Contents

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

Laboratory 1

Hexadecimal to Seven-Segment Converter

1.1 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 1.1.

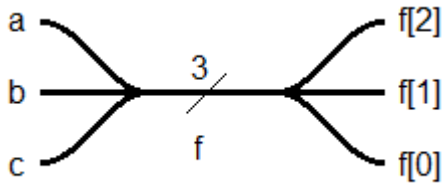


Figure 1.1: A schematic illustration of combining bits into a vector, f, and then accessing the individual bits of f.

Let's explore these ideas with the code snipped show in Listing 1.1. 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.

Listing 1.1: 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; 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. Listing 1.2 shows an always statement that uses the value of a signal x to compute the value of f.

Listing 1.2: A 3-input, 3-output function realized with an always statement.

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

    always @(*)
        case (x)
            3'b000: f = 3'b000;
            3'b001: f = 3'b000;
            3'b010: f = 3'b000;
            3'b011: f = 3'b000;
            3'b100: f = 3'b000;
            3'b101: f = 3'b000;
            3'b110: f = 3'b000;
            3'b111: f = 3'b000;
        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 (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 Listing 1.2 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.

1.2 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 shown in Table 1.1.

Table 1.1: 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.

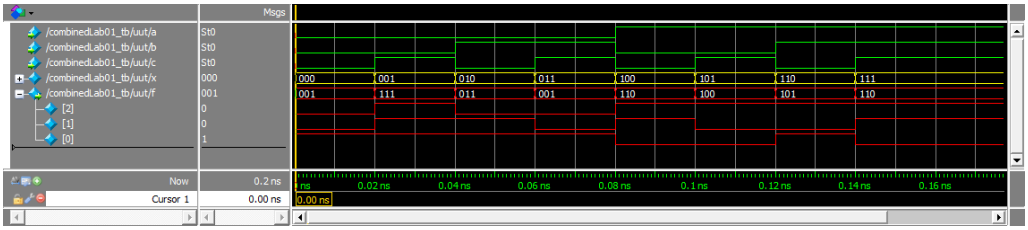
3. Create a project for these two files using the steps from last week's lab.

4. Modify *combinedLab01.v* so that *combinedLab01* outputs the values given in Table 1.1.

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.

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 1.1.

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 1.2 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 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.

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 1.2 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 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 pin assignment in Table 1.2. We will use this assignment in the next section.

Figure 1.2: A simple Verilog design synthesized and downloaded onto the development board.

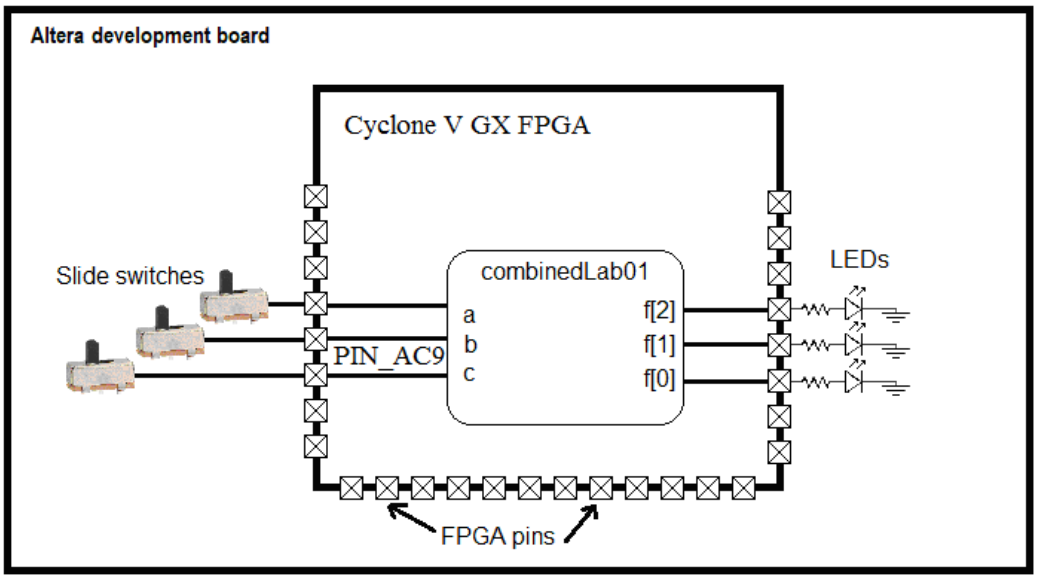


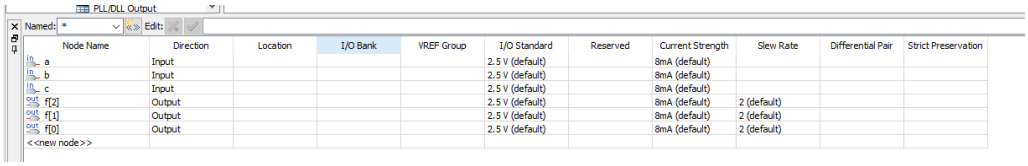
Table 1.2: Pin Assignment Table for combinedLab01.

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			

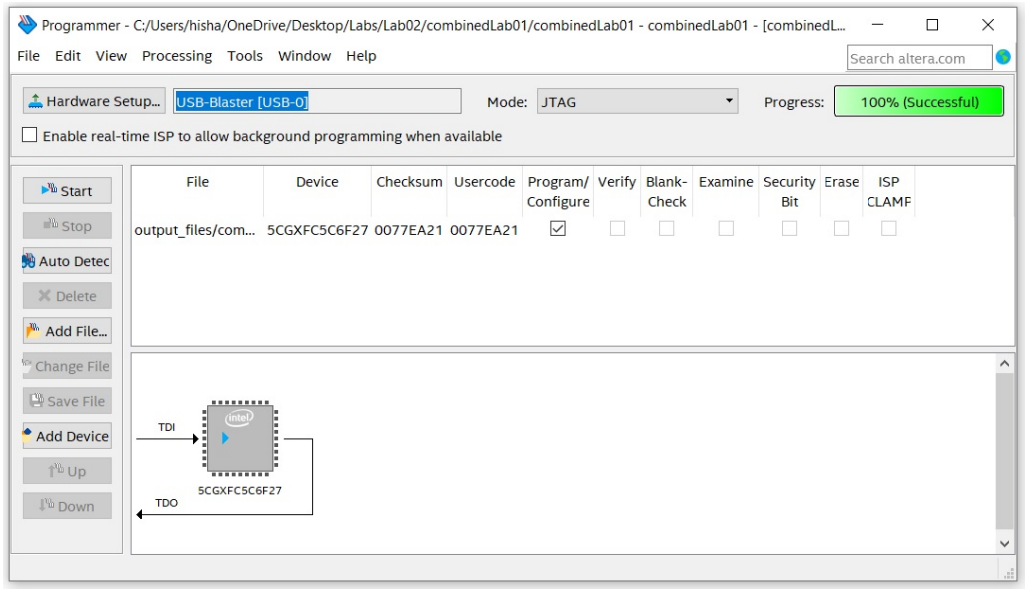
Synthesizing a Verilog Module

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....

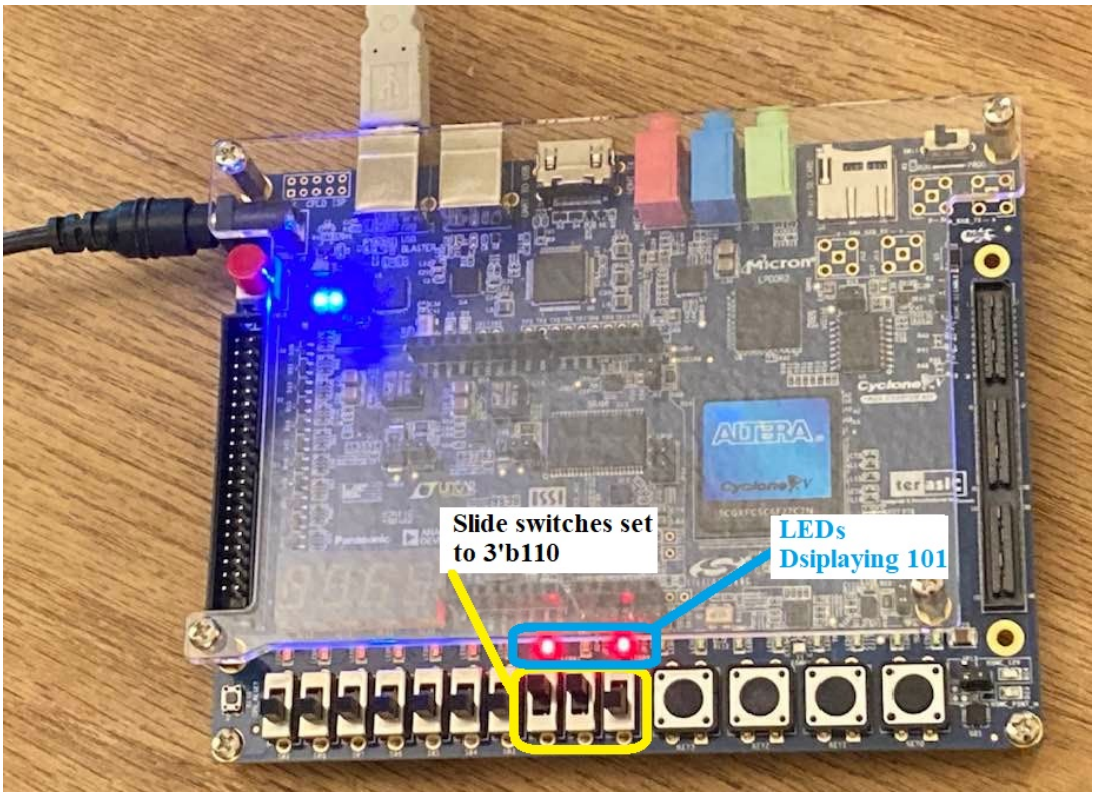
b. In the Hardware Setup select USB-Blaster [USB=0] from the Currently selected hardware pull-down

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 1.3: The development board properly configured and running the combinedLab01 Verilog file.



1.3 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.

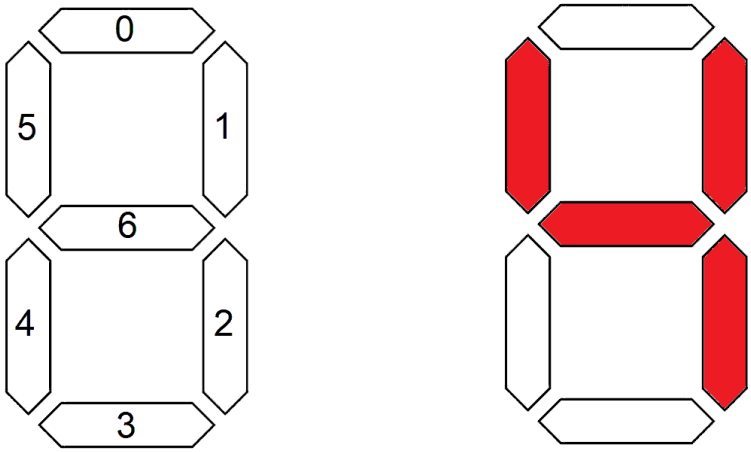


Figure 1.4: Left, the proper numbering of the segments. Right, illuminating segments to form the number 4.

The pattern of segments to be illuminated for each digit is shown in Figure 1.4. 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 1.5 shows the proper formatting for all the values between 0 – f.

Figure 1.6 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.

1. Complete the Table 1.3 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 1.3: 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							

x	seg[6]	seg[5]	seg[4]	seg[3]	seg[2]	seg[1]	seg[0]
0110							
0111							
1000							
1001							
1010							
1011							
1100							
1101							
1110							
1111							

Complete the pin assignment tables for the inputs and outputs of the hexadecimal to seven segment converter given in Table 1.4.

Table 1.4: Pair of pin assignment tables for the hexToSevenSeg 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.							

Now you are ready to write the Verilog code for the hexadecimal to seven segment converter, assign pins to the inputs and outputs, and synthesize and download the module to the development board.

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*
5. 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.

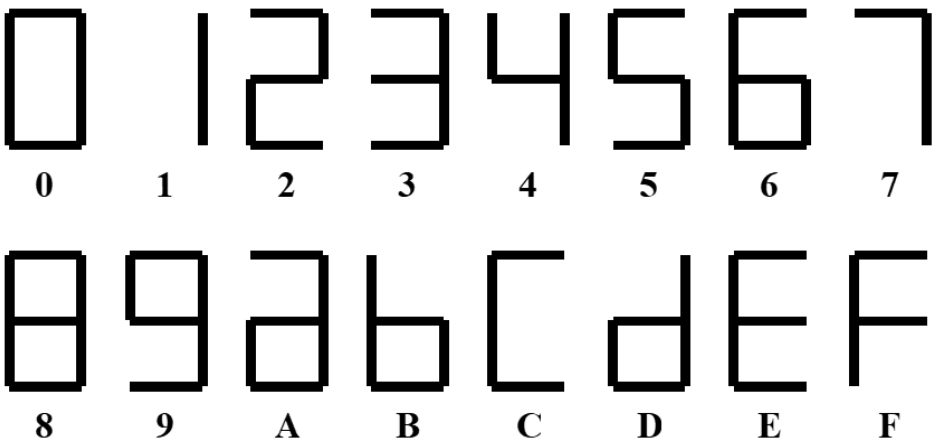


Figure 1.5: The proper arrangement of LEDs to form hexadecimal characters.

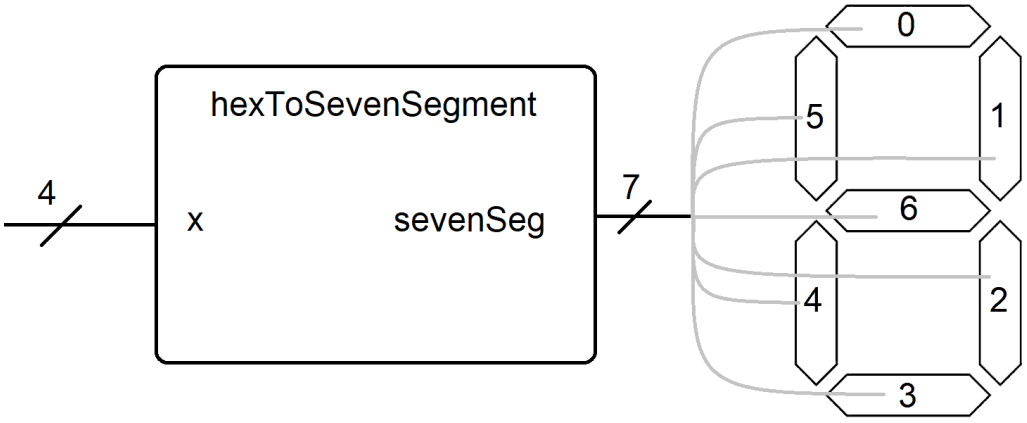


Figure 1.6: The hexToSevenSeg Verilog module driving a 7-segment display.

6. 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.

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 *hexToSevenSeg*.

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

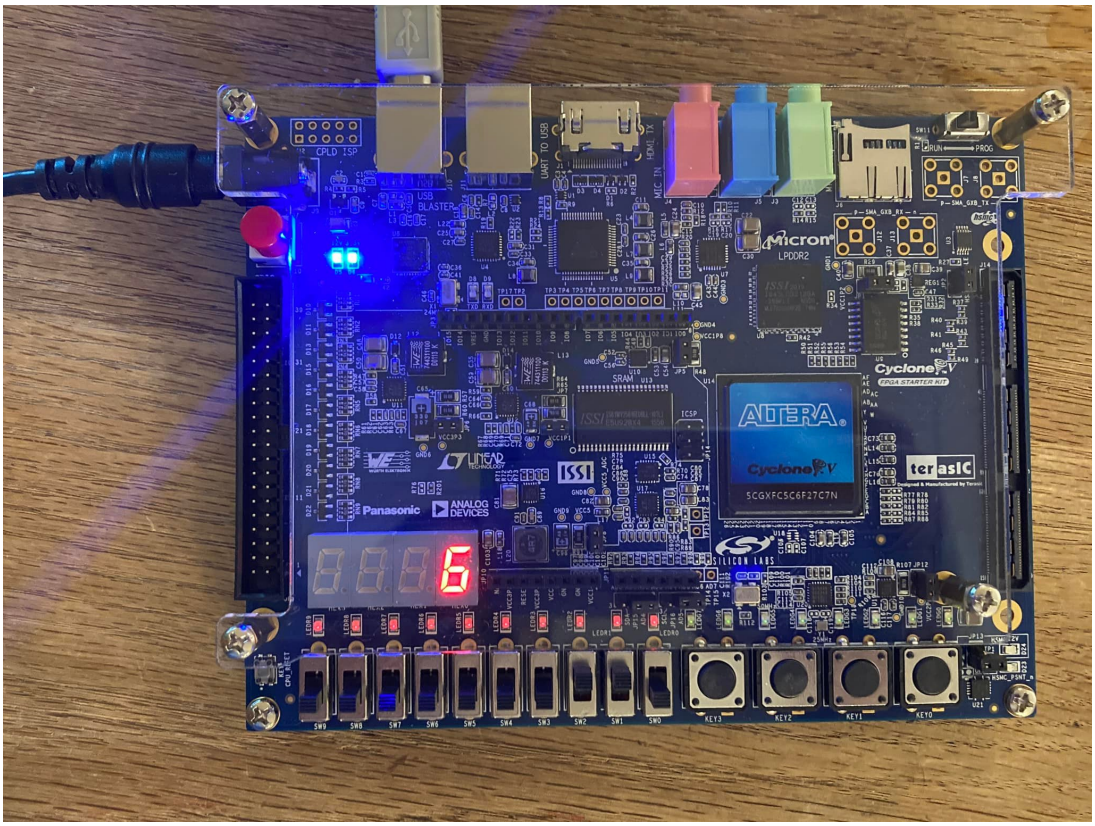


Figure 1.7: The development board properly configured and running the hexToSevenSegment Verilog file.

1.4 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.

Combine lab 1

- Link Truth Table for combinedLab01 function (Table 1.1)
- Link Timing diagram for combinedLab01 function
- Link Pin assignment for combinedLab01 (Table 1.2)

Hexadecimal to 7-segment

- Link Truth Table for hexToSevenSeg function (Table 1.3)
- Link Verilog code for hexToSevenSeg function – just the always/case statement
- Link Timing diagram for hexToSevenSeg function
- Link Pin assignment tables for hexToSevenSeg (Tables 1.4)
- Demonstrate operation in lab