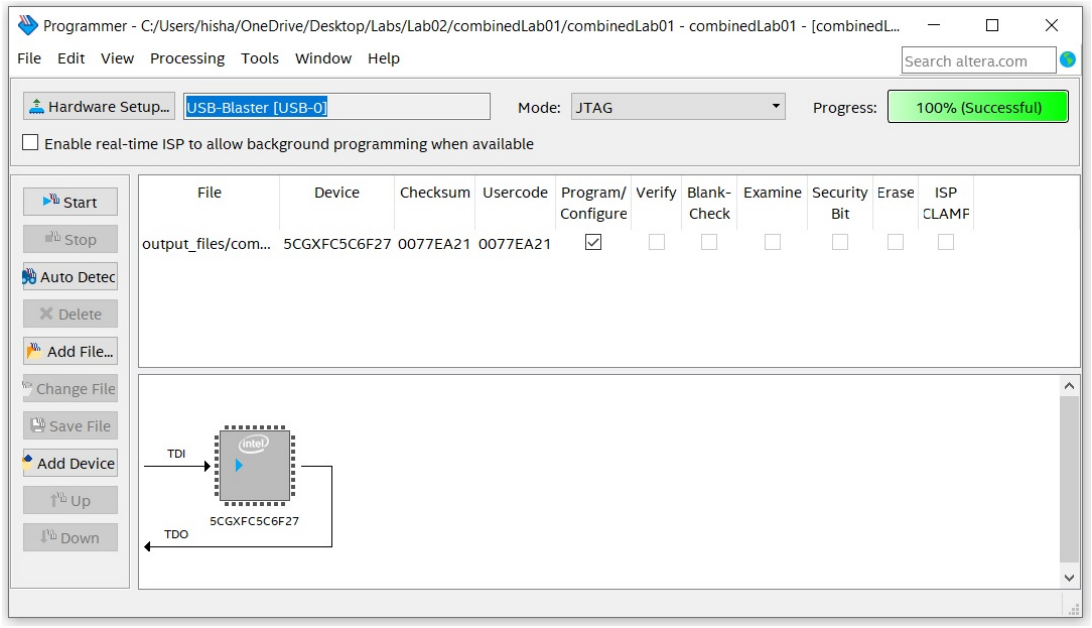
Synthesizing a Verilog Module

It's time to realize the *combinedLab01* Verilog file to FPGA. To do this follow these steps:

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.

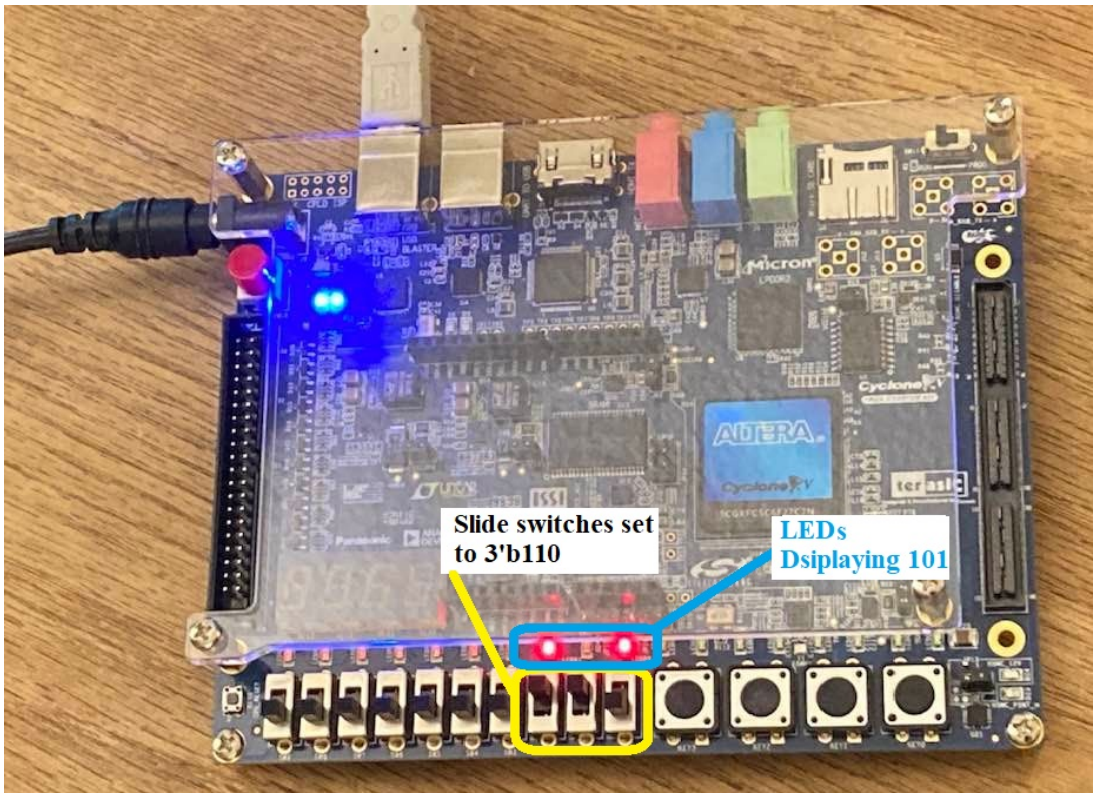


Figure 6: The Development board properly configured and running the combinedLab01 Verilog file.

2.3 Part 2: 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.

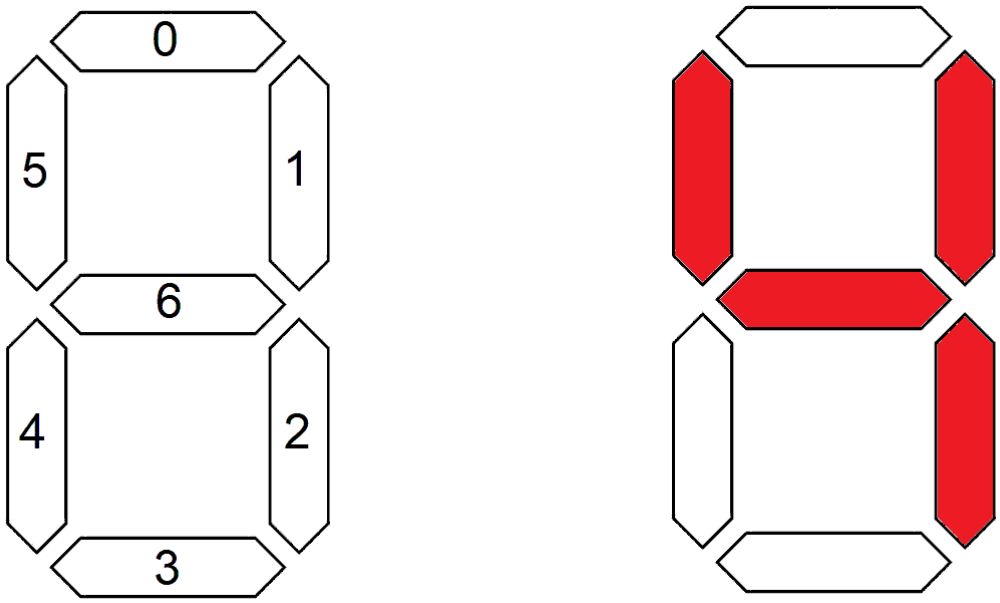


Figure 7: Left, the proper numbering of the segments. Right, illuminating segments to form the number 4.

The pattern of segments to be illuminated for each digit is shown in Figure 7. 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 8 shows the proper formatting for all the values between 0 – f.

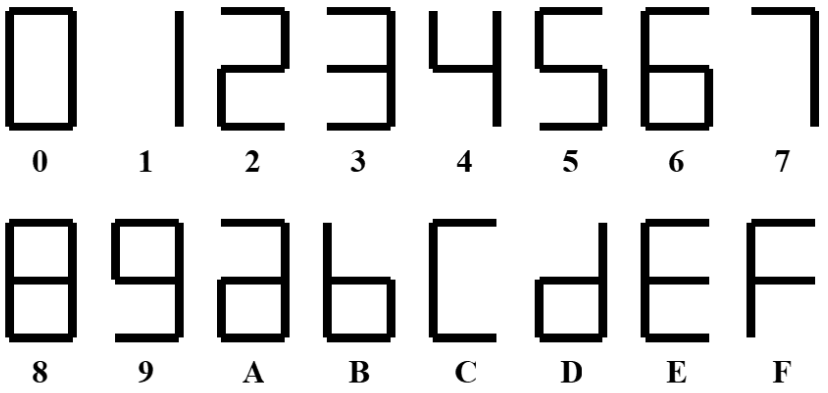


Figure 8: The proper arrangement of LEDs to form hexadecimal characters.

Figure 9 shows the Verilog module you will be building in this lab - a circuit that converts a 4-bit input, representing a hexadecimal value, into the binary values to illuminate a 7-segment display with active low LEDs.

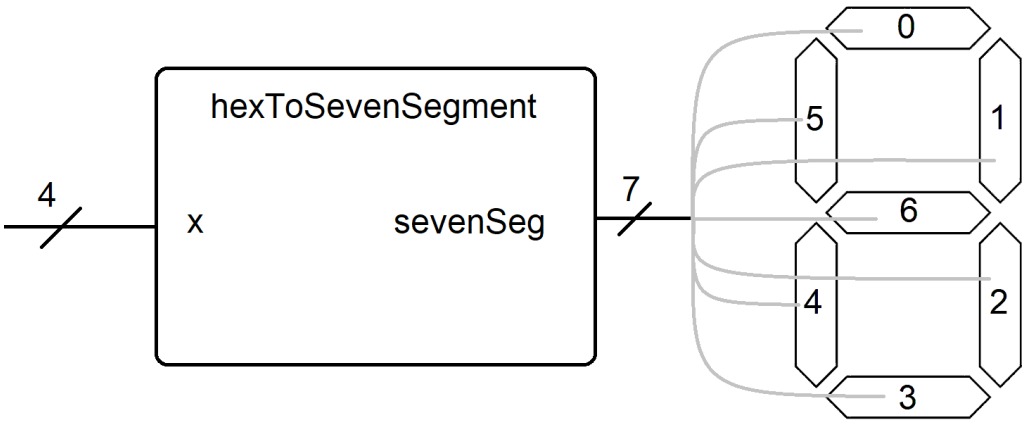


Figure 9: The hexToSevenSeg Verilog module driving a 7-segment display.

1. Complete the following table to illuminate the active low led segments to generate proper hexadecimal characters. I've renamed the sevenSeg output "seg" in the following table in order to make everything fit nicely.

Table 2.3: Table 3: Truth table or 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							

Complete the pin assignment tables for the inputs and outputs

Table 2.4: Table 4: Pair of pin assignment tables for the hexTo-SevenSeg component.

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

2. Create **a new project** folder within your *lab2* directory called *hexToSevenSeg.*

3. Download *hexToSevenSeg.v* and *hexToSevenSeg _tb.v* from Canvas to the project directory.

4. Create a project for these two files.

5. Complete the case statement for *hexToSevenSeg.v*

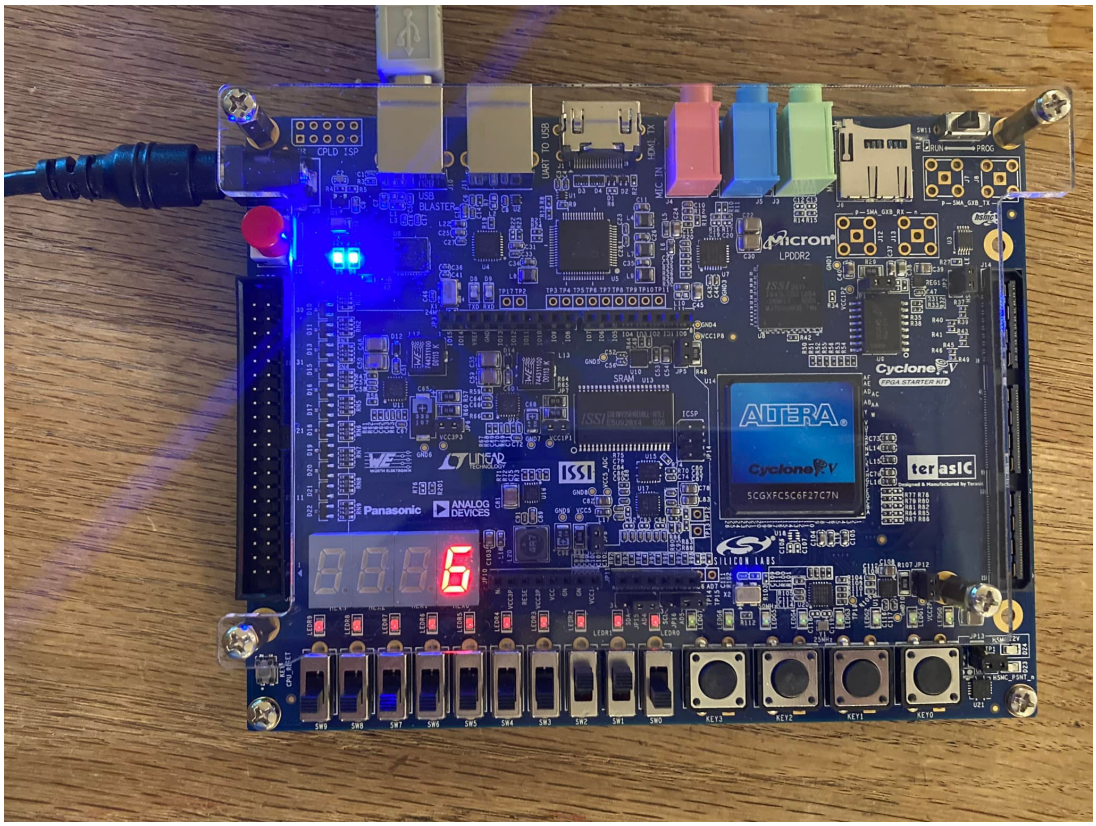
6. Modify *hexToSevenSeg _tb.v* so that *hexToSevenSeg* is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0,0 and going to 1,1,1,1.

7. 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.

8. 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.

9. From the information in the timing diagram, produce a truth table for *hexToSevenSeg.*

10. Synthesize your design, bask in the glow of another success.



2.4 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.

Part 1: Truth Table for combinedLab01 function (Table 1)
Timing diagram for combinedLab01 function
Pin assignment for combinedLab01 (Table 2)
Part 2: Truth Table for hexToSevenSeg function (Table 3)
Verilog code for hexToSevenSeg function – just the always/case statement
Timing diagram for hexToSevenSeg function
Pin assignment tables for hexToSevenSeg (Tables 4)
Demonstrate operation in lab

Laboratory 3

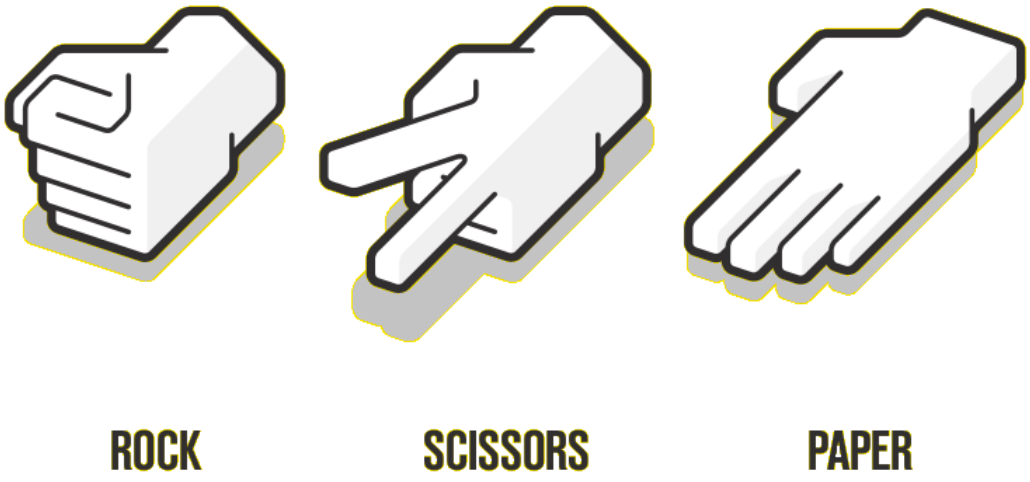
Rock Paper Scissors

3.1 Objective

The objective of this lab is to use the concepts learned to build a digital system to play a game of rock, paper, scissors.

Rock Paper, Scissors

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. You goal in this lab is to create a digital version of rock, paper, scissors using 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).

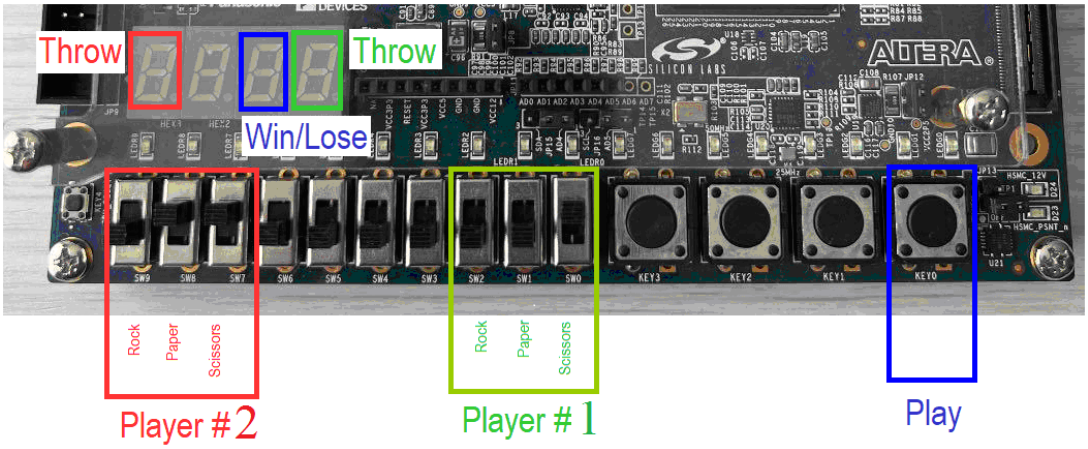


Figure 3.1: The input and output you should use to realize your digital system.

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.

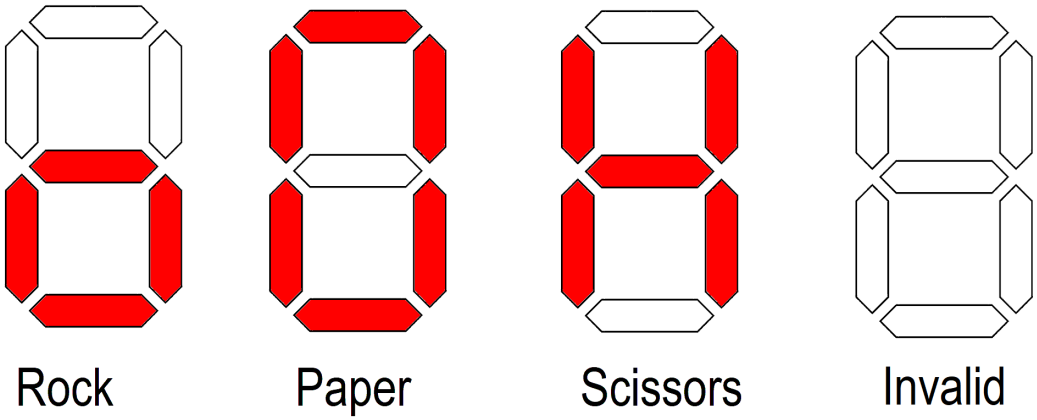


Figure 3.2: Illuminated patters for the different plays.

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

System design

There are an almost unlimited number of ways that you could implement this digital system. Since this is your first lab building a system, you must use the system architecture shown in Figure 3.3. 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.

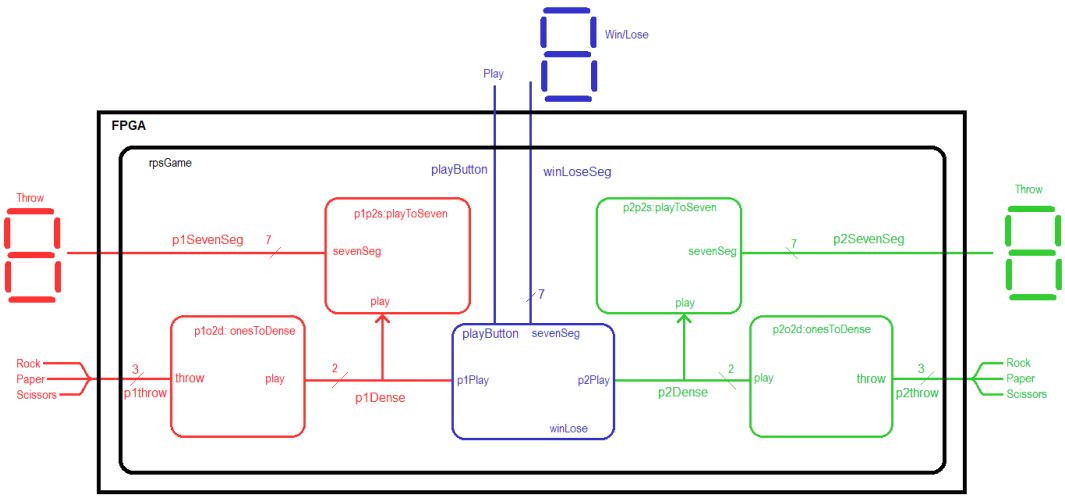


Figure 3.3: System architecture for the rock, paper, scissors system.

3.2 onesToDense Module:

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 function.

- 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 function.

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 your kmap. 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.

3.3 playToSeven Module:

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.

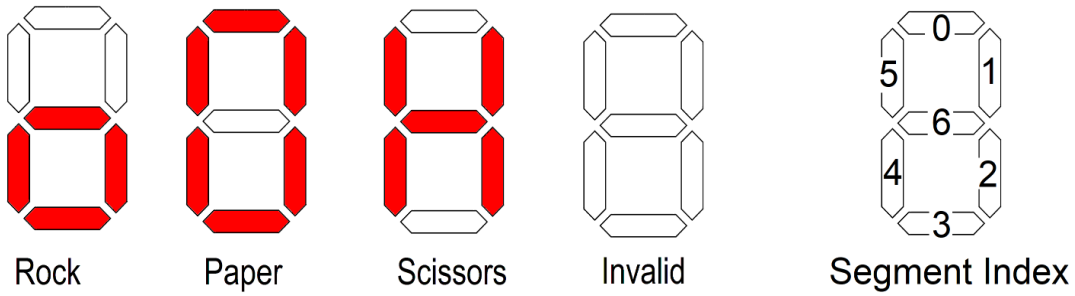


Figure 3.4: 7-segment display patterns for the different throws along with the 7-segment display segment indices.

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 rock is coded 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: Table to determine the bit values for 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.

3.4 winLose Module:

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.

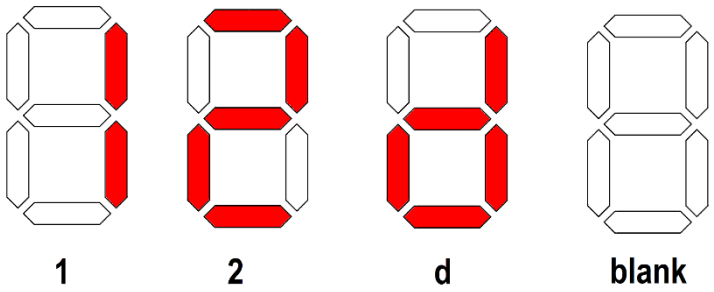


Figure 3.5: The illuminated patterns displayed on the win/lose 7-segment display.

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				
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.

- 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

  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 (last in the list) 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.

3.5 rpsGame Module:

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.
- 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);`

3.6 Pin Assignment:

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.

3.7 Turn in:

You may work in tem 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:

- link Complete Table reftable:onesToDense 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:

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

winLose Module:

- link 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:

- link Completed pin assignment table for 7-segment, slide switches and button.