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.

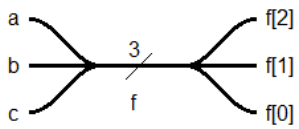


Figure 2.1: An illustration showing three bits combined into a vector, `f`, and then accessing the individual bits of `f`.

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 2.2 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

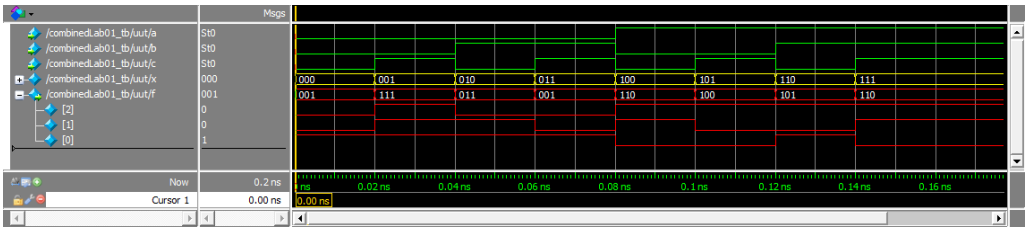
Let's explore vectors and the always statement by combining the three functions created in last weeks assignment into one function.

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.



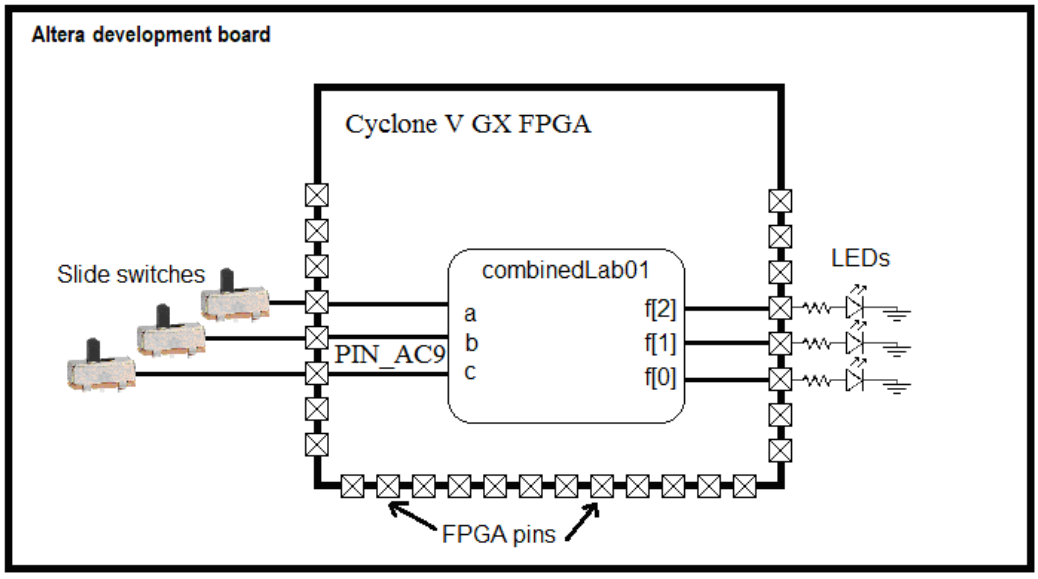
8. From the information in the timing diagram, produce a truth table. Compare the truth table generated from the data in the timing diagram to that you generated in Table 2.1.

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.

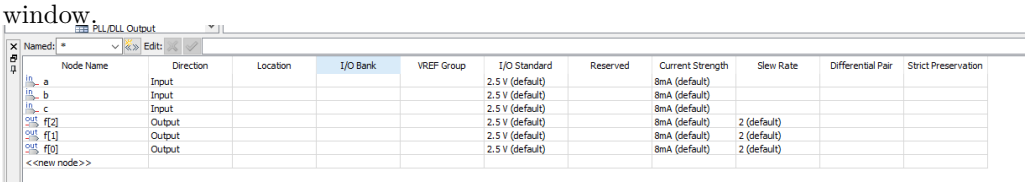
Table 2.2: Pin Assignment Table for combinedLab01.

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

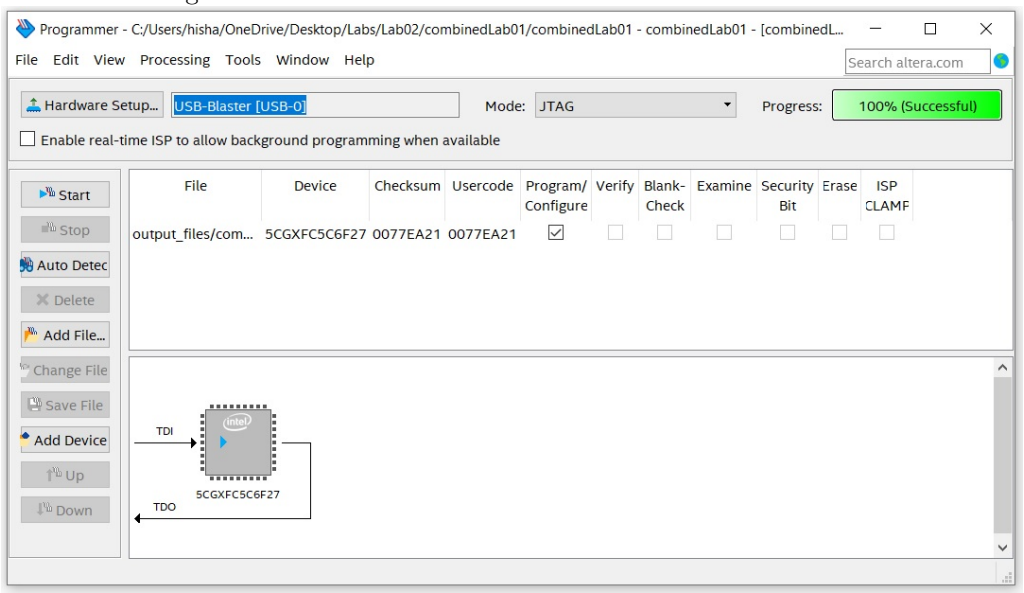
It's time to complete the synthesis process by downloading the combinedLab01 Verilog file along with its pin-assignment to the FPGA. Work through the follow these steps to accomplish this.

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.

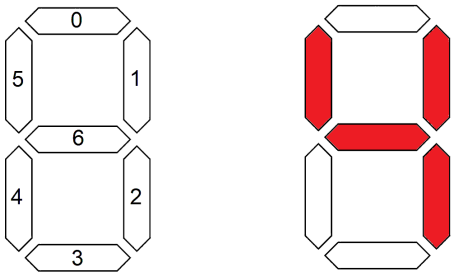


Figure 2.3: Left, the proper numbering of the segments. Right, illuminating segments to form the number 4.

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.

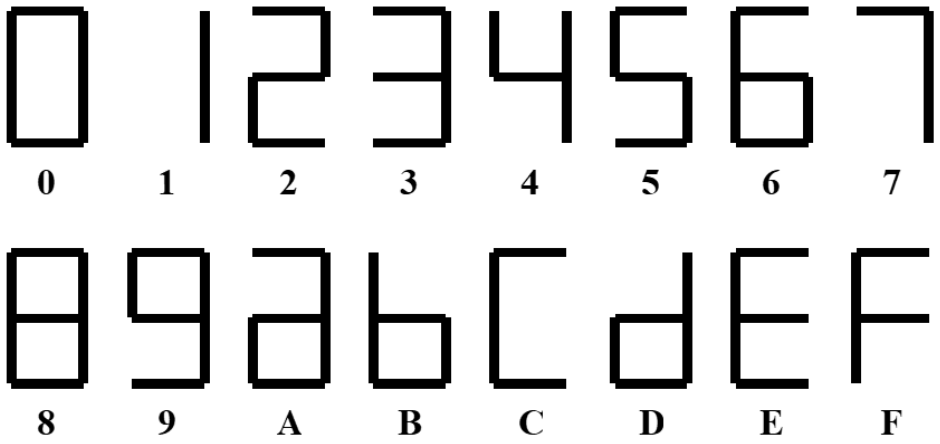


Figure 2.4: The proper arrangement of LEDs to form hexadecimal characters.

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*

2.8 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.*

2.9 Pin-Assignment and 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.

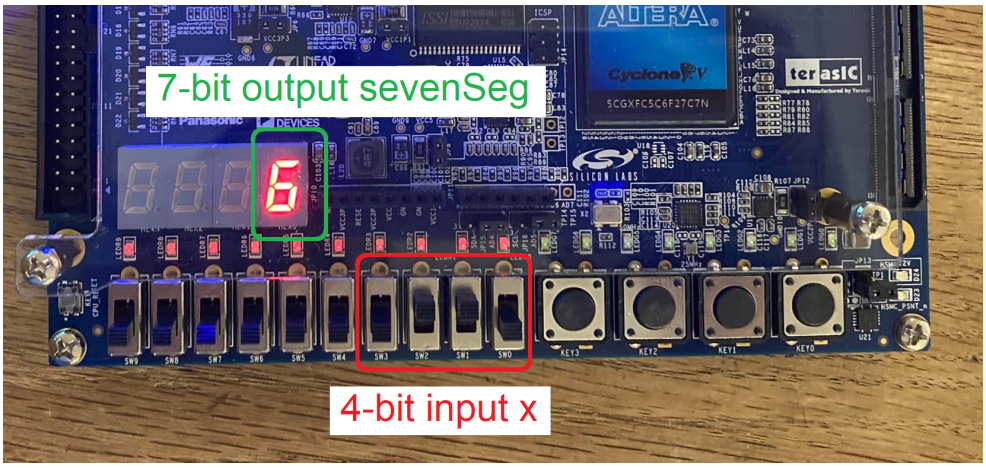


Figure 2.5: The input set to 4'b0110 displaying a 6 on the 7-segment display.

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.

2.10 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.