Contents

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

Laboratory 1

Introduction to Verilog

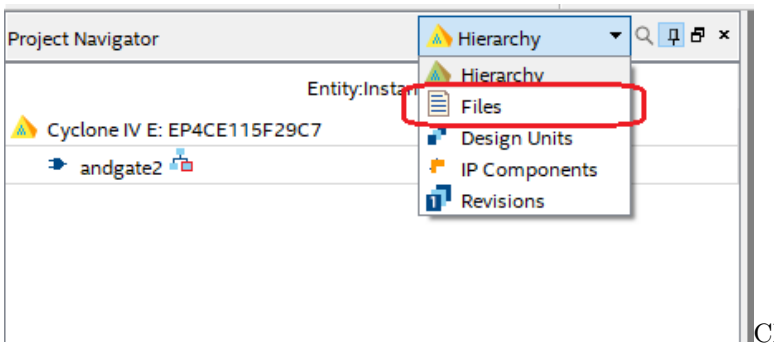
1.1 Objective

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*.

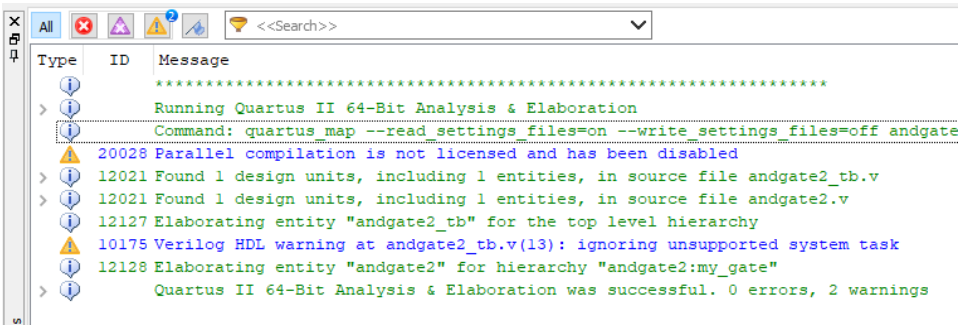
15. If you added the Verilog file in the correct directory and included it in the project, a Verilog file should pop up on the right.

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.

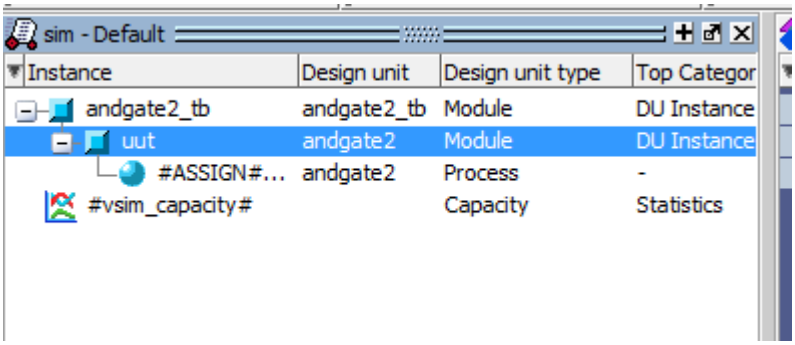


18. In the main Quartus II window, click *Tools -> Run Simulation Tool -> RTL Simulation.* The ModelSim program will launch. This may take a few moments, be patient. If you get a pop-up Nativelink Error window, then go back and check and fix the directory in step 11.

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.*

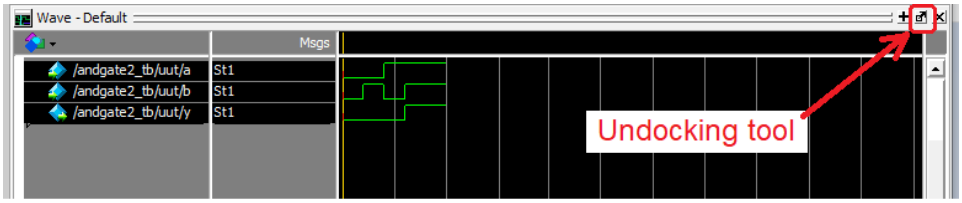
21. In the sim pane, right mouse click on uut and select *Add Wave.*



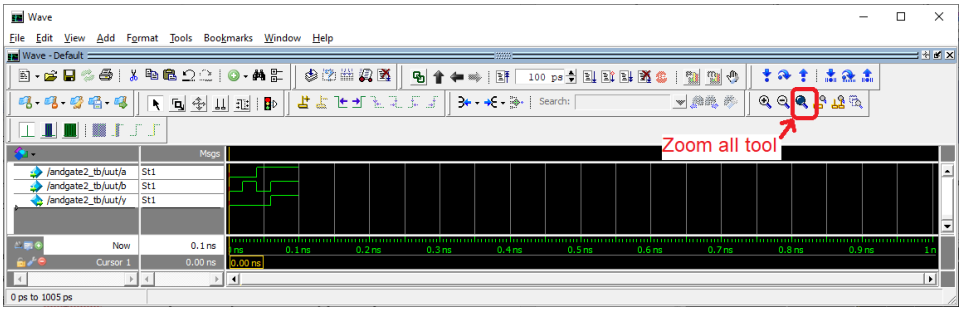
22. Choose *Simulate -> Run -> Run 100*. You should see inputs and output from andgate2. If you see only a small green portion of the waveform on the left margin of the timing diagram, you will need to zoom in on the waveform as follows. First click somewhere in the timing diagram (area under "Undocking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

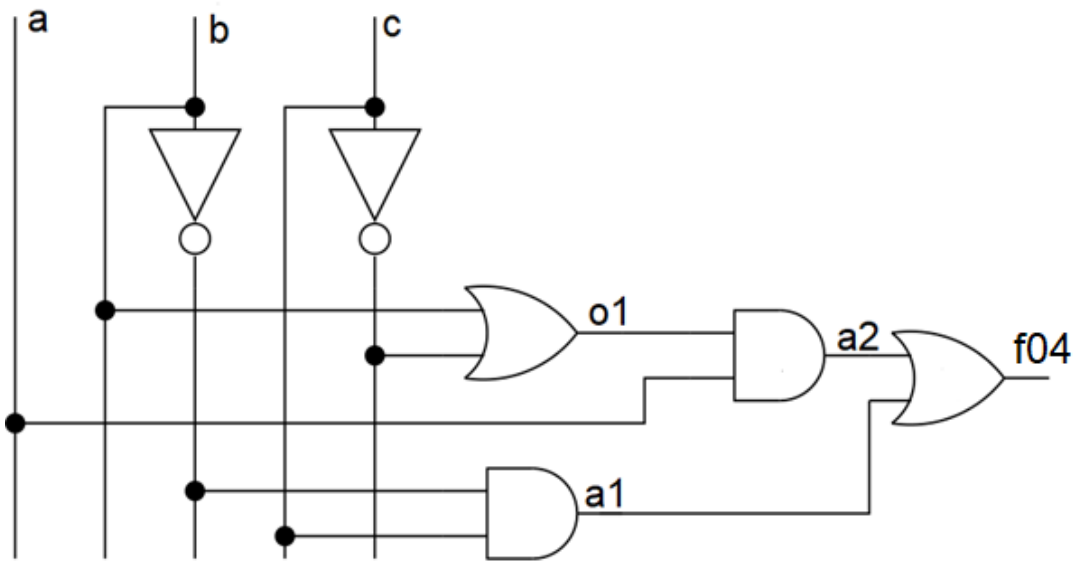
7. From the information in the timing diagram, produce a truth table.

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 2.1.

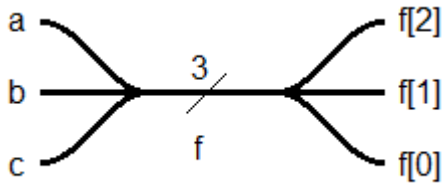


Figure 2.1: A schematic illustration of combining bits into a vector, f, and then accessing the individual bits of f.

Let's explore these ideas with the code snipped show in Listing 2.1. 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 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; 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 2.2 shows an always statement that uses the value of a signal x to compute the value of f.

Listing 2.2: 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 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.

2.2 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 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.

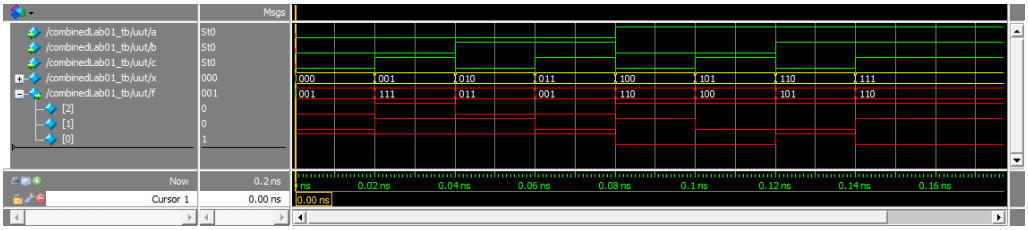
3. Create a project for these two files using the steps from last week's lab.

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 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 2.1.

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 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 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 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 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 2.2. We will use this assignment in the next section.

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

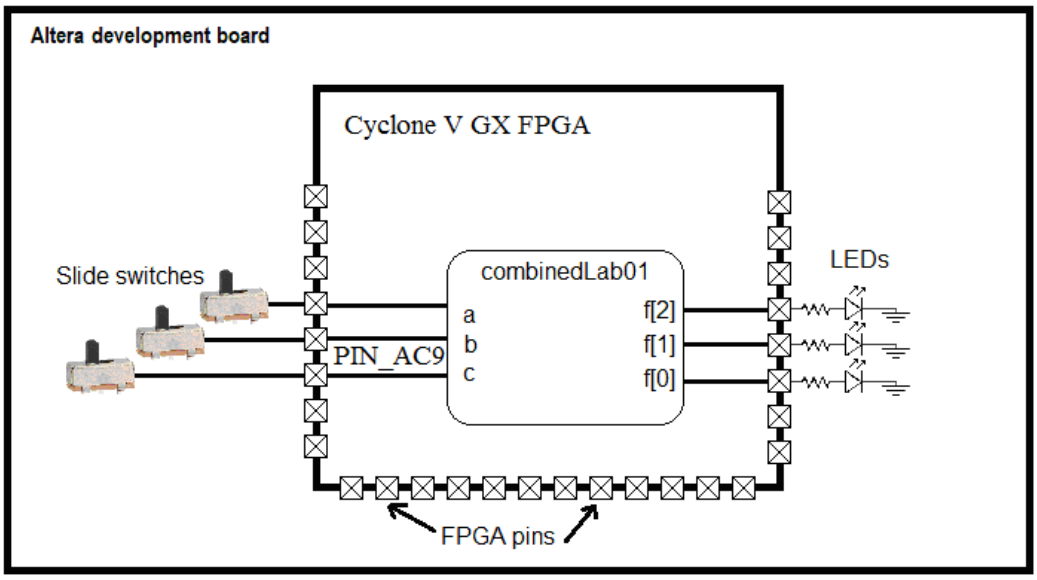


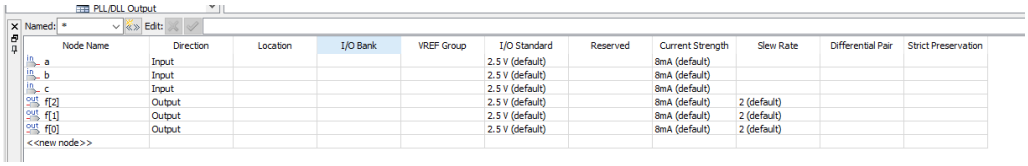
Table 2.2: Pin Assignment Table for combinedLab01.

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			

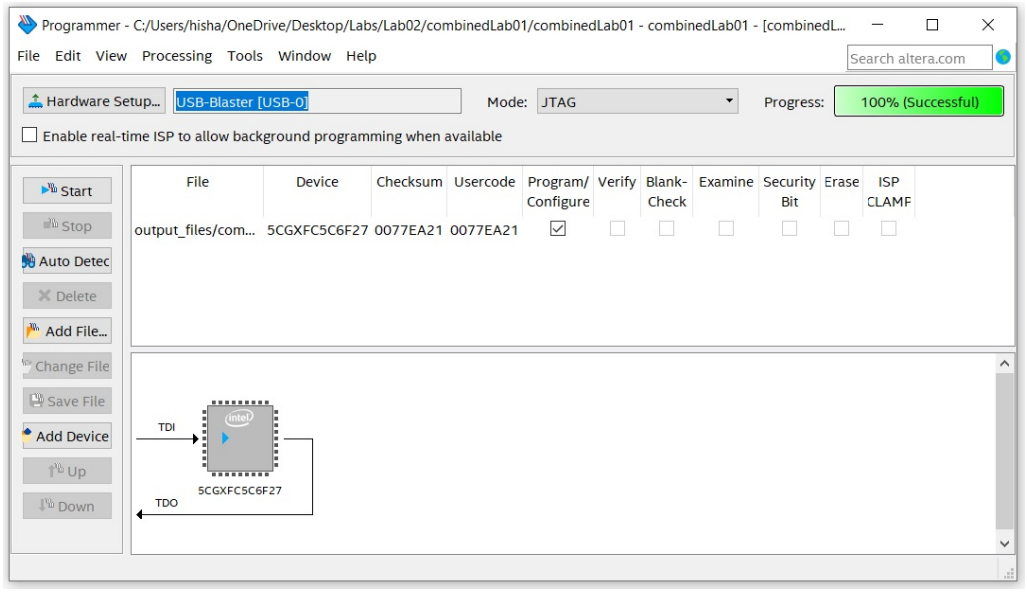
Synthesizing a Verilog Module

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....

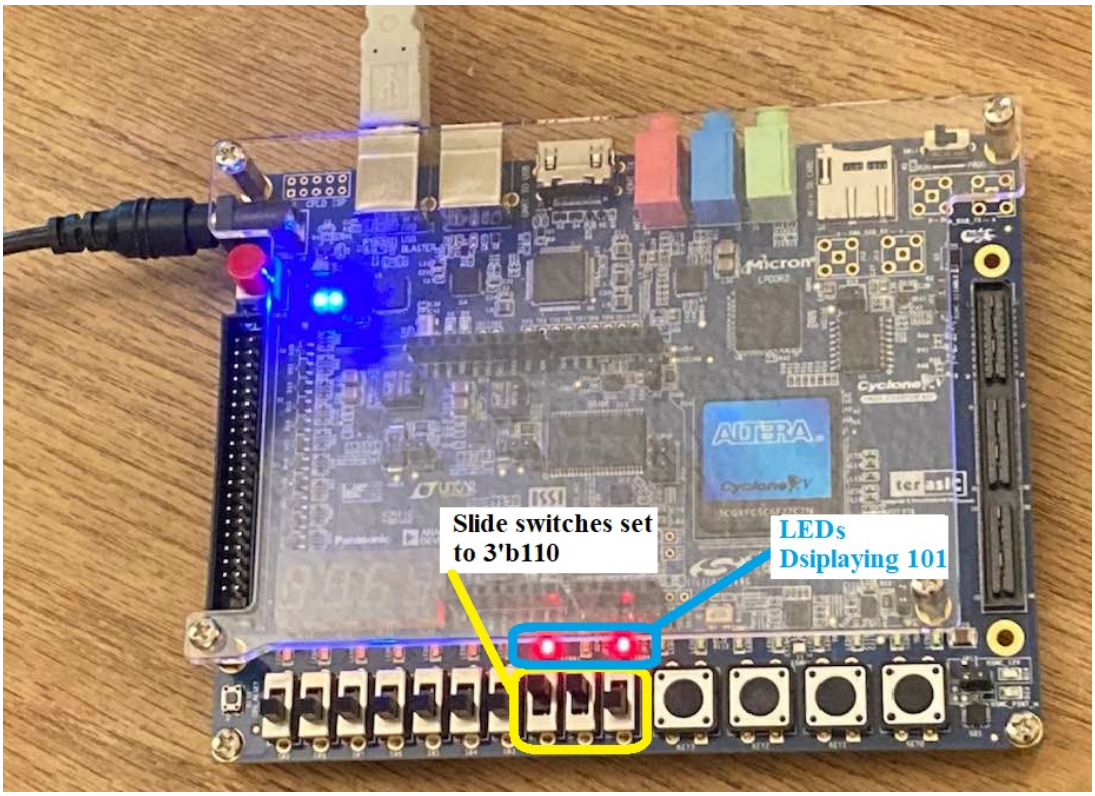
b. In the Hardware Setup select USB-Blaster [USB=0] from the Currently selected hardware pull-down

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



2.3 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.

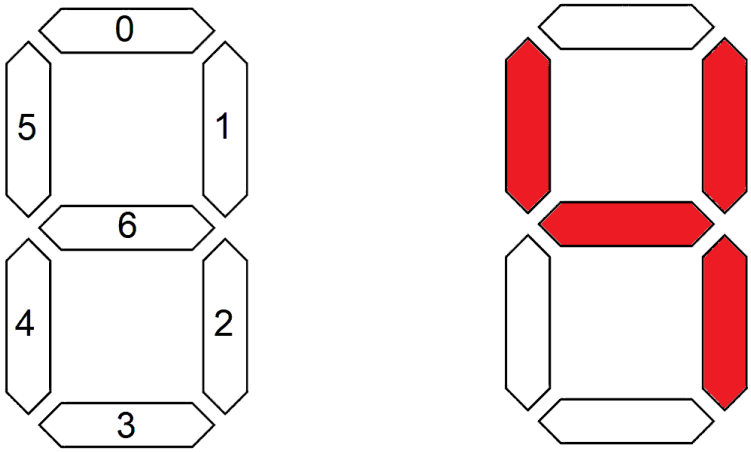


Figure 2.4: Left, the proper numbering of the segments. Right, illuminating segments to form the number 4.

The pattern of segments to be illuminated for each digit is shown in Figure 2.4. 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 2.5 shows the proper formatting for all the values between 0 – f.

Figure 2.6 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 2.3 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 2.3: 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 2.4.

Table 2.4: 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.

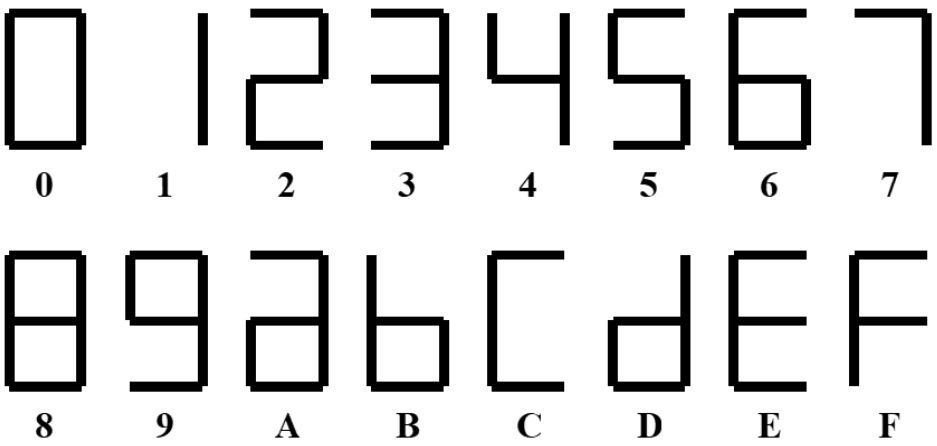


Figure 2.5: The proper arrangement of LEDs to form hexadecimal characters.

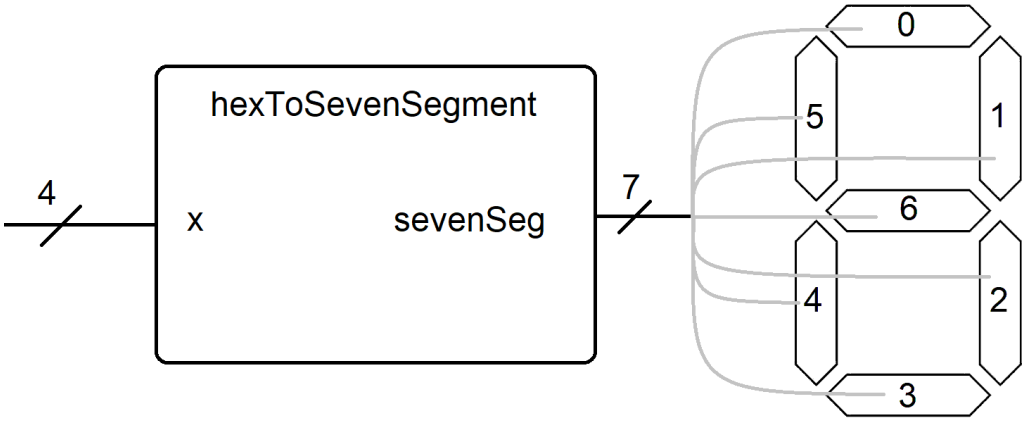


Figure 2.6: The hexToSevenSeg Verilog module driving a 7-segment display.

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.

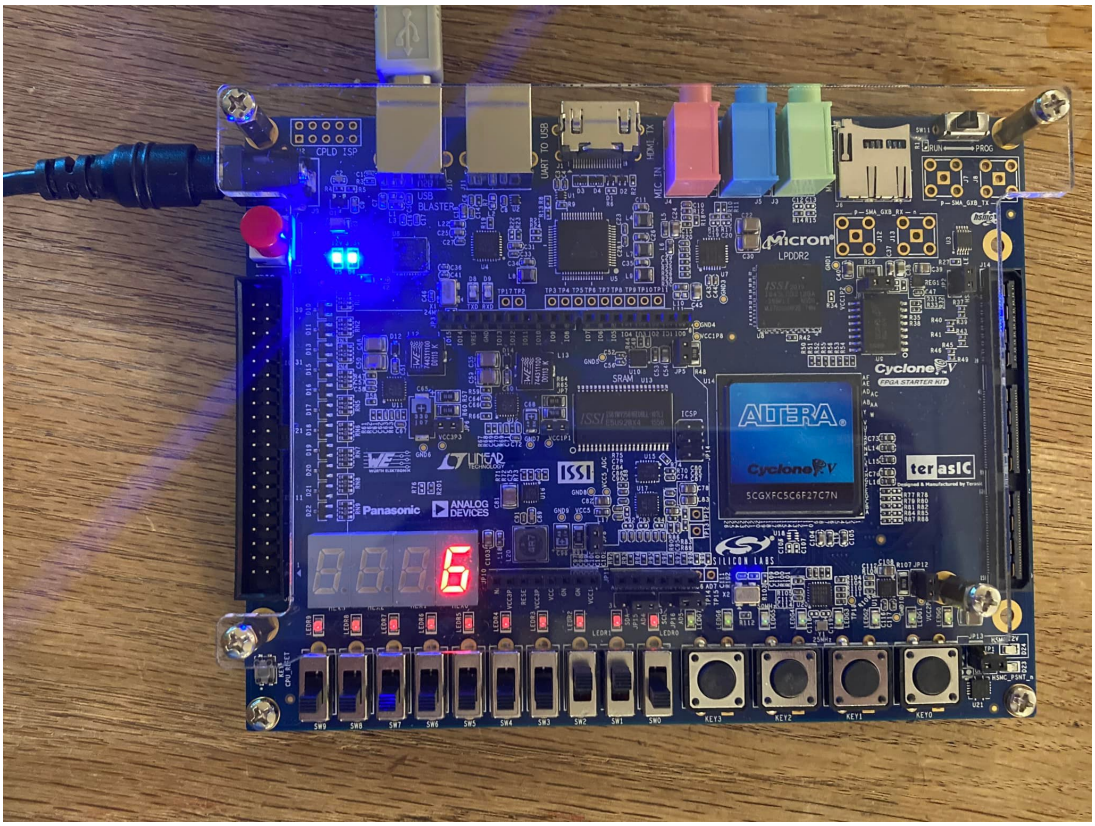


Figure 2.7: The development board properly configured and running the hexToSevenSegment Verilog file.

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.

Combine lab 1

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

Hexadecimal to 7-segment

- Link Truth Table for hexToSevenSeg function (Table 2.3)
- Link Verilog code for hexToSevenSeg function – just the always/case statement
- Link Timing diagram for hexToSevenSeg function
- Link Pin assignment tables for hexToSevenSeg (Tables 2.4)
- Demonstrate working hexadecimal to seven segment module on development board.

Laboratory 3

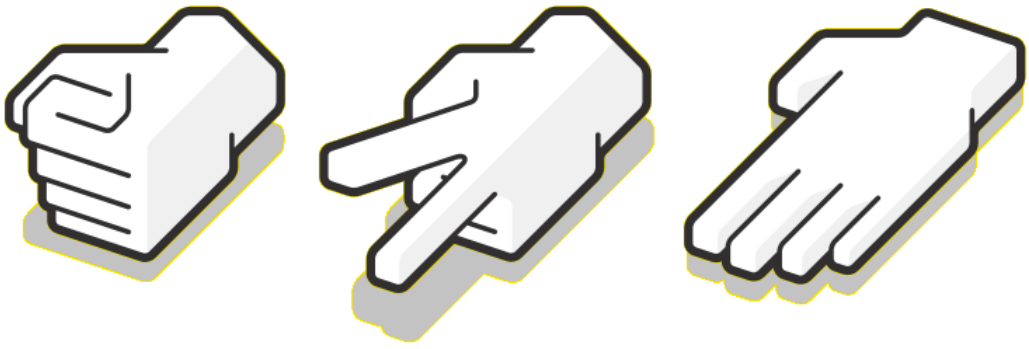
Rock Paper Scissors

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.



ROCK SCISSORS PAPER

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).

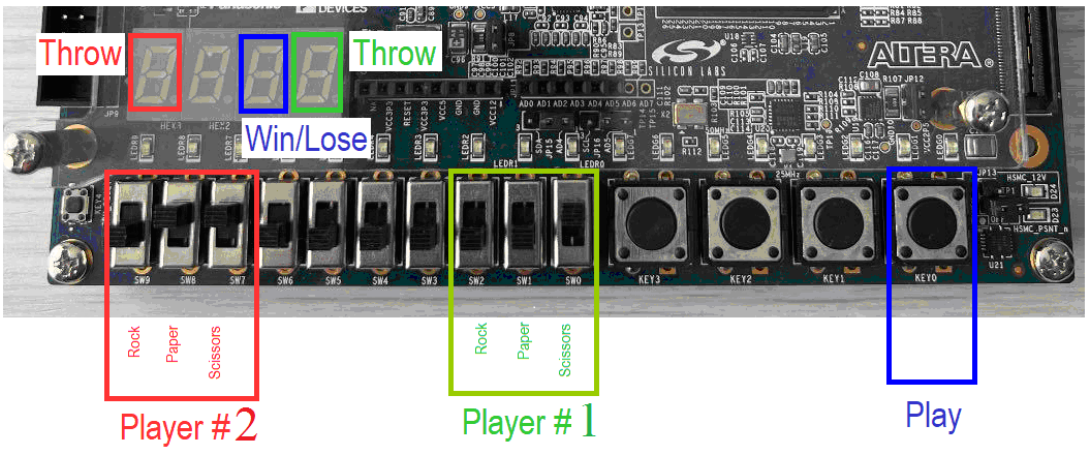


Figure 3.1: The input and output you should use to realize your digital system.

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.

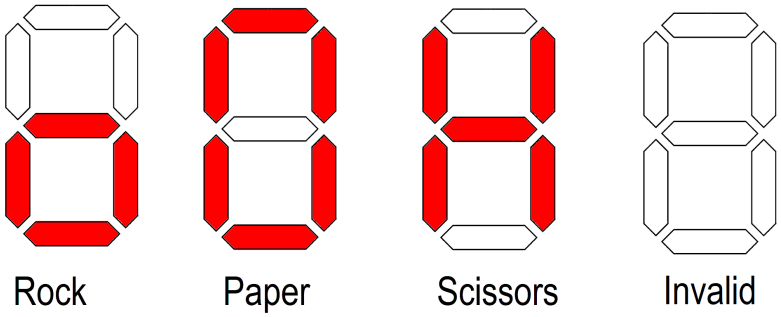


Figure 3.2: Illuminated patters for the different plays.

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.

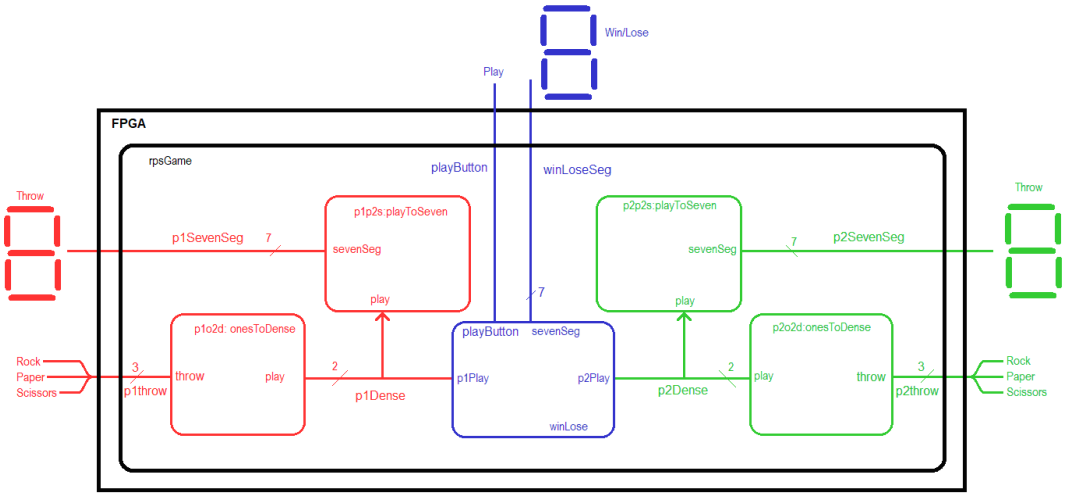


Figure 3.3: System architecture for the rock, paper, scissors system.

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.

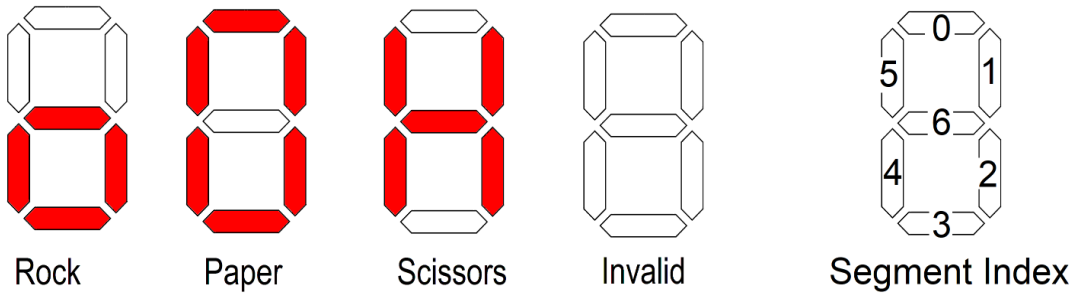


Figure 3.4: 7-segment display patterns for the different throws along with the 7-segment display segment indices.

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.

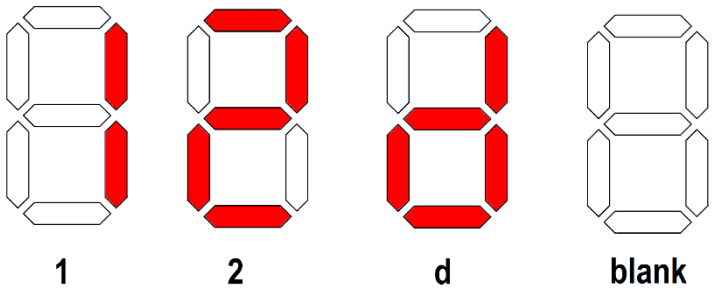


Figure 3.5: The illuminated patterns displayed on the win/lose 7-segment display.

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 3.3.

- 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 (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 3.3.

- 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.

Laboratory 4

High Low Guessing Game

4.1 Objective

The objective of this lab is to use a combination of basic building block and custom combinational logic blocks to realize a complex digital circuit.

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*.

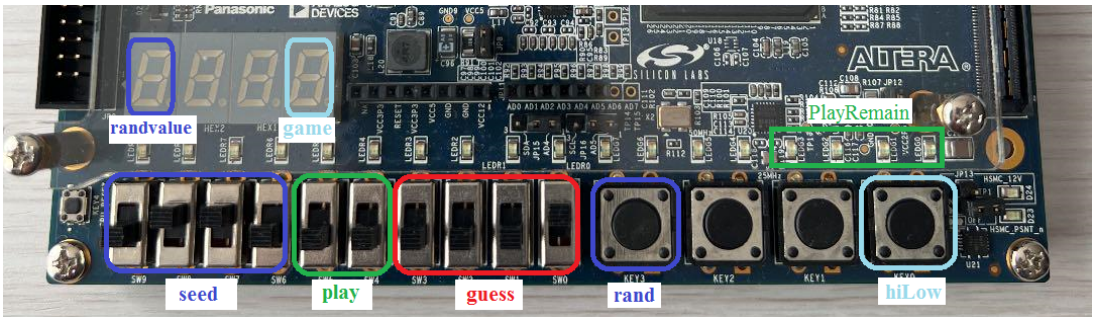


Figure 4.1: The input and output you should use to realize your digital system.

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.

System design

There are an unlimited number of ways that you could implement this digital system. For this lab, I want you to use the system architecture shown in Figure 4.2. A few comments about the visual notation used in this schematic are in order.

1) Lines with the same name are connected together. For example, the rand button input is connected to the 2:1 mux in the upper left corner of the FPGA.

2) When a signal is sent to multiple devices in close proximity, a grey line is used to show that the signal is "underneath a device. For example, the rand signal is sent to the select input of both 2:1 muxes in the upper left corner of the schematic. These unconnected connections are called "air-wires."

3) The input signals are color-coded to correspond to the colors used in Figure 4.1.

4.2 2:1 Mux Module:

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.

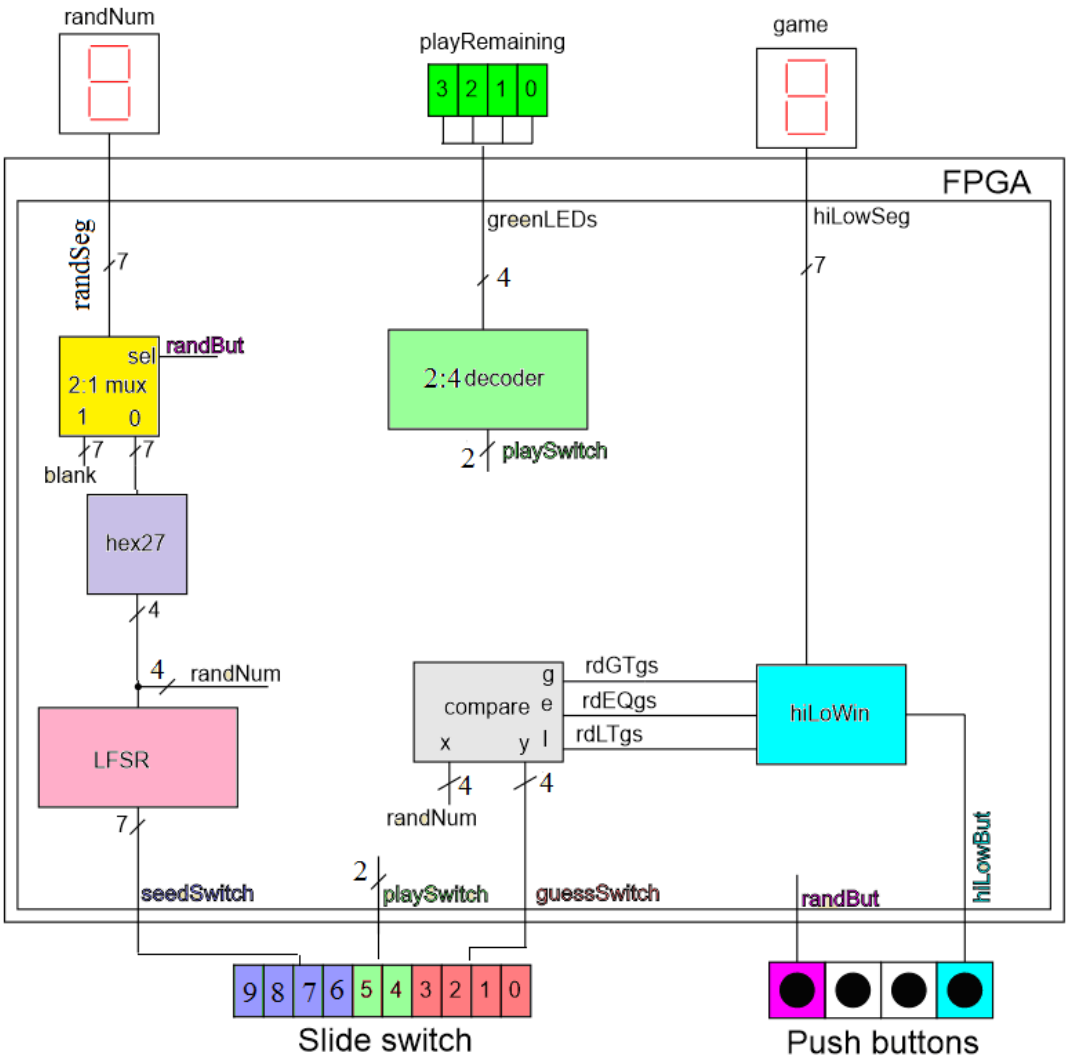


Figure 4.2: System architecture for the guessing game.

I have provided you the Verilog code for a 2:1 mux 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 5.2.

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

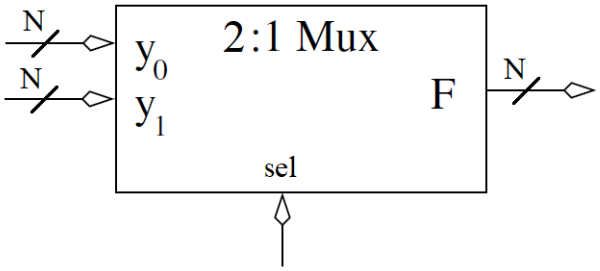


Figure 4.3: The schematic representation of a 2:1 mux.

The signal width, **N**, shown in Figure 4.3 is a placeholder for some integer value in an instantiation. I designed the 2:1 mux Verilog module so that you could specify the value of **N** using the #() specifier shown in the bottom line of code in Listing 1. A component that you can instantiate with different signal widths is called "generic". This modifier is frequently included in the module's name. The default value for the signal width is 8-bit. Be careful, Verilog will allow you to instantiate a genericMux2x1 module without the #() specifier, defaulting to 8-bit signals for **F**, $\mathbf{y_0}$, and $\mathbf{y_0}$ and allowing you to provide signals with different vector sizes. It will do this with a warning that you can find in the Compilation Report tab -> Analysis & Synthesis folder -> Connectivity Checks folder. Click on the offending module and you will see the following error report. Not much, but you have been warned.

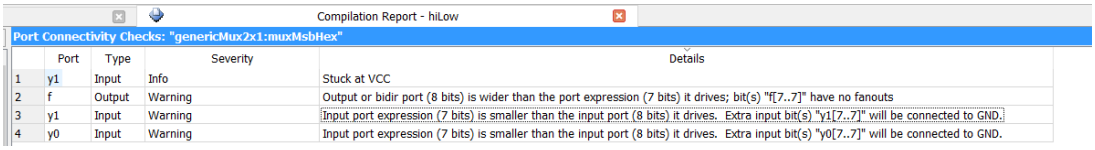


Figure 4.4: Warning that you messed up your vector sizes in a module instantiation.

4.3 Compare Module:

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.

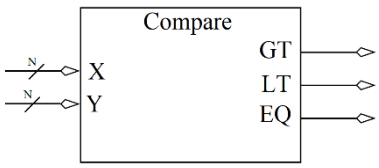


Figure 4.5: A schematic representation of a N-bit comparator.

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

I have provided you the Verilog code for the N-bit comparator 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.

```
// 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 vector width of the **X** and **Y** inputs using the #() specifier. As an example, I've provided an instantiation from my hiLow module in the lower two lines in Listing 2. Like the mux, the default vector width for the inputs is 8-bits. Same warning about vector size mismatch applies to comparators.

4.4 hexToSevenSeg Module:

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.

4.5 2:4 Decoder Module:

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 **4-s** will have an output of 1. All the y outputs whose subscript is greater than **3-s** will have an output of 0.

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

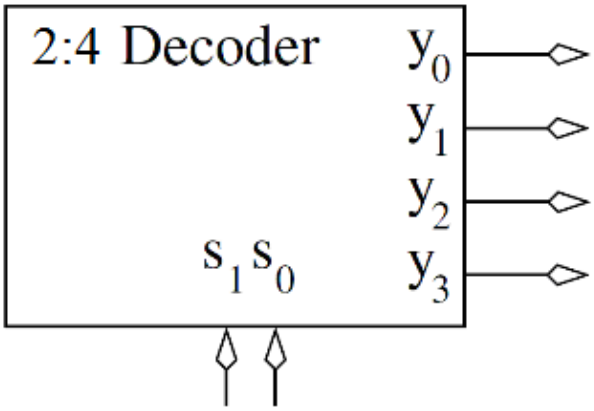


Figure 4.6: A 2:4 decoder.

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".

4.6 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.

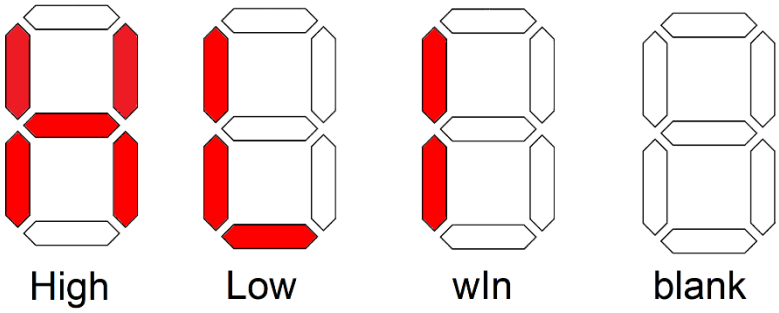


Figure 4.7: The illuminated patterns to inform the guesser about the magnitude of their guess.

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 8.1. Note that the code shown Listing 8.1 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".

4.7 LFSR Module:

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.

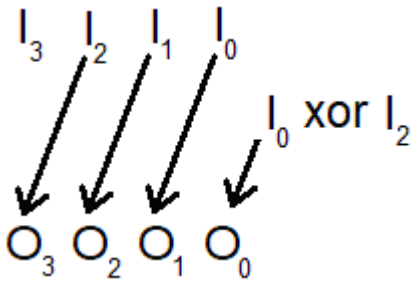


Figure 4.8: A schematic illustration of a 4-bit LFSR operation.

Figure 4.8 shows 3 of the 4 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.

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 7'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, I want you 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).

4.8 hiLow Module:

The hiLow module stiches together all the modules created and provided to you. The hiLow module declaration contains all the signals shown in Figure 4.2. I've provided my module declaration here so that you can more easily relate these to the pin assignments in the pin assignment section.

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

To complete this module, you will need to instantiate the modules in Figure 4.2. To see how to do this, I've grabbed the 2:1 mux out of the system architecture and reproduced it in Figure 4.9. Let's write the Verilog code to instantiate this module in the hiLow module.

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, I named this signal **RandHex**.

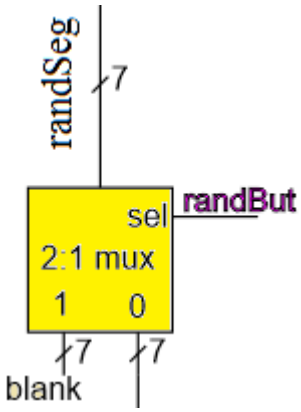


Figure 4.9: A small piece of hardware from Figure 4.2.

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 unique name. I called my instance **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, I want you to:

- 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).
- When you instantiate a module

- 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

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

4.9 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. Note, this is where I had most of my errors.

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]

4.10 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.

LFSR Module:

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

hiLow Module:

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

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

Pin Assignment:

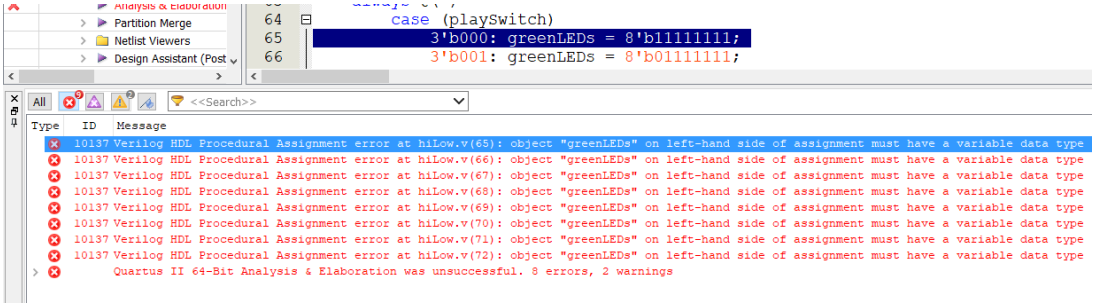
- Link Completed pin assignment table for all the signals in the hiLow module.

4.11 Debugging the Lab:

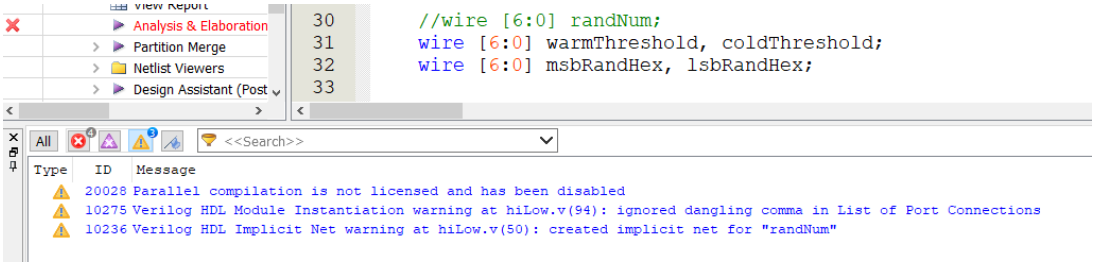
This laboratory typically generates a variety of new errors that you have not seen before. As a result, I have added this section on useful debugging techniques to help you more effectively interpert the compilers output to locate errors in your code. The following is an example story of someone debugging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

After I put together all the components, I ran Start Analysis & Elaboration. It took me a while to find and fix all my errors. I found that by clicking on the Error icon (red x) or Warning icon (yellow triangle) in the console area, I could eliminate a lot of the clutter and focus on the reporting.

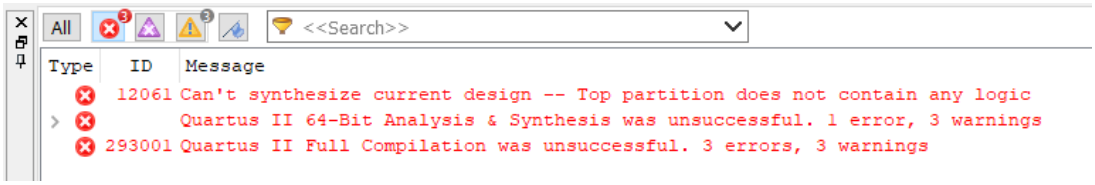
I defined output wire [7:0] greenLEDs; I should have used output reg [7:0] greenLEDs; because greenLEDs is the output of an always/case statement.



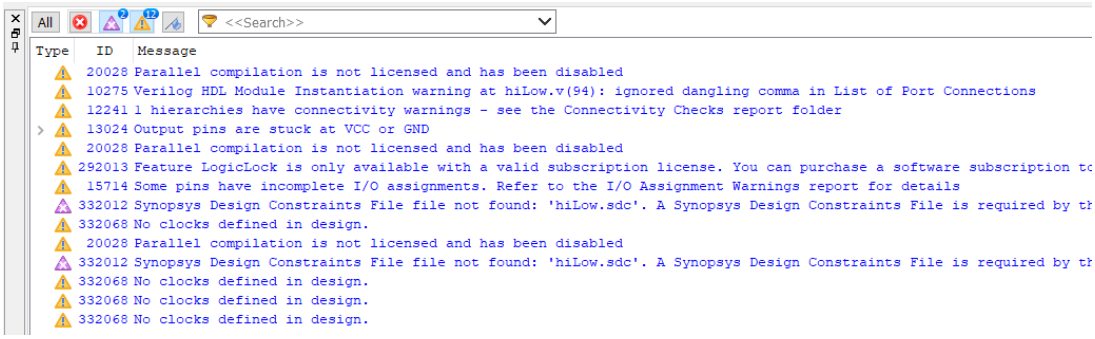
I forgot to include the declaration of randNum – actually I just commented it out to get this error. The top warning always appears and the second is a result of an unused output on my adder (more about this in the next lab).



I accidentally left a testbench as the top-level entity and tried to synthesize.

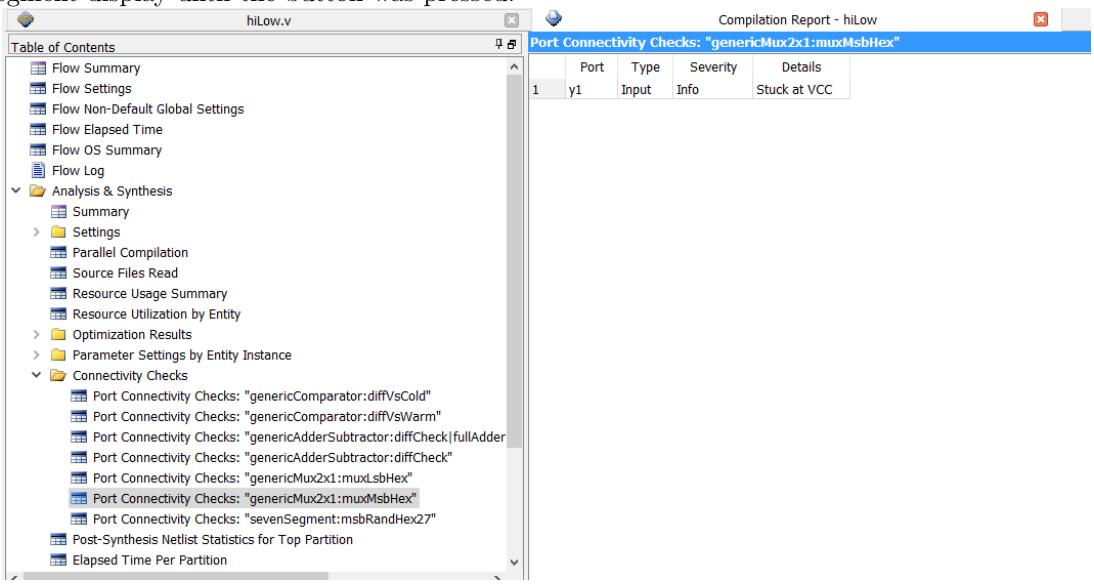


These are all the Critical Warnings and Warnings that I got on my final, working version. You should NOT attempt to fix these “errors”.



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

means that I hardwired one of the inputs to the 2:1 mux to all 1's in order to blank out the 7-segment display until the button was pressed.



Laboratory 5

High Low Guessing Game With Hints

5.1 Objective

The objective of this lab is to modify existing code to add increased functionality. The design requires utilization of basic building block and custom combinational logic blocks to realize a complex digital circuit.

The Guessing Game with Hints

This week's assignment asks you to add some enhanced functionality to the guessing game. Since we are adding functionality to the guessing game, it's worth reviewing the guessing game because we will use some of the terms in the description of our enhanced functionality. 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 exchange until the guesser correctly identifies the *secret number*.

In this week's assignment, you will add circuitry to provide an indication of how far the user's guess is from the secret number by telling them if their *guess* is hot (close to the *secret number*), warm (kind-of close to the *secret number*), or cold (far away from the *secret number*).

The user input and output, shown in Figure 5.1 are the same as last week's assignment with the exception of the **hotCold** button and **clue** 7-segment display.

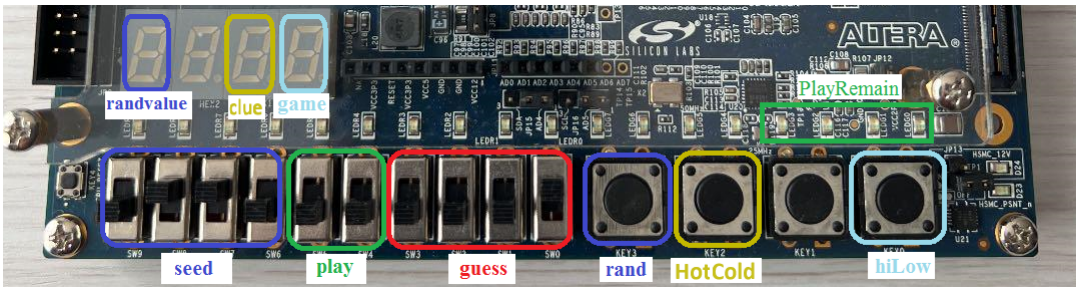


Figure 5.1: The input and output you should use to realize your digital system.

The functionality of the inputs and outputs from last week's assignment are unchanged; the **hotCold** button and **clue** 7-segment display operate as follows.

The player can request a more refined evaluation of their guess by pressing the **hotCold** button. To make this evaluation, the absolute value of the difference between the *guess* and *secret number* is computed and then comapred against warmThreshold and coldThreshold as shown in Figure 5.2.

- If difference < warmThreshold the guess is Hot
- If (difference >= warmThreshold) and (difference < coldThreshold) the guess is Warm
- If difference >= coldThreshold the guess is Cold

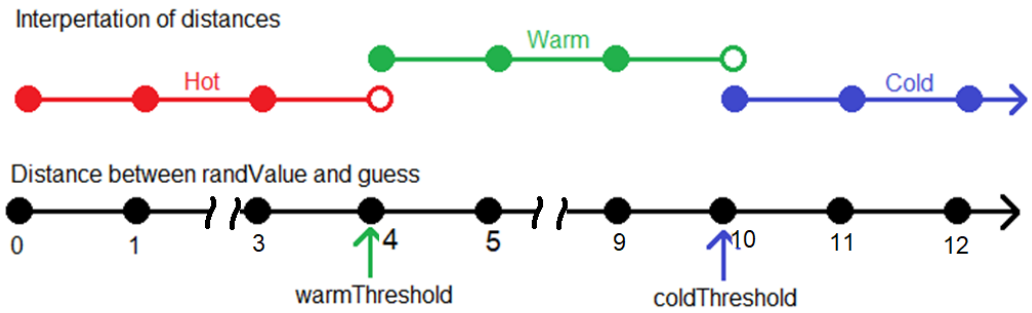


Figure 5.2: The interpretation of the quality of a guess in terms of thresholds.

The difference is always a positive number. For example, if the user's *guess* was 43 and the secret number was 45, then the difference would be 2 making this guess Hot (according to Figure 5.2). If *guess* was 45 and the *secret number* was 43, the difference would be 2 and this guess would also be Hot. Explore the relationship between guess, secret number and the Quality of the guess by completing Table 5.1.

Table 5.1: Determine the quality of a guess at the secret number. Your answer may be a number, pair of numbers, a range or a pair of ranges. Assume a 4-bit word size for guess and the secret number and warmThreshold = 4 and ColdThreshold=10.

guess	*secret number*	difference	Quality
14	11		
8	12		
4	14		
8		2	Hot
	8	[4 to 9]	Warm
	2	[10 to 15]	Cold

The 7-segment display called clue will communicate the quality of the user's guess to the user. It will do this by displaying 'C' if the guess is **C**old, 'A' if the guess is w**A**rm, 'H' if the guess is **H**ot.

System design

There are an almost unlimited number of ways that you could implement this digital system. For this lab, I want you to use the system architecture shown in Figure 5.3. A few comments about the visual notation used in this schematic are in order.

1) Lines with the same name are connected together.

2) When a signal is sent to multiple devices in close proximity, a grey line is used to show that the signal is "underneath a device.

3) The input signals are color-coded to correspond to the colors used in Figure 5.1.

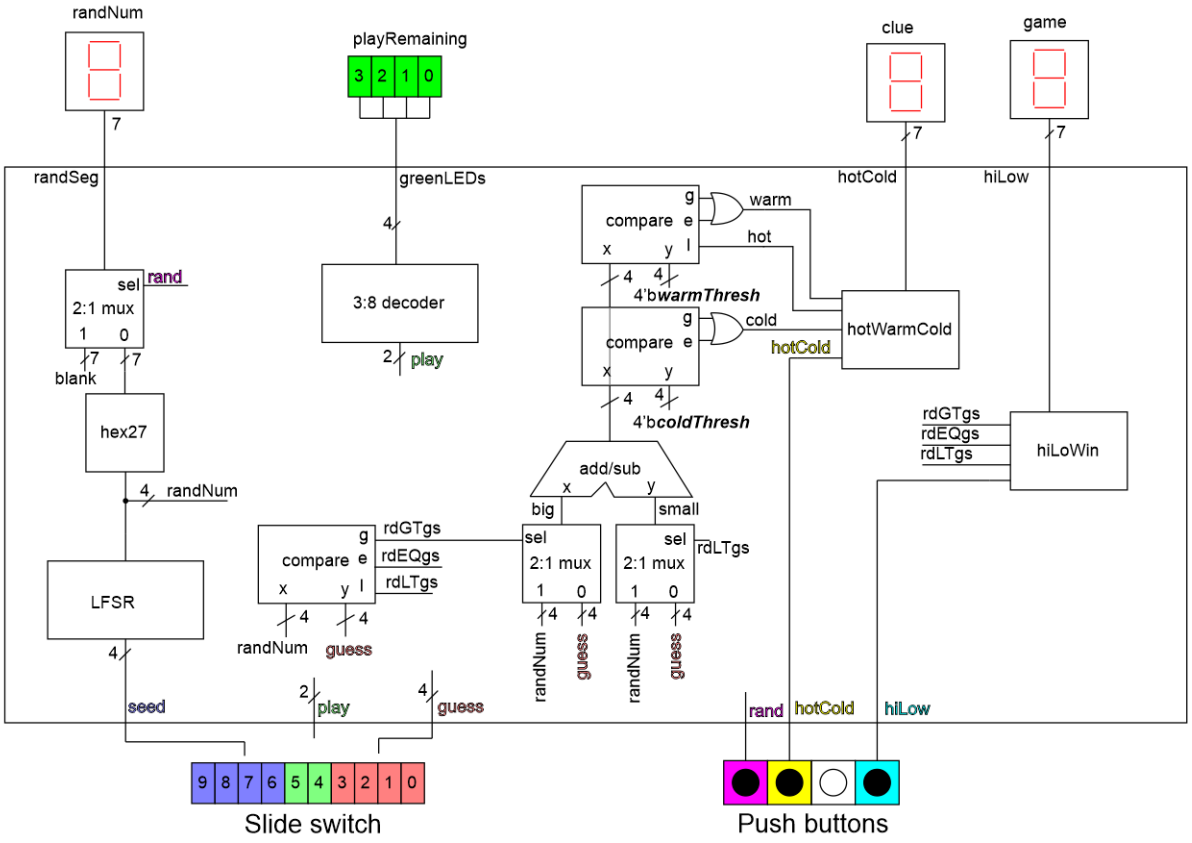


Figure 5.3: System architecture for the guessing game with hint. The select inputs on the big and small 2x1 muxes is intentionally not shown – you will need to figure this out.

The new circuitry is shown in the middle of Figure 5.3 starting with the comparator whose inputs are *randNum* and *guess*. Actually, this comparator should already be in your circuit because it provides its outputs as the input to the hiLoWin module used in the previous lab. This comparator is needed to determine the difference between *randNum* and *guess* as follows.

The difference between the *guess* and *secret number* (called *randNum* in Figure 5.3)that you computed in Table 5.1 is the absolute value of the difference between *guess* and the *secret*

number. We will realize this functionality by placing the larger of *guess* or *randNum* on the x input of the adder subtractor shown in Figure 5.3. The smaller of *guess* or *randNum* is placed on the y input of the adder subtractor. This routing of x and y to the adder subtractor is performed by a pair of 4-bit 2x1 muxes whose select inputs are controlled by one of the comparator's three outputs. The adder subtractor in Figure 5.3 is configured to subtract by hardwiring its fnc input to 1'b1. You will need to determine which single output from the comparator feeds the select input for this pair of muxes.

Let's call the output of the adder subtractor *difference*. This *difference* is compared to the warmThreshold and coldThreshold using a pair of comparators. The values of warmThresh and coldThresh are set in code using the signal declaration and signal assignment statements shown in Listing 5.1.

Listing 5.1: The signal declaration and assignment for guess thresholds.

```
wire [3:0] warmThreshold, coldThreshold;
assign warmThreshold = 4'b0100;
assign coldThreshold = 4'b1010;
```

The output from the warmThreshold and coldThreshold comparators is used as input to the hotWarmCold logic to display an appropriate character on the **hotCold** 7-segment display. You will derive this logic as you work through this lab.

5.2 2:1 Mux Module:

This module was discussed in Lab 4.

5.3 Compare Module:

This module was discussed in Lab 4.

5.4 Add/Sub Module:

A N-bit adder subtractor is a basic building block in many digital systems. The N-bit adder subtractor shown in Figure 5.4 adds its N-bit input x and N-bit input y when fnc = 0 and places the result on the N-bit output subDiff. When fnc = 1 the sumDiff output equals x-y. When the inputs and output are interpreted as a 2's complement values, the sovf output equals 1 when the computation results in an overflow. When the inputs and output are interpreted as binary numbers, the uovf output equals 1 when the computation results in an overflow.

I have provided you the Verilog code for the N-bit adder subtractor on Canvas. This module instantiates N full-adders. Thus, you will need to include the full adder module contained in the file fullAdder.v in your project to instantiate a genericAdderSubtractor instance. Listing 2 shows the module declaration for the genericAdderSubtractor. Note that the output from the module follows the fnc input. The module instantiation shown in Listing 2 corresponds to the system architecture shown in Figure 5.3. Since the inputs to the adder subtractor in the system architecture will not generate overflow, the overflow outputs from this adder subtractor are not needed. When you do not need an output from a module, you can leave its parameter slot unfilled. This explains the pair of empty fields at the end of the module instantiation shown in Listing 2.

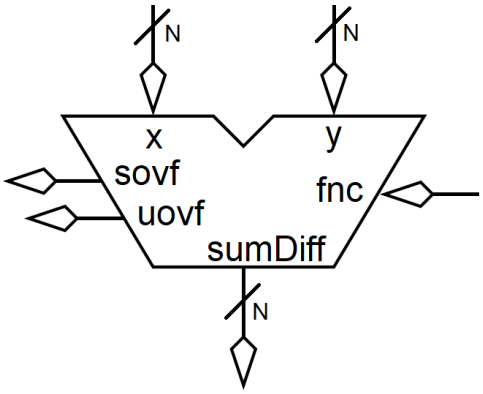


Figure 5.4: A schematic representation of a N-bit adder subtractor.

Listing 5.2: Top, module definition for an adder subtractor. Bottom, module instantation of the adder subtractor in Figure 5.3.

```
// Module definition for the comparator
module genericAdderSubtractor(a, b, fnc, sumDiff, sovf, uovf);

// Module instantiation for an adder subtractor in hiLow digital circuit
genericAdderSubtractor #(4) prox(big, small, 1'b1, difference, , );
```

Like the mux and comparator, the adder subtractor is a generic module. This means that you need to specify the vector width of the **X** and **Y** inputs and **sumDiff** output using the #() specifier. Pay close attention to match the value of this generic and the size of the input and output vectors.

Discrete Logic block:

The input for this logic will come from the output of the two comparators that compare difference and the warm or cold threshold, see Figure 5.3. It would be helpful to take a moment to copy down the logic between the adder subtractor and the hotWarmCold module output in your notes. We will call the comparator that compares *difference* (the output from the adder subtractor) and *warmThresh*, the warm comparator. The other comparator will be called the cold comparator for obvious reasons.

The warm and cold comparators generate a total of 6 signals sent to the "discrete logic" box in Figure 5.3, but the logic in "discrete logic" does not need all of them. To get a better handle on this, let's look at some examples to help uncover the relationship between the magnitude of the difference between the *guess* and *randNum* and the output of the warm and cold comparators. Note, that in the system architecture shown in Figure 5.3, *difference* is applied to the x input of the comparators and the *warmThresh* and *coldThresh* are applied to the y inputs.

Let's assume *coldThresh* = 10 and *warmThresh* = 4. Complete Table 5.2 by comparing the value in the column labeled "*difference*" to *warmThresh* and *coldThresh* and asserting the appropriate comparators outputs. Note, the comparator outputs are prefixed by "w" for warm comparator output and "c" for cold comparator output.

Let's do one row together, *difference* = 9. If we consider the warm comparator, the x input equals 9 (the value of *difference*) and the y input equals 4 (the value of *warmThresh*). Since 9 is greater than 4, the output wGT will equal 1 and wEQ and wLT will both equal 0. If we consider the cold comparator, the x input equals 9 (the value of *difference*) and the y input equals 10 (the value of *coldThresh*). Since 9 is less than 10, the output cLT will equal 1 and cEQ and cGT will both equal 0. Finally, since *difference* equals 9 and this is between the warm and cold thresholds, the quality of the guess should set warm = 1 and hot and cold to 0.

Table 5.2: Complete the following table to determine which comparator outputs are needed to determine the quality of a guess. Let warmThresh = 4 and coldThresh = 10.

difference	warmThresh comparator			coldThresh comparator			Hot	Warm	Cold
	wGT	wEQ	wLT	cGT	cEQ	cLT			
3									
4									
5									
9	1					1		1	
10									
11									

Now, you need to use the values in Table 5.2 to determine the logic for each of the three outputs from the "discrete logic" block shown in Figure 5.3. To do this, look at the conditions that cause each of the outputs and write an expression using AND and OR to describe when that output equals 1.

```
Cold =        // write logic description
Warm =        // write logic description
Hot =        // write logic description
```

For this block of code:

- Make three assign statements, one for hot, warm and cold
- Use only & and | operations.
- Use parenthesis to ensure proper order of operation.
- The *hot*, *warm* and *cold* signals should be "wire" type.

hotWarmCold block:

You will implement the hotWarmCold logic using an always/case statement that takes in as input the 3 outputs from the "discrete Logic" block, *hot*, *warm* and *cold*. These three outputs form a 1's hot code because a guess can only be one of these three conditions at a time. When the **hotCold** button is pressed, the output of the hotWarmCold block forms an illuminate 7-segment representation of the quality of the guess shown in Figure 5.5. The 7-segment output from the hotWarmCold block should be blank when the **hotCold** button is unpressed.

For this block of code:

- Use an always/case statement.

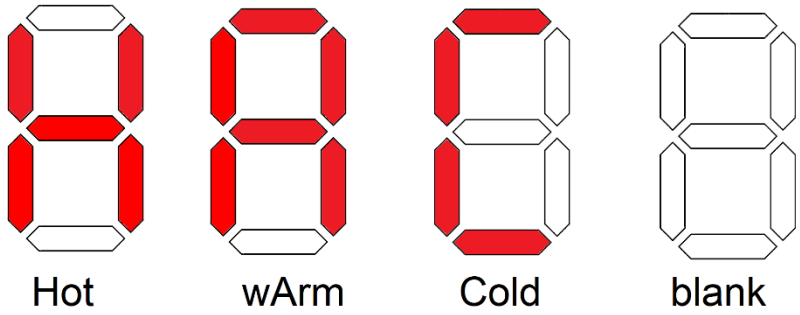


Figure 5.5: The illuminated patterns to inform the guesser about the magnitude of their guess.

- Form a 4-bit vector from the *hot*, *warm*, *cold* signals and the **hotCold** button.
- Use this 4-bit vector as the input to the always/case statement.
- Make sure that the output signal has "reg" type.
- Include inline comments prior to the always/case statement describing the pattern that is displayed on the 7-segment display for each possible output. An example is given in Listing 3.

Listing 5.3: A comment block describing the pattern of illuminated segment for each guess hint..

```
//*****************************************
// Logic to display the quality of the guess
//       Hot  = 'H' = <show binary code>
//       wArm = 'A' = <show binary code>
//       Cold = 'C' = <show binary code>
//
//                  hex[0]
//                  _____
//       hex[5]    |     |        hex[1]
//                 |     |
//                  _____  hex[6]
//       hex[4]    |     |
//                 |     |        hex[2]
//                  _____
//                  hex[3]
//*****************************************
```

5.5 hiLow module:

If you were not able to get the previous lab working, just implement the functionality identified in this assignment and make the *randNum* come directly from the seed switch. In this case,

you should use the top module declaration in Listing 5.4. If you got the previous lab working correctly, then you should use the bottom declaration in Listing 5.4.

Listing 5.4: The module declaration for the enhanced hiLow module if you did or did not get the previous lab working.

```
// Didn't get previous lab working, then use this module declaration
module hiLow(seedSwitch, guessSwitch, hotColdBut, hotColdSeg);

// Did you complete the previous lab successfully? Then use this module de
module hiLow(seedSwitch, playSwitch, guessSwitch, randBut, hotColdBut, hiLov
```

I intentionally left out the connection between the *randNum* and *guess* comparator and the pair of 2:1 muxes that route the larger of *randNum* and *guess* to the x input of the adder subtractor and the smaller of of *randNum* and *guess* to the y input. Note that *randNum* and *guess* are on different inputs to the 2 muxes so that a single output of the comparator will work for both multiplexers. Note, when *randNum* and *guess* are equal, it does not matter which is routed to the x or y input because the output of the adder subtractor will equal 0.

For this block of code:

- Instantiate genericMux2x1 using the module provided in the previous lab.
- Instantiate genericCompare using the module provided in the previous lab.
- Instantiate genericAddSub using the module provided in the Canvas folder for this lab.
- Make sure to include the fullAdder module in your project.
- Use descriptive names for internal signal.
- Use descriptive names for component instance names.

5.6 hiLow_tb module:

The testbench for this module checks hot, warm and cold for guesses that are too high and too low. I carefully selected these values to check the edge cases, meaning on either side of the warm and cold thresholds.

```
warmThresh = 4'b0110 = 4
coldThresh = 4'b0110 = 10
```

Table 5.3 contains the values that you will use to test your circuit. Before using the testbench, you need to understand what your circuit should output. The signal names in the top row of Table 5.3 are borrowed from the system architecture in Figure 5.3. Fill in the missing binary and decimal values for the cells in the guess, big, small and difference columns. In the Comment column, put the quality of the guess as either "Hot", "Warm" or "Cold".

Table 5.3: Table : The values used in the hiLow testbench.

Test	seed	randNum	guess	big	small	Difference	Comment
1			4'b1111				
2			=14				
	4'b1010	4'b0100					

Test	seed	randNum	guess	big	small	Difference	Comment
3			4'b1101 =				
4			4'b1000 =				
5			4'b0111 =				
6	4'b1111	4'b1110	4'b0011				
7			=4				
8			=5				
9			4'b1010 =				
10			4'b1011 =				
11			4'b1110 =				

5.7 Pin Assignment:

Use the image of the Development Board in Figure 5.1 and the information in the User Guide to determine the FPGA pins associated with the input and output devices used by the hiLow module.

Segment	randSeg	hotColdSeg	hiLowSeg
seg[6]	AC22		Y18
seg[5]	AC23		Y19
seg[4]	AC24		Y20
seg[3]	AA22		W18
seg[2]	AA23		V17
seg[1]	Y23		V18
seg[0]	Y24		V19

	seedSwitch	playSwitch	guessSwitch
slide[3]	AE19	N/A	AC8
slide[2]	Y11	N/A	AD13
slide[1]	AC10	AB10	AE10
slide[0]	V10	W11	AC9

randBut	Key[3]	Y16
hotColdBut	Key[2]	
hiLowBut	Key[0]	P11

G[3]	G[2]	G[1]	G[0]
E9	D8	K6	L7

5.8 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.

System Architecture

- Complete Table 5.1.

Discrete Logic block:

- Complete Table 5.2
- Link Logic for hot, warm, and cold signals

hiLow Module:

- Link Verilog code for the body of the hiLow module (courier 8-point font single spaced), leave out header comments.
- Complete Table 5.3.
- Link 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

seedSwitch	radix unsigned	Green trace
randNum	radix unsigned	Lime green trace
GuessSwitch	radix unsigned	Lime green trace
Big radix	unsigned	Cyan trace
Small	radix unsigned	Cyan trace
Difference	radix unsigned	Blue trace
hotWire	default	Orange trace
warmWire	default	Orange trace
coldWire	default	Orange trace
hotColdSeg	hexadecimal	Red trace

  I do not want the signals from the testbench, but rather the signals from inside the hiLow module. You can do this in ModelSim, by expanding the hiLow_tb instance in the left ModelSim pane shown in Figure 5.6 and selecting "uut". Since uut is an instance of the hiLow module, all the signals accessible in the hiLow module are shown in the center Object pane in Figure 5.6. You can add any of these signals by clicking on them and dragging them into the rightmost Wave pane shown in Figure 5.6.

  When compete, your testbench should look like the timing diagram in Figure 5.7.

Demo:

- Demonstrate your completed circuit by the start of next week's lab.

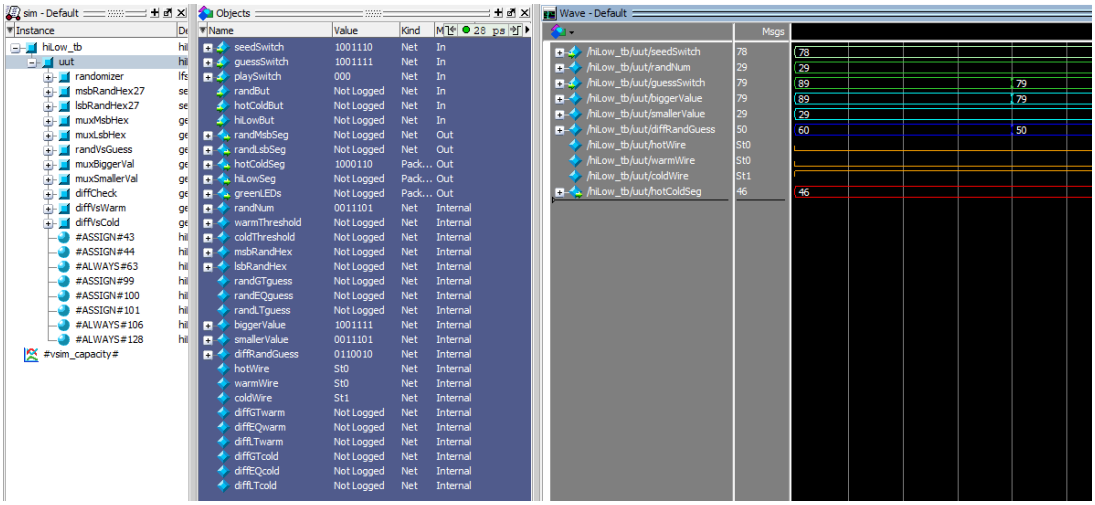


Figure 5.6: Use the design hierarchy to add signals to the testbench.

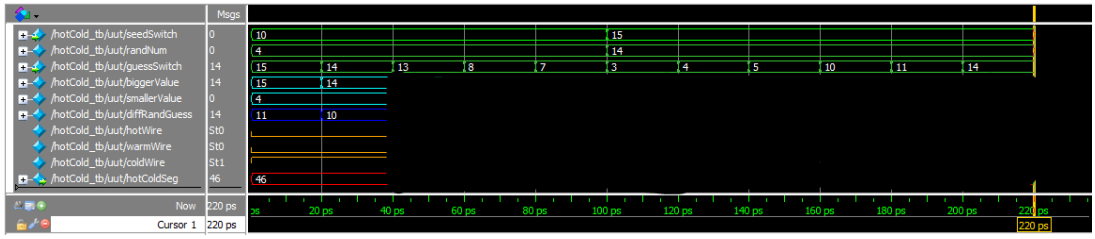


Figure 5.7: A partially obscured timing diagram generated by the testbench.

Clearing errors - Tips from the professor

When my program executed successfully, I got the warnings shown in Figure 5.8. These are mainly the result of the unused overflow outputs from the adder subtractor. You can filter out all the compile messages by clicking on the yellow triangle (with the blue three in this case) on the top line of the console window. Note, if there are several related warnings, they will have one top-level warning with all the instances accessible by clicking the expander arrow (it looks like “>”) to the left of the warning triangle.

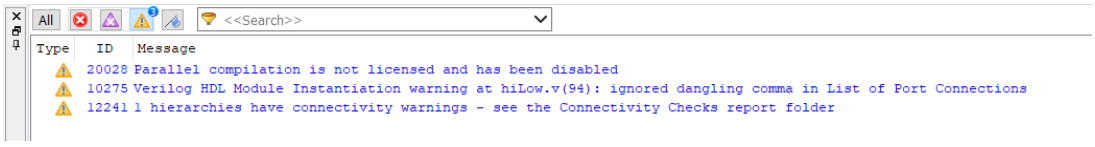


Figure 5.8: The messages console filtered by warnings.

I have found the Connectivity Checks folder in the Compilation Report to help me quickly track down errors. To use it, open the Connectivity Checks folder, click on a Port Connectivity Checks item and read the report in the right pane. In the report shown in Figure 5.9, I selected

the genericAdderSubtractor and note I hardwired fnc to 1 so that it always subtracts. This report also shows that the overflow outputs are unconnected because we left them open using a pair of commas talked about earlier.

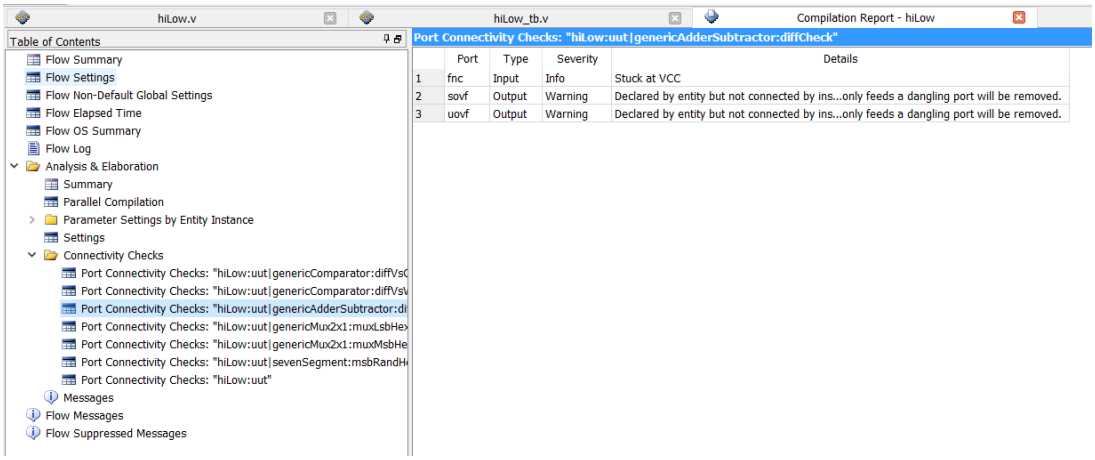


Figure 5.9: Connectivity Checks report for a working hiLow circuit.

Laboratory 6

Calculator With Friendly Output

6.1 Objective

The objective of this lab is to modify existing code to add increased functionality. The design requires utilization of basic building block and custom combinational logic blocks to realize a complex digital circuit.

Basic Calculator

This week you are going to build a very basic calculator that can add or subtract 4-bit values. On the surface, this should require nothing more than connecting some slide switches to the x and y inputs of an adder/subtractor which sends its output to a 7-segment display. And for the most part this is correct. However, instead of displaying the input and output of the adder as hexadecimal values, you will display them as 2-digit decimal values. The user input and output are shown in Figure 6.1. The user enters a pair of 4-bit operands using the left-most slide switches, **xSlide** and **ySlide**. The value entered for **xSlide** is displayed on the two (red) **xDisplay** 7-segment displays. The value entered for **ySlide** is displayed on the two (green) **yDisplay** 7-segment displays. The leftmost the **addSub** buttons specify the operation performed on **xSlide** and **ySlide**. The result is **xSlide** + **ySlide** or **xSlide** - **ySlide**.

The **interp** button determines how the values are displayed on the 7-segment display. When unpressed, the 7-segment displays show the decimal value, when pressed, the 7-segment displays show 2's complement. This will be explained in the next section. As we have only 4 7SDs on board, the same 7SD of operand Y will be used to show the operation result (yellow) when the **yOrResult** button is pressed.

6.2 System Architecture

The system architecture shown in Figure 6.2 shows the adder subtractor processing the xSlide and ySlide inputs. The 4-bit x, y and result values are processed by the sigUnsig box before being displayed on the 7-segment displays. It is now time to turn our attention to this module.

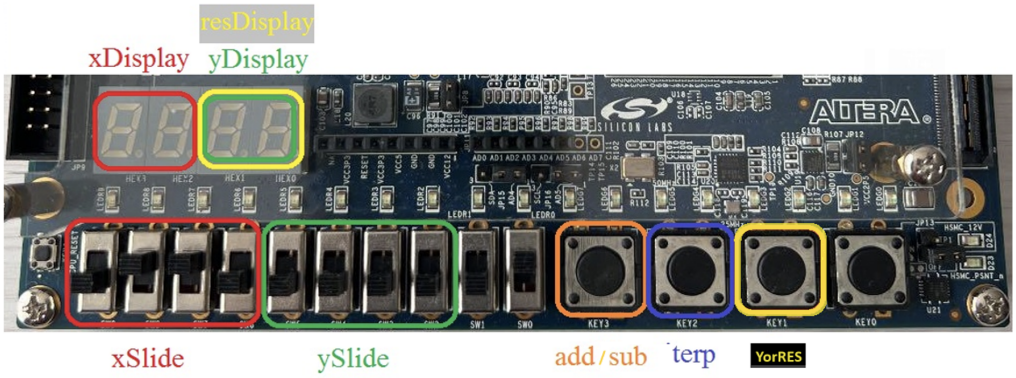


Figure 6.1: The input and output of the calculator digital circuit.

6.3 sigUnsign Module

The significant design problem in today's lab comes in this section, building the sigUnsig module. This module takes in a 4-bit value and displays a 2-digit signed or unsigned representation on a pair of 7-segment displays. Before we go into the internal organization of this module, look at its module declaration in Listing 6.1.

Listing 6.1: Module declaration for the sigUnsig module.

```
module sigUnsig(x, interp, ovf, msDisplay, lsDisplay);
    input  wire [3:0]  x;
    input  wire        interp;
    input  wire        ovf;
    output wire [6:0]  msDisplay, lsDisplay;
```

The 4-bit input x is interpreted as either signed (2's complement value) when *interp = 1* or unsigned (regular binary number) when *interp = 0*. The interpreted value is displayed on the pair of 7-segment display with the tens-digit, blank, or minus sign being displayed by *msDisplay* and the units digit being displayed by *lsDisplay*. If the *ovf* input equals 1, the conversion is overruled and both displays show "X" (which looks a lot like a capital letter "H"). Because we will need it in the next section, Figure 6.3 is the logical arrangements of segments in a 7-segment display. Remember that the segments are active low, meaning a logic 0 illuminates a segment. Thus, the 7-bit code 7'b0100100 illuminates the pattern "2".

Let's start the design of the signUnsig module by looking at the high-level input/output of the module by completing Table 6.1. Do this by filling in the segments of the 7-segment displays that are illuminated for each of the inputs. Then write the binary and hexadecimal value to illuminate those patterns.

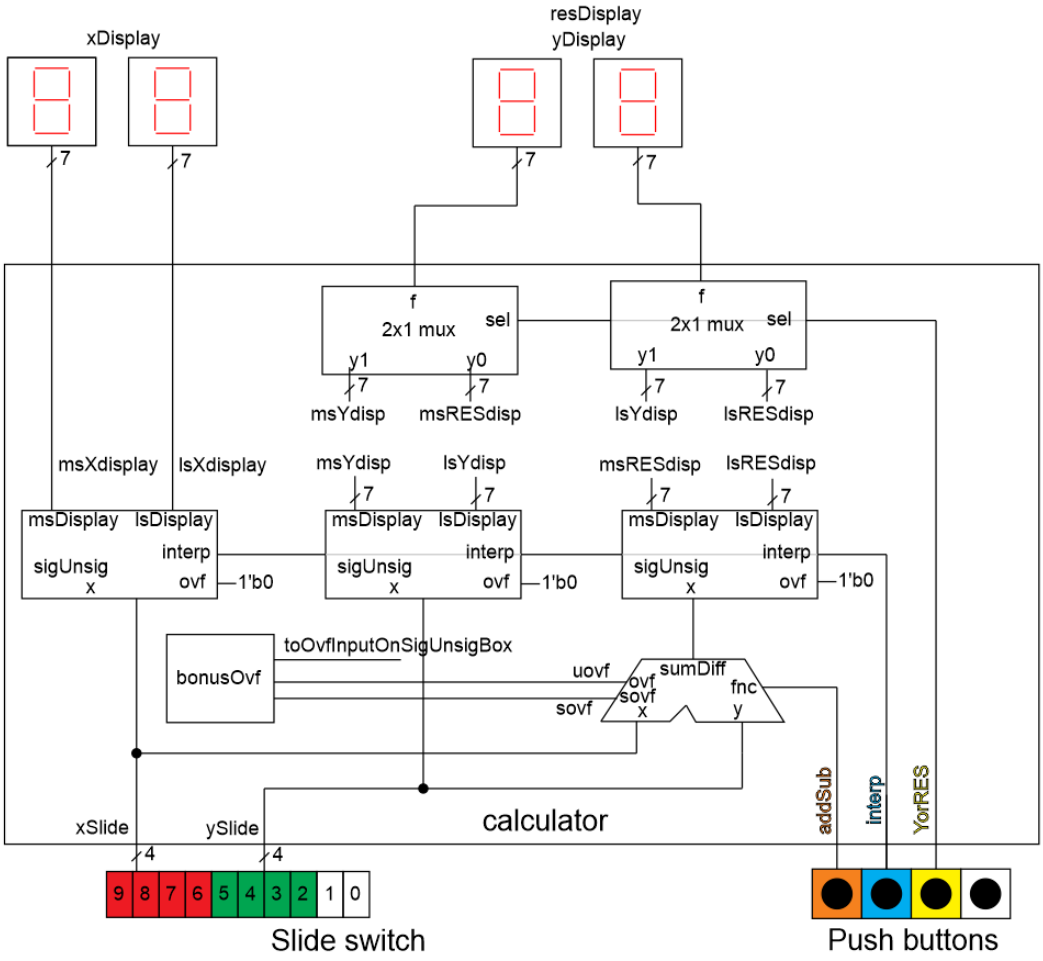
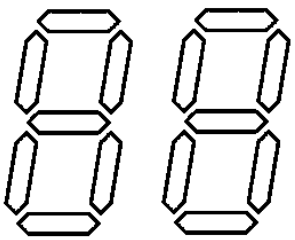
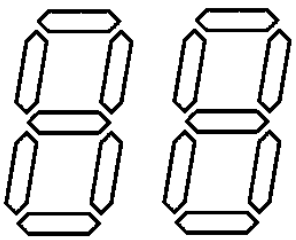
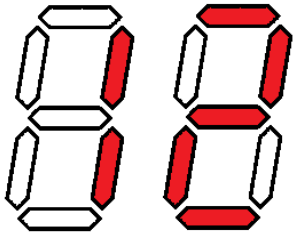
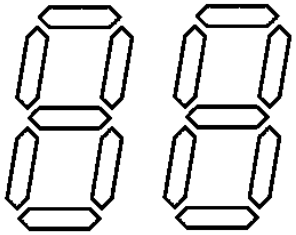
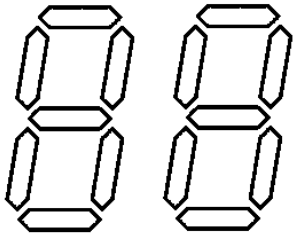
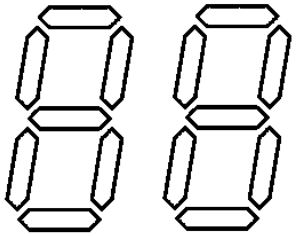


Figure 6.2: The system architecture of the calculator.

Table 6.1: For each set of inputs to the signUnsig module, determine the 7-segment display pattern.

Input	7-segment pattern		Input	7-segment pattern
4'b0010 interp = 1 ovf = 0	 msDisplay = 7'b lsDisplay = 7'b		4'b0111 interp = 0 ovf = 0	 msDisplay = 7'b lsDisplay =7'b

Input	7-segment pattern		Input	7-segment pattern
4'b1100 interp = 0 ovf = 0	 msDisplay = 7'b1111001 = 7'h79 lsDisplay =7'b0100100 = 7'h24		4'b1000 interp = 1 ovf = 0	 msDisplay = 7'b lsDisplay =7'b
4'b1100 interp = 1 ovf = 0	 msDisplay = 7'b lsDisplay =7'b		4'b1010 interp = 1 ovf = 1	 msDisplay = 7'b lsDisplay = 7'b

In order to better understand the output from the signUnsig box, complete Table 6.2 by filling in the values of *msDisplay* and *lsDisplay* for a signed and unsigned interpretation – assume that *ovf*=0 while completing this table. If the interpreted value is positive and a single digit then leave *msDisplay* blank. If the interpreted value is negative then assign *msDisplay* "-". If the interpreted value is greater than 10, assign *msDisplay* "1".

For example, let the 4-bit input x equal to 4'b1100. If x is interpreted as unsigned then its value is 12. In this case your hardware should assign *msDisplay* "1" and *lsDisplay* "2". If x is interpreted as a signed value, the 7-segment displays should show -4, by assigning *msDisplay* "-" and *lsDisplay* "4". Complete the remaining rows of the table.

Table 6.2: The output of the sigUnsig module when ovf=0.

4-bit input x	*interp = 0* Unsigned		*interp = 1* Signed	
	msDisplay	lsDisplay	msDisplay	lsDisplay
4'b0000 4'b0001	blank	0	blank	0
4'b0010				
4'b0011				
4'b0100				
4'b0101				
4'b0110				
4'b0111				
4'b1000				

4-bit input x	*interp = 0* Unsigned		*interp = 1* Signed	
	msDisplay	lsDisplay	msDisplay	lsDisplay
4'b1001				
4'b1010				
4'b1011				
4'b1100	1	2	-	4
4'b1101				
4'b1110				
4'b1111				

Note, when there is overflow, you should assign both the *msDisplay* and *lsDisplay* "X".

You will assign the *msDisplay* output one of four values (three from Table 6.2 and the "X" for overflow) using a 4:1 mux that is provided to you on Canvas. You will arrange the inputs to this mux using the logic that you will complete in Listing 6.2. Note that the four data inputs to this mux (y0, y1, y2, y3) are constants; the inputs to this mux do not depend on x.

You will assign the *lsDisplay* output one of four values (three from Table 6.2 and the "X" for overflow) using a 4:1 mux that is provided to you on Canvas. You will arrange the inputs to this mux using the logic that you will complete in Listing 6.2. Some of the data inputs to the mux depend on x. For example, if interp = 1 (signed) and if x is less than 0, then *lsDisplay* should show the 2's complement of x (and *msDisplay* should display "-"). Instead of flipping the bits and adding 1, you should form the negative of x by subtracting x from 0. On the other hand, if interp = 0 (unsigned) and if x is greater than or equal to 10, then *lsDisplay* should show the units digit of x (and *msDisplay* should display "1"). Form this units digit by subtracting 10 from x.

Complete the code in Listing 6.2. You can assign the value blank, -, constants, x, or a function of x as needed. Note that this code is NOT to be used in your actual code for this lab.

Listing 6.2: Logic that determines the output of the 4:1 muxes in Figure 6.3.

```
if      ( (interp == 0) && (x < 10) ) {                    // y0 input
    msDisplay =                  lsDisplay =

} else if ( (interp == 0) && (x >= 10) ) {                 // y1 input
    msDisplay =          lsDisplay =

} else if ( (interp == 1) && (x >= 0) ) {                  // y0 input
    msDisplay =          lsDisplay =

} else if ( (interp == 1) && (x < 0) ) {                   // y2 input
    msDisplay =                  lsDisplay =

}
```

Now we are ready to put the pieces of the sigUnsig module together. The building blocks in Figure 6.4 are captured in the organization described by Listing 6.2, along with some extra hardware.

You are responsible for connecting the inputs of the 4:1 mux and adder subtractors using the logic described in Listing 6.2. To help you do this, first complete Table 6.3. Take notice of

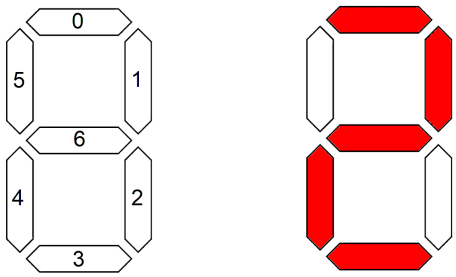


Figure 6.3: The logical arrangements of the segments in a 7-segment display.

the comments in the Listing 6.2 to determine which data inputs to associate with each of the muxes 4 data inputs. You should associate the overflow case with the y3 input.

Table 6.3: The input values to the 4:1 muxes in Figure 6.4.

input	y3	y2	y1	y0
digSel	2'b11			
msDisplay	"X"		1	
lsDisplay				x

In addition, you should add the following to Figure 6.4:

- Wire the inputs of the comparator to determine to generate a signal xGE10 which is logic 1 when x is greater than or equal to 10.
- Wire the inputs of the adder subtractor according to Table 6.3.
- Wire the input of the rightmost hexToSevenSeg .

The last step in building this module is to describe the behavior of the glueLogic box. This function chooses which input of the 4 mux inputs to route to the output. Before you do this, you will need to create a signal *sign* which equals 1 when x represents a negative value (when interpreted as a signed value) and equals to 0 when x represents a positive value (when interpreted as a signed value). Logically speaking, this is a trivial operation – it does not require any logic gates.

Now, we can examine the contents of the glueLogic box. Do this by completing the truth table in Table 6.4.

Table 6.4: Truth table for the glueLogic box.

ovf	interp	sign	xGE10	digSel
1	x	x	x	
0	0	x	0	
0	0	x	1	
0	1	0	x	

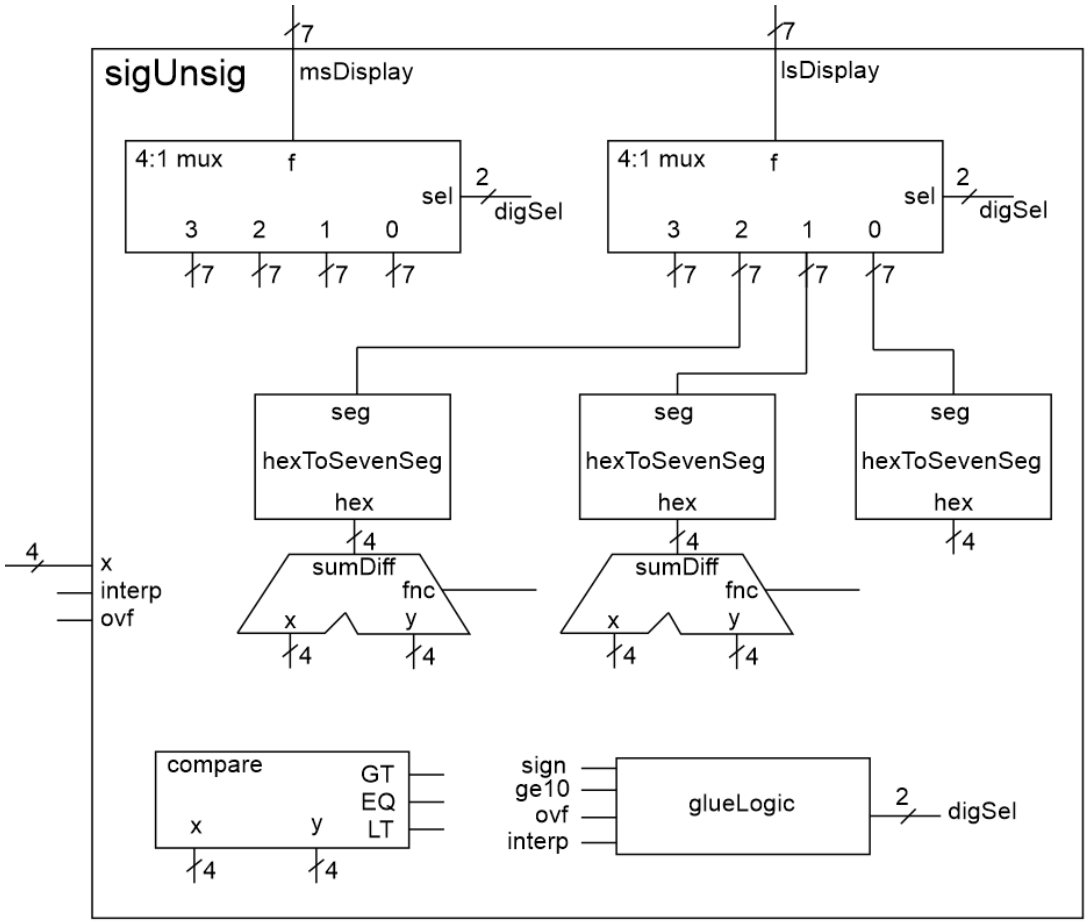


Figure 6.4: The internal architecture of the signUnsig module.

0	1	1	x	

It would make sense to use an always case statement to realize the logic in Listing 6.2. However, an always case statement requires each of the 16 difference cases to be explicitly enumerated. However, the truth table in Listing 6.2 is most efficiently described using don't cares in the input. Fortunately, the always/casez variation (note the "z" at the end of "case") allows don't cares in the input in the form of "?". For example, for the second row in Listing 6.2, the {ovf, interp, sign, xGE10} vector has don't cares for the *sign* value. Therefore, the case for this row is 4'b01?0. It is imperative that you include a "default" case whenever you use a always/case statement. This combination of cases is shown in Listing 6.3.

Listing 6.3: The always/casez statement allows don't cares in the input.

```
always @(*)
   casez ({ovf, interp, xGE10, x[3]})
       4'b01?0: digSel = 2'b00;
```

```
        default: digSel = 2'b11;
    endcase
```

The Verilog code for the signUnsig module consists of 8 instantiation statements and an always/casez statement. For this module, I want you to:

- Use the module declaration given in Listing 6.1.
- Use the module definitions for
  - Generic Mux4x1 posted on this lab's Canvas folder
  - sevenSegment created in lab 02
  - genericAdderSubtractor posted on a previous lab's Canvas folder
  - genericComparator posted on a previous lab's Canvas folder
- Use localparm to give names to the 7-bit constant patterns (fill in the values for x).
  - localparam [6:0] displayBlank = 7'bxxxxxxx;
  - localparam [6:0] displayOne = 7'bxxxxxxx;
  - localparam [6:0] displayMinus = 7'bxxxxxxx;
  - localparam [6:0] displayX = 7'bxxxxxxx;
- Provide meaningful names to the wires in the module.
- Properly tab-indent your code
  - Single level for wire declarations
  - Single level for component instantiations
  - Two levels for casez statement
  - Three levels for casez values

6.4 Bonus Ovf Logic

The default configuration of the system architecture ignores any overflow generated by the adder subtractor. If you choose, you may implement the logic necessary to determine if overflow occurs in the selected interpretation. In order to receive credit, your circuit needs to work under all combination of addSub and interp. Overflow for unsigned subtraction will require some careful analysis.

Your solution should have 2 LEDs, one for signed and one for unsigned. The unsigned overflow LED should illuminate when overflow will occur if the numbers are interpreted as unsigned numbers. The signed overflow LED is on when an overflow will occur if the numbers are interpreted as two's complement numbers.

For example, if the x and y inputs are 1001 and the operation is addition, then both signed and unsigned LEDs will illuminate.

6.5 Pin Assignment

Use the image of the development board in Figure 6.1 in and the information in the Cyclone V GX Kit User Manual (posted on the class web page) to determine the FPGA pins associated with the input and output devices used by the calculator module

Table 6.5: Pin Assignment for the calculator.

Segment	msXdisplay	lsXdisplay	msYorRESdisplay	lsYorRESdisplay
seg[6]				
seg[5]				
seg[4]				
seg[3]				
seg[2]				
seg[1]				
seg[0]				

	x	y
slide[3]		
slide[2]		
slide[1]		
slide[0]		

YorRES	Key[1]	
interp	Key[2]	
addSub	Key[3]	

6.6 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.

signUnsig Module

- Complete Table 6.1.
- Complete Table 6.2.
- Complete the code in Listing 6.2.
- Complete Figure 6.4, including:

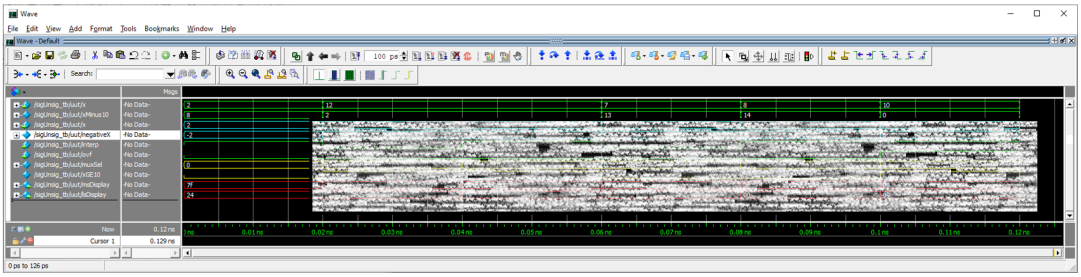
  - Constant values on inputs of 4:1 mux
  - Constant value on the input of the right-most hexToSeventSeg
  - Value on the input of the adder subtractors
  - Values on the input of the comparator

- Complete Table 6.3.
- Complete Table 6.4.
- Verilog code for the body of the sigUnsig module (courier 8-point font single spaced), leave out header comments.
- Run the testbench for the sigUnsig 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

signal	radix	Color for trace
x radix	unsigned	Green
xMinus10	unsigned	Green
x	decimal	Cyan
negativeX	decimal	Cyan
interp	default	Green
ovf	default	Green
digSel	unsigned	Yellow
xGE10	default	Yellow
msDisplay	hex	Red
lsDisplay	hex	Red

I do not want the signals from the testbench, but rather the signals from inside the sigUnsig module. You can do this in sigUnsig, by expanding the sigUnsig_tb instance in the left ModelSim pane and selecting "uut". Since uut is an instance of the sigUnsig module, all the signals accessible in the sigUnsig module are shown in the center Object. You can add duplicates of signals by repeating the drag-and-drop operation.

Your completed timing diagram should look something like the following.



Pin Assignment:

Complete the pin assignment in table 6.5.

Laboratory 7

Cellular Automata

section

Objective

The objective of this lab is to design a digital circuit using a basic memory element.

7.1 Theory - 1-dimensional cellular automata

A common research theme in intelligent systems is emergent complexity. This is the idea that complex behavior can arise from an interconnected arrangement of simple processing elements following simple rules. As a testbench for this idea, Stephen Wolfram[1] explored emergent complexity in a 1-dimensional cellular automata (1-DCA) network. A 1-DCA is an array of processing elements, each of which has one of two states (called alive and dead) and is interconnected to the neighbors immediately to its left and right. The next state of each processing element depends on its current state and the current state of the neighbors to its immediate left and right. Let's explore the evolution of a 1-DCA using the setup shown in Figure 7.1, note that cells which are alive are shaded and cells which are dead, white.

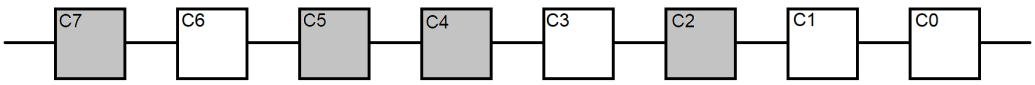


Figure 7.1: A 1-DCA consisting of 8 processing elements. Alive cells are shaded grey, dead white.

Let's examine the next state of the 1-DCA shown in Figure 7.1 using the rule where:

- An alive cell stays alive when exactly 1 of its neighbors is alive, else it dies.
- A dead cell comes alive when exactly 1 of its neighbors is alive, else it stays dead.

[1] Wolfram, S. "Statistical Mechanics of Cellular Automata." Rev. Mod. Phys. 55, 601-644, 1983.

Complete the unfilled entries in Table 7.1 using the initial state shown in Figure 7.1 and the rule above. We will imagine that our processing elements are placed on a ring. This implies that the left neighbor of processing element C7 is C0 and the right neighbor of processing element of C0 is C7.

Table 7.1: The next state of a processing element depends on the current state and the number of alive neighbors.

Cell	Current State	# Alive neighbors	Next State
C0	Dead		
C1	Dead		
C2	Alive	0	
C3	Dead		
C4	Alive		Alive
C5	Alive		
C6	Dead		
C7	Alive		

You are going to build a digital system to calculate the next state of a 17-element array of processing elements. In order to do this, you will need a way to easily specify the rule used to determine the next state of a cell because you are going to be able to change it. Before doing this let's agree to associate the value of 1 for an alive cell and illustrate it as a filled black square and associate the value 0 with a dead cell and will illustrate it as an empty square.

We will formalize the next state rule by enumerating the next state of a cell for every configuration of its current state and its neighbor's state. Since a cell and it's two neighbors can each have two states, there are a total of 8 configurations. As an example, I've graphically illustrated the rule used in our previous example, in Figure 7.2. This arrangement shows the 8 configurations arranged so the binary code of the current states goes from 3'b000 at right to 3'b111 at left. The next state for each configuration is shown below the current state configuration.

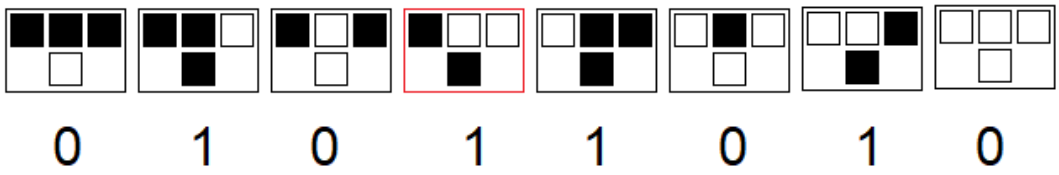


Figure 7.2: The next state of the center cell depends on the current state of a cell and its neighbors.

As an example, let's look at the 4^{th} from left arrangement, outlined in red. In the top row of this arrangement, the center cell is dead and one of its neighbor's is alive. Note that this configuration has binary code 3'b100. Using the rule used to complete Table 7.1, the next state of this cell is alive, shown black. The next state value is placed below each configuration and the collection forms an 8-bit number 8'b01011010, which when interpreted as a decimal value is equal to 90. We will call the rule used to update the cells in Table 1, rule 90.

The fact that we are numbering this rule, implies that there may be others. In fact, there are 256 different rule sets. You can create new rules by substituting new combinations

of alive and dead states for each next states shown in Figure 7.2. Not all rules generate interesting behavior. For example, rule 0 immediately kills any and all alive cells producing a desert. Stephen Wolfram explored every combination of rules and characterized the resulting patterns into one of four classes depending on their sophistication. Wolfram illustrated this sophistication by arranging successive iterations of the 1-DCA as rows. As an example, if you start with a single alive cell and run Rule 90 over many iterations, drawing iteration below its predecessor, you get the image shown in Figure 7.3. Some of you may recognize this figure as the Sierpiński Triangle[2], an elementary fractal.

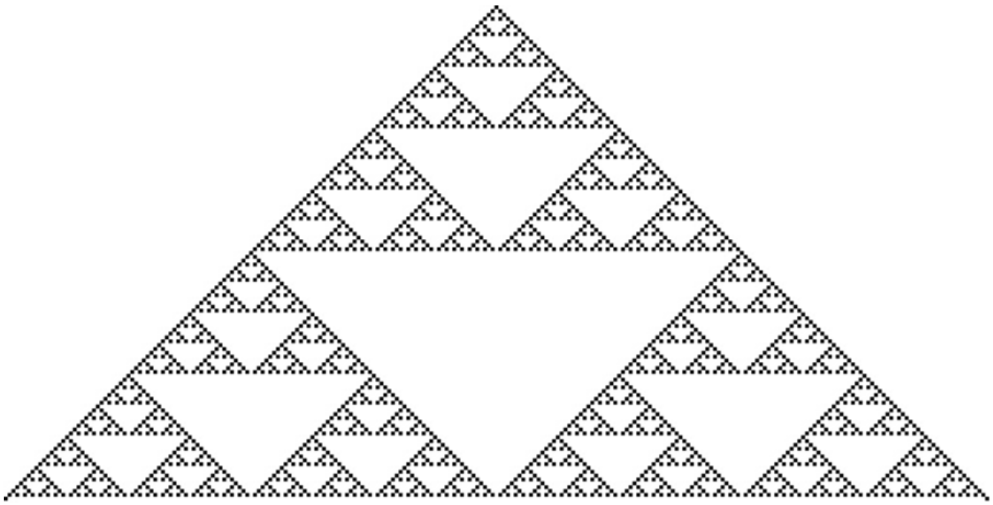


Figure 7.3: The evolution of a single alive cell (shown at top) under Rule 90.

You now have all the information and terminology you need to build a digital system to compute the next state of a 1-DCA, so let's get to it.

7.2 Implementation - 1-dimensional cellular automata

You will use the inputs and outputs shown in Figure 7.4 to realize a 9 cell 1-DCA. I chose 9 cells so that we could have a unique center cell. Each of the 9 **Initial State** slide switches corresponds to one of the 9 cells. In order to set the state of the cells the **loadRun** slide switch must be in the down position. Moving the Initial State up/down will set its corresponding cell to 1/0 respectively when the array is clocked. A cell that is alive will illuminate its associated **CurrentState** LED. After the initial state of the cells is set, you should move the **loadRun** slide switch up and set the 8-bit rule value using the **Rule** slide switches. Every time that the array is clocked, the **CurrentState** LEDs will show the array.

The **reset** button will reset the state of all the cells to 0 – dead. The clock is a bit more complex than you might expect because of switch bouncing. When you press one of the push buttons on the Cyclone V GX board, the signal generated may not transition smoothly from logic 0 to logic 1. Instead, it may do something like that shown in Figure 7.5. In this figure, a single press of the button created four transitions from logic 0 to logic 1. This happens because the metal contact attached to the round plastic button physically bounced off the

[2] https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle

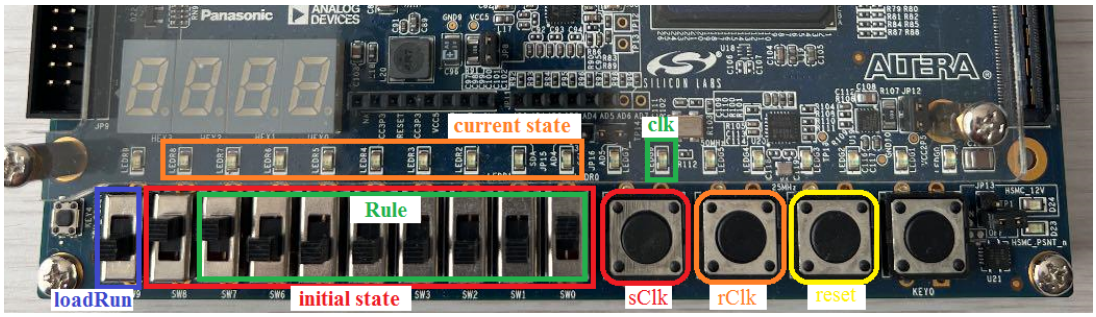


Figure 7.4: The input and output of the 1D cellular automata.

metal contacts attached to the body of the button. This is more prone to happen if you quickly and sharply jab at the button.

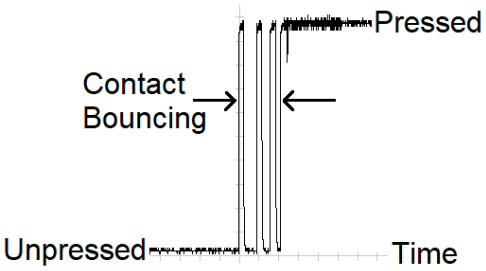


Figure 7.5: Switch bouncing makes generating a single clock edge problematic.

However, even the most casual of presses may generate switch bouncing, so you are going to design a digital circuit (an SR latch) to prevent this from happening. When you press the **sClk** button, the clock signal will be set. When you press the **rClk** button, the clk signal will be reset. If the **sClk** or bounces, the clock line will not bounce because this switch bounce will only cause the clk line to be repeatedly be set when it already set. Likewise, with the **rClk** signal repeatedly resetting the clk signal if **rClk** bounces. Finally, in order for you to know the state of the clk, its logic level is display on the **clk** LED.

Since the buttons driving the rClk and sClk signals are logic 0 when pressed, you will want to invert these two signals before feeding them into the SR latch as shown in Figure 7.6. Thus, to toggle the clock signal you will press/release the sClk button to set clk to 1. Then press/release the rClk button to clear the clk to 0. The back to sClk, etc...

7.3 System Architecture

The system architecture shown in Figure 7.7 shows the overall organization of the cellular automata. Slide switches [0-7] are used as initial state when the loadRun slide switch is set to 0. When the loadRun slide switch is set to 1, slide switches [0-7] are used as the evolution rule. The clock is generated by the SR latch whose inputs are sClk and rClk. The state of the cells are displayed on the 9 red LEDs. The processing elements forming the array are called singleCell and discussed in the next section.

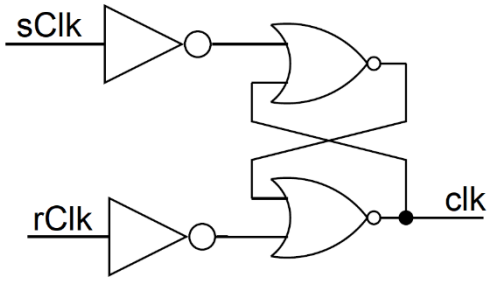


Figure 7.6: The organization of the SR latch to run the clock requires inverters in the sClk and rClk inputs and an SR latch to help remove signal bouncing.

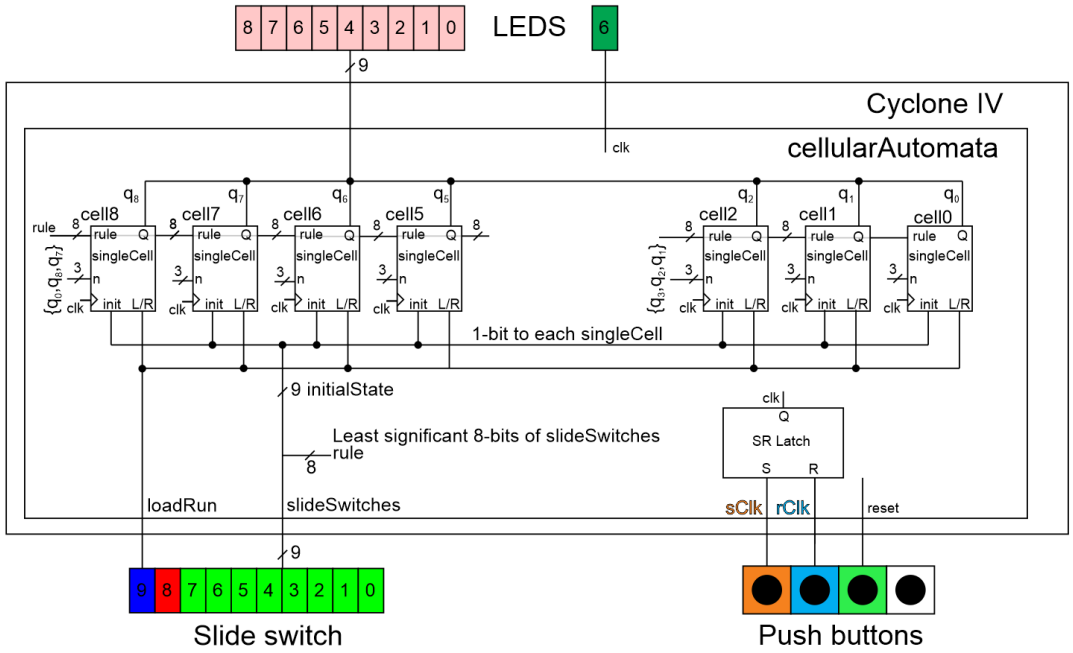


Figure 7.7: The system architecture of the cellularAutomata. Due to tight spacing, only the n inputs to cell8 and cell2 are shown.

The Verilog code for the cellularAutomata has been partially provided to you. You will need to complete the missing pieces. For this module, I want you to:

- Use the cellularAutomata.v file provided in the Canvas folder as the starting point.
- Make a vector for the *rule* and *initialState* from the *slideSwitches* input vector.
- Make a 3-bit vector input for each singleCell by appending its current state to the two neighbors current state using the "{}" operators. See the singleCell module for more information.
- Connect the ends of the CA together
  - Make cell 8 have cell 0 as its "left" neighbor in Figure 7.7
  - Make cell 0 have cell 8 as its "right" neighbor in Figure 7.7
- Use cross-couples NORs to realize the SR-latch. This means that you should have two lines of Verilog Code for the SR-latch, both starting with "assign".
- You are encouraged to use the generate statement to instantiate singleCell 1-7. Due the ring architecture, you will need to instantiate cells 0 and 7 individually. For an example of the generate statement, look at the adderSubtractor provided to you in a previous lab.

7.4 singleCell module:

The significant design problem in today's lab comes in this section, building the singleCell module. The module interface for the singleCell module in Figure 7.7 used some shorthand for the single names due to the space constraints. The internal organization and module interface for the singleCell module is shown in Figure 7.8. For example, the output currentState in this figure was called Q in Figure 7.7. You should be able to decipher the rest of the signal abbreviations used in Figure 7.7.

The most complex portion of logic in the singleCell module is the box labeled "nextState". This circuit has 11-bit of input and while it may seem a bit daunting at first, the Verilog code to realize this function is pretty straightforward when you have the right perspective. Table 7.2 lists all the combination of state for a cell and its 2 neighbors in the left column (n+ is the state of the neighbor to the left, n the state of the cell itself, and n- the state to the right). These 8 combinations are the case values in the always/case statement for the nextState logic. You need to know the output for each of these cases. As an example, complete the nextState column in Table 7.2 for Rule 90 using the information in Figure 7.2. In the "Rule bit" column in Table 7.2, generalize the nextState column to the bit values in the 8-bit rule vector that is passed into the singleCell module.

Table 7.2: The input/output relationship for the nextState functionality in Figure 7.8.

{n+, n, n-}	nextState	Rule bit
3'b000		
3'b001		
3'b010		rule[2]
3'b011		

{n+, n, n-}	nextState	Rule bit
3'b100	1	
3'b101		
3'b110		
3'b111		

For the singleCell module, I want you to:

- Use the singleCell.v file provided in the Canvas folder as the starting point.
- Use an always/case statement for the nextState logic
- Use the module definitions for:
  - Use the genericMux2x1 from a previous lab.
  - Use the dffNegEdge module provided in the Canvas lab folder for this lab.
- Provide meaningful names to the wires in the module.
- Properly tab-indent your code
  - Single level for wire declarations
  - Single level for component instantiations
  - Two levels for case statement
  - Three levels for case values

7.5 Pin Assignment

Use the image of the Cyclone V GX Board in Figure 7.1 and the information in the Cyclone V GX User Guide to determine the FPGA pins associated with the input and output devices used by the cellular automata module.

slide[9]			Clk LEDG6	
slide[8]			led [8] LEDR8	
slide[7]			led [7]	
slide[6]			led [6]	
slide[5]			led [5]	
slide[4]			led [4]	
slide[3]			led [3]	
slide[2]			led [2]	
slide[1]			led [1]	
slide[0]			led [0] LEDR0	

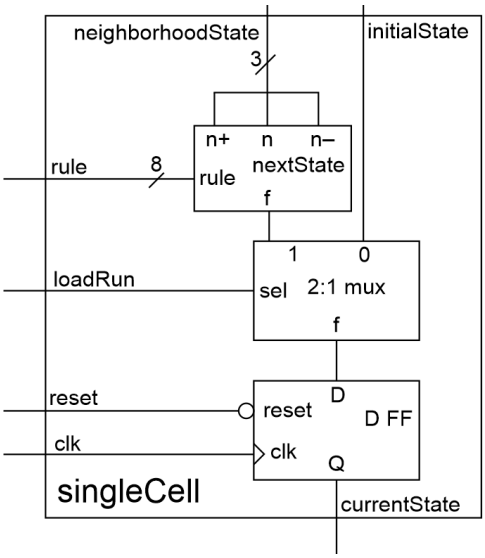


Figure 7.8: The architecture and module interface for the singleCell module.

sClk	Key[3]	
rClk	Key[2]	
reset	Key[1]	

Example Executions

To provide you with some examples to run on the Altera boards, the following table illustrates the evolution of the CA with different initial conditions and different rules.

- In the left most column contains several different rows
  - The row labeled "Rule #" is the rule that you will enter on the slide switches. The binary code of the rule is provided to make setting your dip switches easier.
  - The row labeled "start" is the initial load value stored in the CA. A value of "x" means that the initial load value doesn't matter (don't care).
  - The rows labeled "iteration" count successive positive edges generated by the clock.
- The column labeled "$q_8q_7q_6q_5$" is the output from the 4 most significant bits of the CA
- The column labeled "q_4" is the output from the middle bit of the CA
- The column labeled "$q_3q_2q_1q_0$" is the output from the 4 least significant bits of the CA

Rule 0: **0000 0000**	$q_8q_7q_6q_5$	q_4	$q_3q_2q_1q_0$
Start:	XXXX	X	XXXX
Final:	0000	0	0000
Rule 255: **1111 1111**			
Start:	XXXX	X	XXXX
Final:	1111	1	1111
Rule 90: **0101 1010**			
Start:	0000	1	0000
1st Iteration:	0001	0	1000
2nd Iteration:	0010	0	0100
3rd Iteration:	0101	0	1010
4th Iteration:	1000	0	0001
5th Iteration	1100	0	0011
6th Iteration	0110	0	0110
7th Iteration	1111	0	1111
Rule 90: **0101 1010**			
Start:	1011	0	1101
1st Iteration:	1011	0	1101
Final:	1011	0	1101
Rule 254: **1111 1110**		1111	1110
Start:	1000	0	0000
1st Iteration:	1100	0	0001
2nd Iteration:	1110	0	0011
3rd Iteration:	1111	0	0111
Final:	1111	1	1111

7.6 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.

Cellular Automata Module:

- Complete Table 7.1.

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

singleCell Module

- Complete Table 7.2.
- Link Verilog code for the body of the singleCell module (courier 8-point font single spaced), leave out header comments.

Overall design

- Run the testbench for the cellularAutomata 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

reset	default	Blue trace
rule	radix unsigned	Blue trace
sButton	default	Green trace
rButton	default	Green trace
clk	default	Yellow trace
currentLifeState	radix hex	Red trace

  I do not want the signals from the testbench, but rather the signals from inside the cellularAutomata module. You can do this in ModelSim, by expanding the cellularAutomata_tb instance in the left ModelSim and selecting "uut". Since uut is an instance of the cellularAutomata module, all the signals accessible in the cellularAutomata module are shown in the center Object. You can add signals using a drag-and-drop operation. Likewise, you can reorder the signals by dragging them. Your completed timing diagram should look something like Figure 7.9.

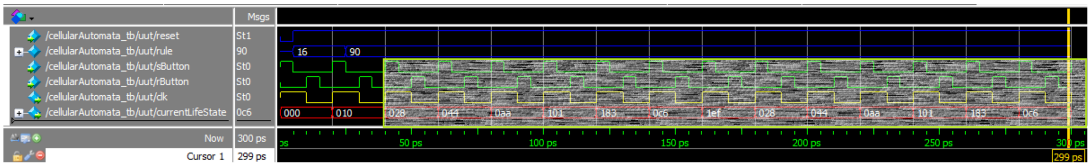


Figure 7.9: Partial timing diagram for the cellular automata.

Laboratory 8

Mod 10 Counter

8.1 Objective

The objective of this lab is to design a mod10 counter to represent the digits of a stopwatch.

8.2 Discussion

A stopwatch is a device that is used to measure time intervals, usually in competitive events. The stopwatch that you will be designing gets its input from two buttons. The stopwatch will measure down to a 1/10th of a second. The time will be displayed using 3 digits which will represent tenths of a second, unit second and tens of seconds. As a result, the stopwatch is limited to measuring intervals of time from 0.1 second to 99.9 seconds.

Before diving into the architecture of the datapath, you will need to first build an important building block, the mod10 counter.

8.3 Mod10 Counter

A mod 10 counter counts up from 0 to 9 and then rolls over back to 0 to count up again. The term "mod" comes from the word modulus. If you take a number x and form "x mod 10" you get the integer remainder after division by 10. For example, 12 mod 10 is equal to 2 because $12/10 = 1$ with a remainder of 2. Note that "x mod 10" will always produce a value between 0 and 9. Thus, a mod 10 counter will count up from 0 to 9 and then back to 0 to start over again.

You will build the mod 10 counter shown in Figure 8.1. The enb input enables the counter to count up on a rising edge of the clock. The synch input causes the mod10Counter to (synchronously) reset of the rising edge of the clock. The roll output indicated when the currentCount is going to roll-over from 9 to 0.

Table 8.1 is the truth table for the currentCount output. When the **enb** input is at logic 1, **currentCount** is incremented mod 10 on the next positive clk edge. When the **synch** input equals 1, **currentCount** goes to 4'b0000 on the next positive **clk** edge. The **synch** input

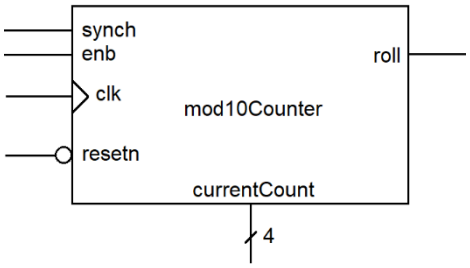


Figure 8.1: The high-level interface for the mod 10 counter.

takes precedence over the **enb** input, so if both are at logic 1 then **currentCount** goes to 4'b0000 on the next positive **clk** edge.

Table 8.1: The truth table for the currentCount output from the mod10Counter.

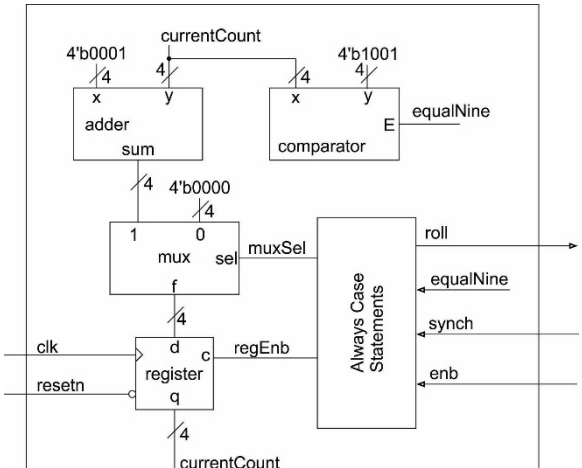
reset	clk	enb	synch	currentCount	Note
0	x	x	x	0	Asynch reset
1	0, 1, ↓	x	x	currentCount	No clk edge
1	↑	0	0	currentCount	Hold
1	↑	0	1	0	Synch reset
1	↑	1	0	(currentCount +1) mod 10	Count up
1	↑	1	1	0	Synch reset

Table 8.2 is the truth table for the roll output. If **enb** is logic 1 when the **currentCount** equals 9, the **roll** output equals logic 1. In all other cases the **roll** output should equal logic 0. Note, the roll output does not depend on the **clk**, it's a combinational logic block.

Table 8.2: The truth table for the roll output from the mod10Counter.

enb	currentCount	roll
1	currentCount <9	0
1	currentCount ==9	1

Now that you have a solid grasp of how the mod10Counter should work, let's turn our attention to how this is accomplished. The internal organization of the mod10Counter is shown in Figure 8.2.



You have been provided with the adder, comparator, mux, and register shown in Figure 8.2. In addition to wiring these building blocks together, you will need to define the logic inside the always/case block.

Use the truth tables in Table 8.1 and Table 8.2 along with the hardware organization in Figure 8.2 to fill in Table 8.3 .

Table 8.3: The truth table for the always/case logic inside the mod10counter.

enb	synch	equalNine	muxSel	regEnb	roll
0	0	0			
0	0	1	x		
0	1	0			
0	1	1			
1	0	0			0
1	0	1		1	
1	1	0			
1	1	1			

When you code your always/case statement, with a three-bit output and then have three *assign* statements that break this 3-bit output into individual bits for muxSel, regEnb and roll.

You will need to demonstrate that your mod10counter operates correctly by running the provided testbench. In order for you to verify correct operation, you need to understand what output the simulation should output so that you can compare that to what your simulation actually producing. Any difference between these two indicate an error (either in your understanding or circuit behavior) that need to be fixed. To do this complete the timing diagram in Figure 8.3 using the value from the testbench.

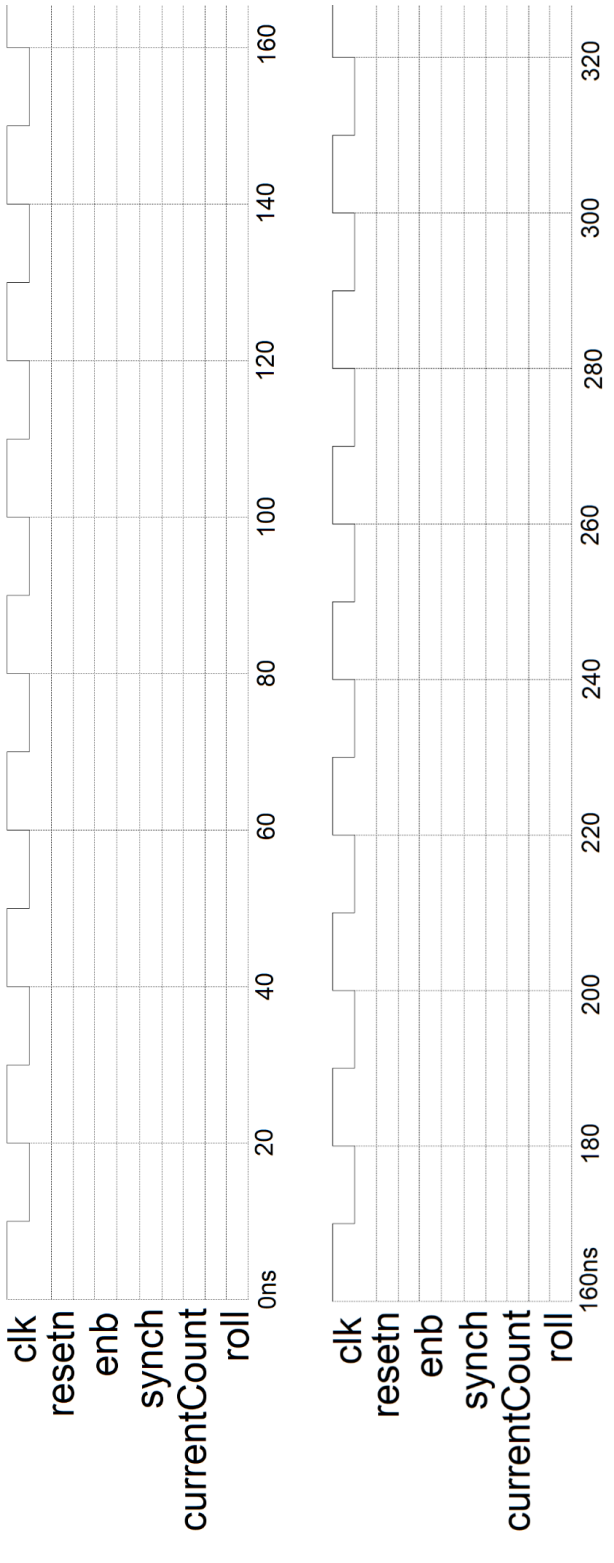


Figure 8.3: A timing diagram for you to fill out based on the testbench code for the mod10Counter. Note the diagram was too wide to be readable in one line, so it has been broken in half at 160ns to make it easier to you to read.

8.4 Do file

You have been required to run simulation for almost every lab. The goal of this requirement is not to just give you an extra task, but rather to expeditated your debugging of your Verilog. However, if you have a lot errors in your design, you may need to run the simulation multiple times. Setting up all the signals, their colors and radix can be a time consuming (and time wasting) task. The solution to this problem is to create a script that sets up all the signals, their order, colors and radixes. This script is called a do file.

Listing 1 shows a partial do file for the mod10counter testbench. The "#" symbol is used to denote comments – any text placed after them is ignored by the do file interpreter.

Listing 8.1: A partial do file for the mod10counter.

```
#########################################
# Search Internet ``modelsim command reference manual''
#########################################
vsim work.mod10counter_tb
restart -f
delete wave *

add wave -position end  sim:/mod10counter_tb/uut/clk
                <<you need more add waves here>>
add wave -position end  -radix unsigned -color greenyellow sim:/mod10counter_tb/uut/currentCount
```

The first three lines start the simulation and delete any waveforms that may be left over from a previous simulation. I included this in the do file because I sometimes rerun a simulation to observe the modules behavior at some earlier time.

The waveforms in the simulation are added using the "add wave" command. If you drag-and-drop a signal name into the waveform area, you will see the add wave command appear in the console are at the bottom of the ModelSim window. For example when I manually added the clk, I see:

```
add wave -position end sim:/mod10counter_tb/uut/clk
```

I then add the color of the wave and the radix in between "end" and "sim". See Listing 1 for a couple of examples. You should use notepad to edit the do file. Do NOT use a word processor like MS Word because they tend to change the extensions of files when you save.

After editing the do file, it should be stored in the following folder:

```
<projectDirectory>\mod10Counter\simulation\modelsim
```

Where <projectDirectory> is the path of your project. If this folder does not exit, then you need to have to run "Start Analysis and Elaboration" at least once. You know you are in the project directory when you see the project QPF file. You should run the do file immediately after launching ModelSim from Quartus. You no longer need to open the work directory and right mouse clicking on the testbench file and select "start simulation." Instead type:

```
do mod10counter_tbWaveSetup.do
```

When you run the do file, you should see something similar to Figure 8.4.

The ModelSim console allows tab completion, a feature that helps you fill-in the characters of long file name. To use tab completion, type the first few characters of a command/filename and then press Tab. If there are no other file names that match what you have typed in, the remainder of the file name will be auto-complete for you. If there is more than one filename choice, the command/filename will be completed up to the ambiguity and the console will provide a list of candidate filenames.

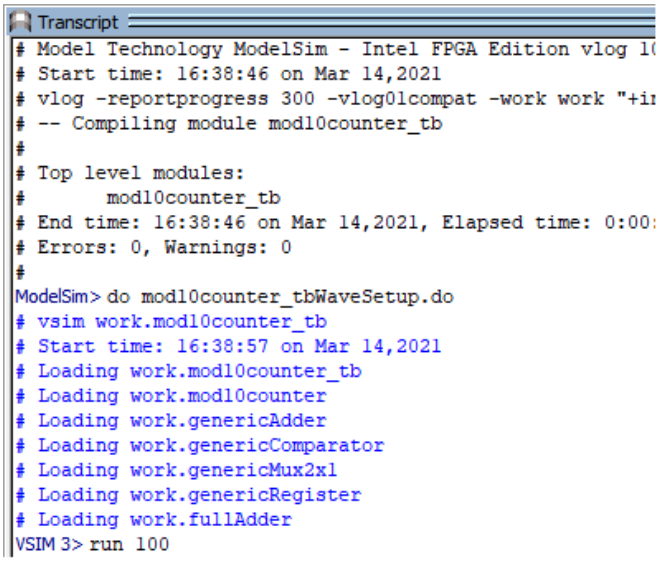


Figure 8.4: The console output when the mod10counter do file is run.

You can issue a variety of commands in the console window. On of my favorite is "run <time>" to simulate some amount of time. I found this VERY handy when debugging my Verilog code.

8.5 Testbench

Edit the do file provided on Canvas to produce the following output.

clk	default	green trace
reset	default	green trace
enb	default	gold trace
synch	default	gold trace
roll	default	yellow trace
currentCount	unsigned	greenyellow trace

You need to look at the values produced by the mod10counter and compare them against the values in Table 8.1. Look for discrepancies starting at time 0 and only advancing the simulation when everything is correct. You will have to transcend the design hierarchy to find the source of your errors. Most of my errors are due to incorrect wiring or modules – wrong names or wrong signal order.

8.6 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.

Mod10 Counter Verilog

- Complete Table 8.3 and Figure 8.3.
- Verilog code for the body of the mode10counter module (courier 8-point font single spaced), leave out header comments.
- Timing diagram of the mod10counter using do file in the testbench.
- Do file.

Laboratory 9

Stopwatch Datapath

9.1 Objective

The objective of this lab is to design a datapath to store and manipulate the information need to run a stopwatch.

9.2 Discussion

A stopwatch is a device that is used to measure time intervals, usually in competitive events. The stopwatch that you will be designing gets its input from two buttons. The stopwatch will measure down to a 1/10th of a second. The time will be displayed using 3 digits which will represent tenths of a second, unit second and tens of seconds. As a result, the stopwatch is limited to measuring intervals of time from 0.1 second to 99.9 seconds.

The stopwatch's behavior is dictated by its 2 buttons called S1 and S2 according to the finite state machine shown in Figure 9.1.

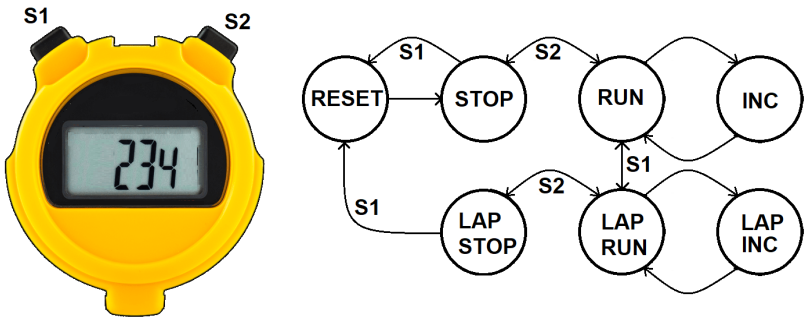


Figure 9.1: A digital stopwatch gets its input from 2 buttons and displays its output on a 7-segment display. The behavior of the stopwatch can be described by this finite state machine (FSM).

To make sense out of the FSM shown in Figure 9.1, its helpful to imagine timing a 4-person relay race. In this race, each athlete runs one lap and then pass a baton to the next runner. The time required for a runner to complete one lap is called their split time. In order to measure each runner's split time, you need to be able to stop the displayed time while allowing the stopwatch to continue to run its internal timer. This is called a lap feature. Let's explore how the lap feature works by timing the mile relay at a Mines track meet. As we go through this scenario, reference the finite state machine in Figure 9.1.

Prior to the start of the race, you push button S1 putting the stopwatch into the RESET state. This clears the internal timer and the displayed time. The stopwatch automatically goes back to the STOP state. You are ready for the start of the race. You are ready when the gun goes off and immediately press button S2 putting the stopwatch into the RUN state. The internal timer is keeping track of the elapsed time and you this the displayed time changing to reflect the internal timer. As soon as the first runner who is finishing their lap hands the baton to the second runner, you press button S1 putting the stopwatch into the LAP RUN state. This causes the displayed time to stop, showing the time at the instant you pressed button S1, while simultaneously allowing the internal timer to keep running. The internal timer is now keeping track of the elapsed time since the start of the race. You calmy write down the displayed time on your clip board (made easier because it is not changing) and then press button S1 putting the stopwatch back into the RUN state. You repeat this process until the last runner comes in. As soon as they do, you press button S2 stopping the internal timer and showing the time the last runner crossed the finish line.

In this scenario we did not put the stopwatch into the LAP STOP state – this state would stop the internal timer and keep the displayed time the moment the S1 button was pressed when the stopwatch was in the RUN state. You should also note, we did not talk about the INC and LAP INC states. Let's explore them now.

Clearly, the stopwatch does not increment the stored and displayed time at 50Mhz. The stored and displayed time count up every 10^{th} of a second. This is managed by the counter/-comparator at the top of Figure 9.2 which asserts the **tenth** signal every 10^{th} of a second. This signal tells the FSM in Figure 9.1 to transition from the RUN state to the INC state. In the INC state the stored time is incremented and the counter at the top of Figure 9.2 is reset back to 0. The LAP INC state performs a similar function for the LAP state.

To summarize:

- RESET – Reset the internal and displayed time values.
- STOP – Stop the 10^{th} second timer and hold the displayed time.
- RUN – Run the 10^{th} second timer and display the stored time.
- INC – Increment the stored time
- LAP RUN – Run the 10^{th} second timer and hold the displayed time.
- LAP INC - Increment the stored time.
- LAP STOP – Stop the 10^{th} second timer and hold the displayed time.

9.3 Datapath Architecture

The datapath for the stopwatch is shown in Figure 9.2. The behavior of the datapath to perform the functions of a stopwatch follows.

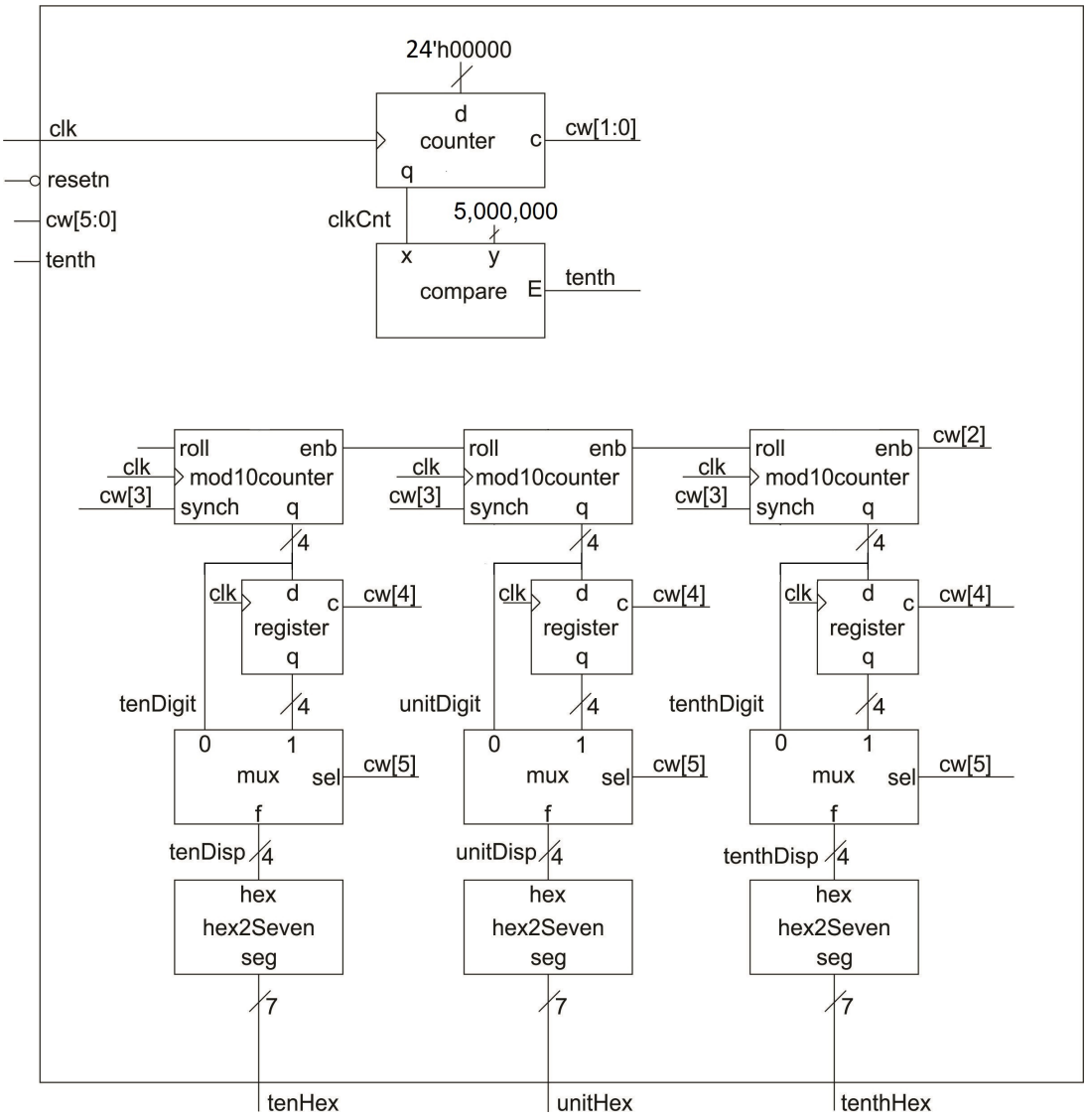


Figure 9.2: The datapath for the stopwatch has a 6-bit control word and displays the time on three 7-segment displays.

The internal time is formed by the counter/comparator combination at the top of Figure 9.2. The **clk** input from the Development Board runs at 50MHz. This will be the rate at which the **clkCnt** output from counter counts up. When **clkCnt** counts from 0 to 5,000,000 then $1/10^{\text{th}}$ of a second has gone by. The **roll** signal indicates when $1/10^{\text{th}}$ of a second has gone by. The control unit you build will build in the next lab, will use this signal to coordinate incrementing the bank of mod 10 counters in the datapath. Note that the bank of mod 10 counters are incremented in a state called INC, that is not shown in Figure 9.1. The output of the mod10 counters, **tenDigit**, **unitDigit** and **tenthDigit** can be latched-up in the register bank so that the datapath can hold the displayed time while still allowing the bank of mod 10

counters to keep track of the elapsed time. The multiplexer controls what is displayed. The 4-bit digit representing a time digit, **tenDisp**, **unitDisp** and **tenthDisp** are converted into a 7-segment pattern before leaving the datapath.

To better understand the datapath, construct the control word for each state. In order to help you, the actions associated with each state are outlined below. In some cases, you will need to make decisions about the control bits. Use your understanding of the datapath's operation and your intuition of how you would want to the datapath to work. If this is uncomfortable, please understand that it's important for you to learn how to make design decisions on your own so that you can reach your full potential as an engineer.

- RESET – clear the values in the registers and counters
- STOP – hold the timer counter and display the mod 10 counters
- RUN – allow the timer counter to count up and display the mod 10 counters
- INC – clear the timer counter and increment the mod 10 counters
- RUN2LAP – Latch up the mod 10 counters in the lap register
- LAP RUN – allow the timer counter to count up, display the latched time
- LAP INC – clear the timer counter and increment the mod 10 counters
- LAP STOP – stop the timer counter and display the latched time

Table 9.1: Control word table for the datapath shown in Figure 9.2

	cw[5] 2x1 mux	**cw[4] lap register**	**cw[3] mod10 reset**	**cw[2] mod10 count**	**cw[1:0] timer counter**
	0 = mod10	1 = load	1 = reset	1 = count up	11 = load
	1 = register	0 = hold	0 = hold	0 = hold	10 = count up
					01 = not used
					00 = hold
RESET			1		
STOP					
RUN					
INC					
RUN2LAP				0	
LAP RUN					10
LAP INC		0			
LAP STOP	1				

Now that you have the control word figured out, you need to write the Verilog code for the datapath. For the datapath module, I want you to:

- Use the datapath.v file provided in the Canvas folder as the starting point.

  - Use the module definitions from previous land and the Canvas lab folder for this lab.

- Provide meaningful names to the wires in the module.
- Properly tab-indent your code
  - Single level for wire declarations
  - Single level for component instantiations
  - Two levels for case statement
  - Three levels for case values

9.4 Datapath Simulation

Before you download your completed datapath to the Development Board, you are going to perform extensive simulations to uncover as many bugs as possible. Trust me, errors are much, much easier to find in a simulation.

There is a practical consideration that will make the simulation more manageable. The counter-comparator combination in Figure 9.2 acts as a clock divider circuit. The counter counts up at 50MHz (the main oscillator frequency). The comparator checks when the count reaches 5,000,000 = 0x4C4B40, meaning that a $1/10^{th}$ of a second has gone by. You do not want to have to run the counter to 5,000,000 in your simulation, it would just take too long. To replace the 5,000,000 constant you need to go into the datapath.v module and modify the following:

- Set `localparam tenthSecondConstant` to 4'h000002 (an arbitrary small constant)
- Set the parameter `N` to 4, this sets the word size of the counter/comparator hardware.

You can use the parameter **N** as the generic parameter in component instantiation for generic components. For example, in my datapath, I have the following genericCounter instantiation - the use of the parameter value #(N) as the width of the counter is legal syntax in Verilog.

```
genericCounter #(N) tenthSecondCounter(clk, resetn, zero24, cw{[}1:0{]}, clkCount);
```

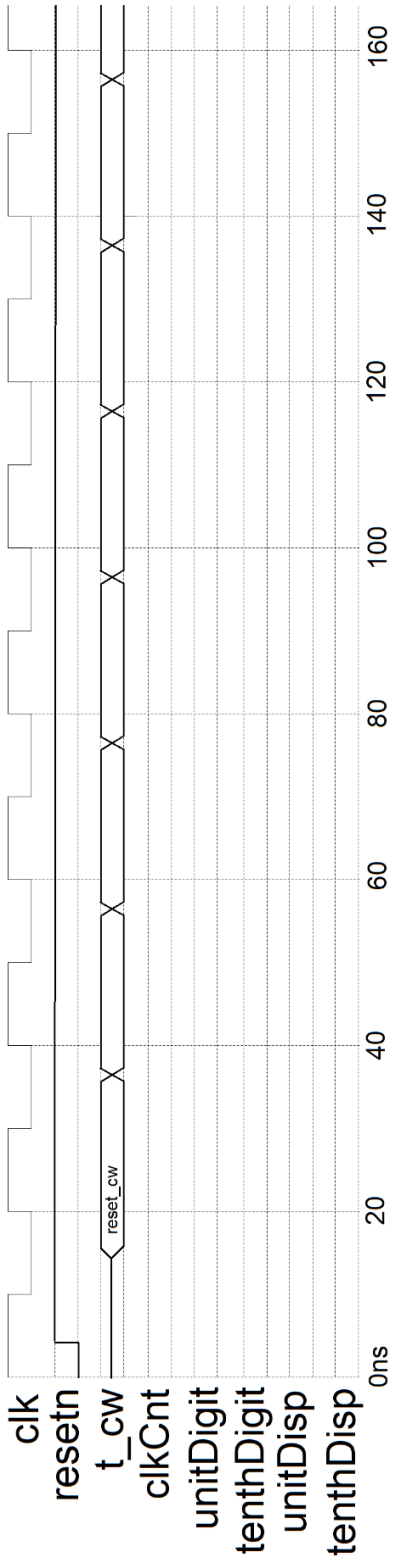
Make a note to yourself to modify **N** and **tenthSecondConstant** values before synthesis. This is also mentioned later in this document.

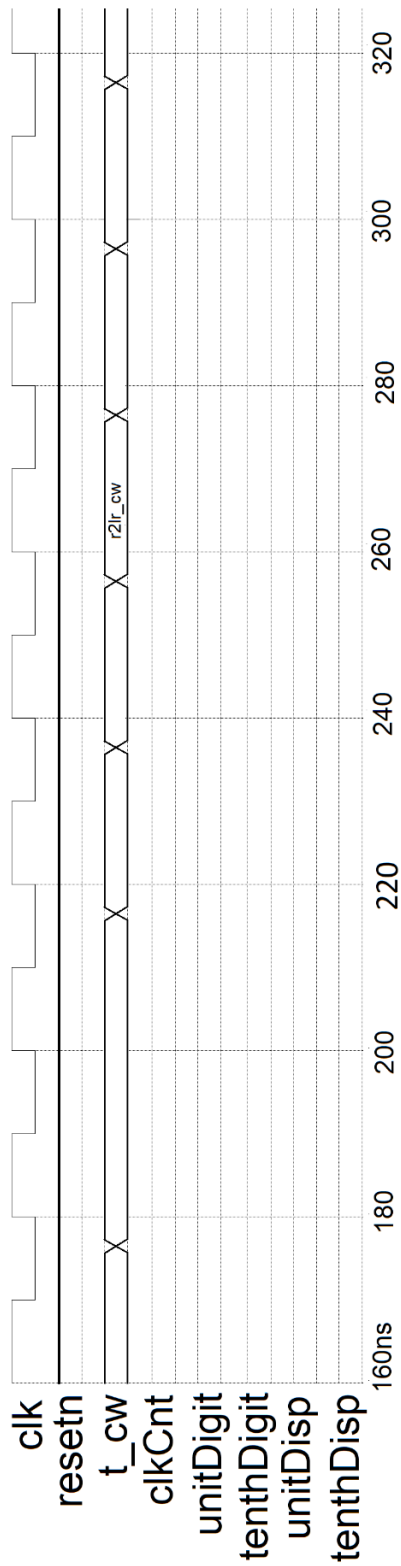
Next modify the control words defined in the datapathLab09_tb.v file using the values from Table 9.1.

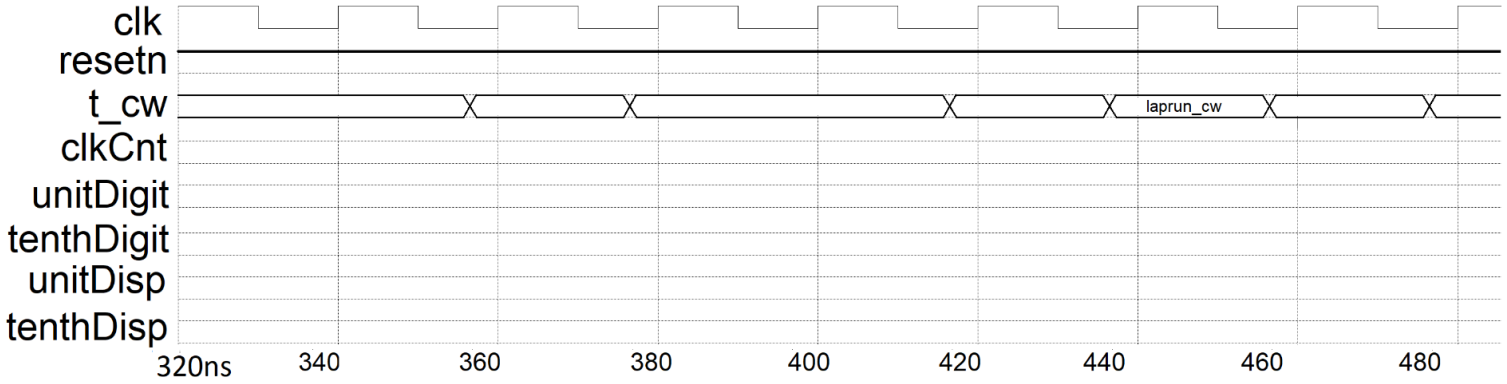
Finally, you need to understand what output the simulation should output so that you can compare that to what your simulation actually producing. Any difference between these two indicate an error (either in your understanding or circuit behavior) that need to be fixed.

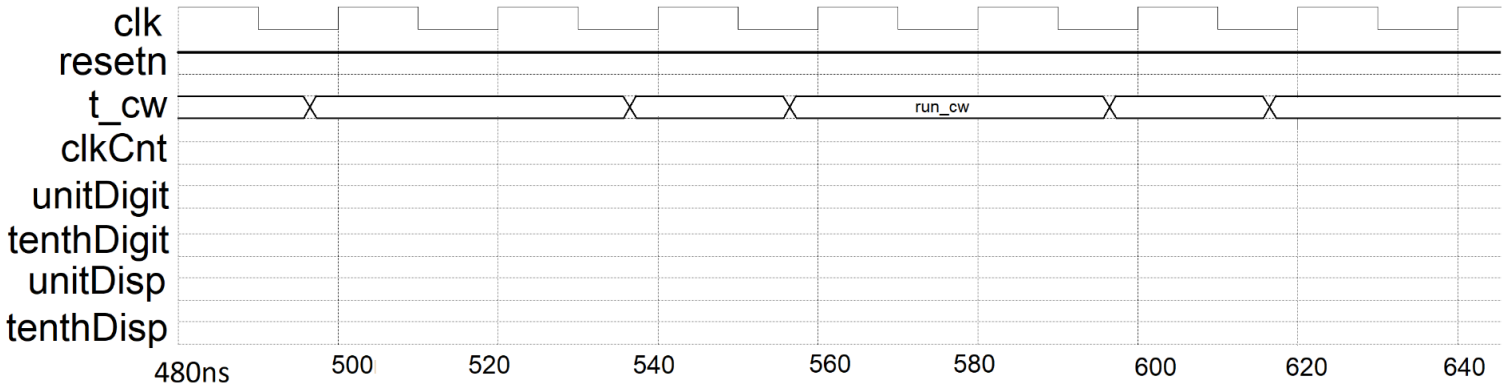
To do this complete the entries in the following timing diagram figures. Use the code in the testbench to figure out the values for the control word and how long they are held. The control word does not necessarily change every 20ns. When it does not, you can just rewrite the control word or edit the image to connect adjacent cells.

The $display statement in the testbench prints out a message when that line of the simulation is reached. Including these statements helps you to understand where your simulation is at and what behavior to expect.


clk
resetn
t_cw
clkCnt
unitDigit
tenthDigit
unitDisp
tenthDisp
reset_cw
0ns
20
40
60
80
100
120
140
160


clk
resetn
t_cw
clkCnt
unitDigit
tenthDigit
unitDisp
tenthDisp
r2lr_cw
160ns
180
200
220
240
260
280
300
320


clk
resetn
t_cw
clkCnt
unitDigit
tenthDigit
unitDisp
tenthDisp
laprun_cw
320ns
340
360
380
400
420
440
460
480


clk
resetn
t_cw
clkCnt
unitDigit
tenthDigit
unitDisp
tenthDisp
run_cw
480ns
500
520
540
560
580
600
620
640

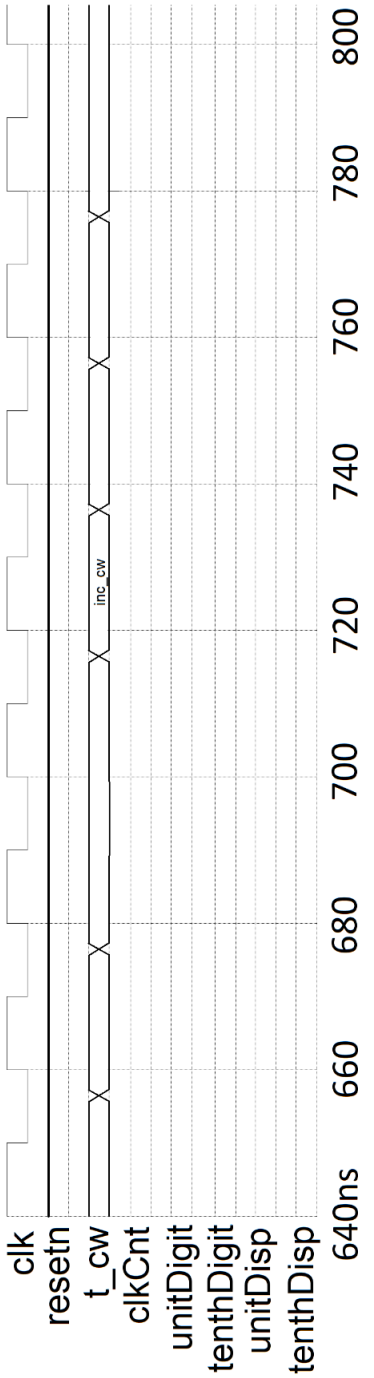


Figure 9.3: Complete the timing diagram using the control words found in the testbench.

Run the testbench for the datapathLab09_tb module using the do file provided. Produce a timing diagram ordered from top to bottom as:

clk	default	green trace
resetn	default	green trace
cw	hex	yellow trace
tenth	default	gold trace
clkCnt	hex	red trace
tenthDigit	hex	green trace
tenthDisp	hex	green trace
unitDigit	hex	greenyellow trace
unitDisp	hex	greenyellow trace

In order to simplify the process of setting up the waveforms, you can script all the wave adds, assignment of radix and colors using the provided DO file as follows:

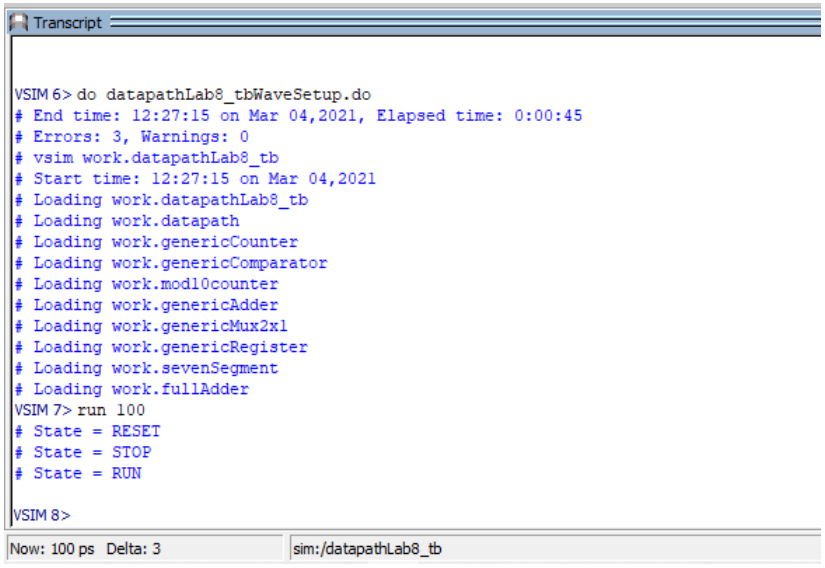
- Download "datapath_tbWaveSetup.do" into the:

  <projectDirectory>\<projectname>\simulation\modelsim

- Open datapath_tbWaveSetup.do file using Notepad. The syntax is pretty straight forward and corresponds to the text displayed in the ModelSim console window as you make modifications to the waveforms.

- You should use the do file immediately after you launch Model Sim by typing:

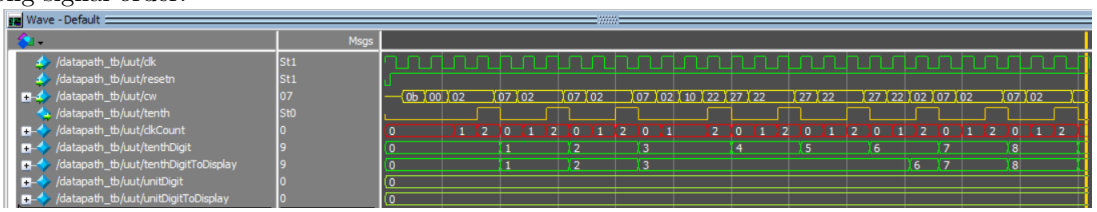
  - VSIM 3> do datapath_tbWaveSetup.do



- You can type "run <time>" in this area (as shown) to simulate some amount of time. I found this VERY handy when debugging my Verilog code.

- Also note that the console has tab completion. This allows you to type the first few characters of a command/filename and press Tab to fill in the rest of the command/filename. If there is more than one choice, the command/filename will be completed up to the ambiguity.

You need to look at the values produced by the datapath and compare them against the values in Table 2. Look for discrepancies starting at time 0 and only advancing the simulation when everything is correct. You will have to transcend the design hierarchy to find the source of your errors. Most of my errors are due to incorrect wiring or modules – wrong names or wrong signal order.



9.5 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.

Datapath

- Complete Table 9.1.
- Verilog code for the body of the datapath module (courier 8-point font single spaced), leave out header comments.
- Complete timing diagram from Figure 9.3.
- Copy of your simulation timing diagram. Use the signal color and order specified.
- Demonstrate that your datapath and do file work in ModelSim.

Laboratory 10

Stopwatch Control Unit

10.1 Objective

The objective of this lab is to design a control unit to control the datapath created in the previous lab so that, together, they can run the stopwatch.

10.2 Discussion

From the previous lab, you should be familiar with the operation of our stopwatch. Briefly, our stopwatch allows a user to measure elapsed time and lap times of a competitive events. Our stopwatch measures time in increments of a tenth of a second, unit second and tens of seconds. Control input comes from 2 buttons called S1 and S2 according to the incomplete finite state machine shown in Figure 10.1.

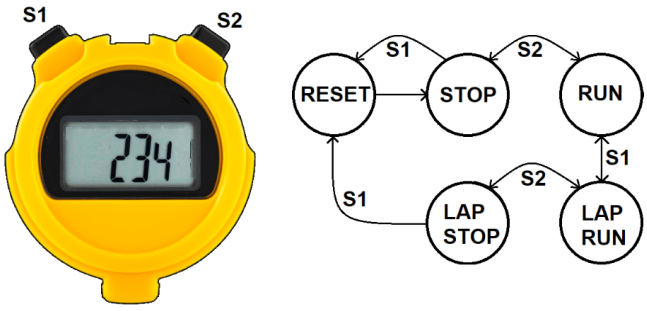


Figure 10.1: A digital stopwatch gets its input from 2 buttons and displays its output on a 7-segment display. The behavior of the stopwatch can be described by this finite state machine (FSM).

There are two reasons that the FSM shown in in Figure 10.1 is incomplete, it does not

reflect the logic level of the buttons on the FPGA development board nor does it account for the time the user holds the S1 and S2 buttons down.

Figure 10.2 shows the schematic of the buttons (key1 ... key4) on the C5G development board that you will use to control the operation of the stopwatch; the S1 and S2 buttons in Figure 10.1. The buttons shown in the schematic are in their nominal position – not being pressed by a user. You should note that the contacts of each button (open circles) are disconnected. This leaves the right side of the button connected to VCC through a resistor as well as the Altera FPGA. This resistor is called a pull-up resistor, and as its name implies, pulls the voltage on the right side of the push button up to VCC – logic 1. When a button is pressed, the contacts are connected and the right side of the push button is connected directly to ground, forcing the voltage on the respective FPGA pin to logic 0.

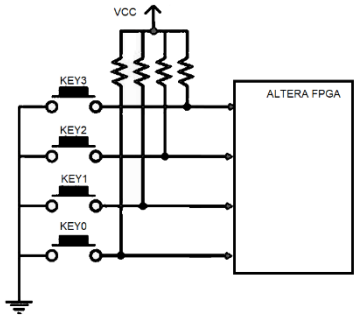


Figure 10.2: FPGA development board buttons used to operate the stopwatch.

To summarize the operation of the buttons shown in Figure 10.2. When a user presses a button, the value on the respective FPGA pin is logic 0. When a button is not pressed, the logic level of the corresponding FPGA pin is logic 1.

You will explore the reason that effect of the user holding down a button in the next section which introduces you to the finite state machine that governs the behavior of your stopwatch.

10.3 Control unit architecture

The control unit for the stopwatch is shown in Figure 10.3.

The arcs between states are labeled with a Boolean condition, which when true, causes the FSM to make that transition. So for example, if the FSM is in the RUN state and the S2 button is pressed (buttons output 0 when pressed hence the arc labeled S2'), the FSM will transition to the R2S state (and stay there as long as the button is held down). When a number "2" appears in the middle of a state name, the number denotes the word "to" which is intended to mean moving from one state to another. The abbreviated names of the source/destination states are written on left/right side of the number "2" respectively. So for example, the state R2S stands for Run to Stop.

These intermediate states are needed because the action of a user pressing a button takes a long time from the perspective of the 50MHz clock on the FPGA development board. For example, consider the incorrectly designed FSM shown in Figure 10.4. The intention of this design is to have the FSM transition from the STOP state to the RUN state when the user pressed the button S2 (remember that when S2 is pressed it outputs a logic 0, hence the arc labeled S2'). What actually happens is that the FSM rapidly toggles between the STOP and

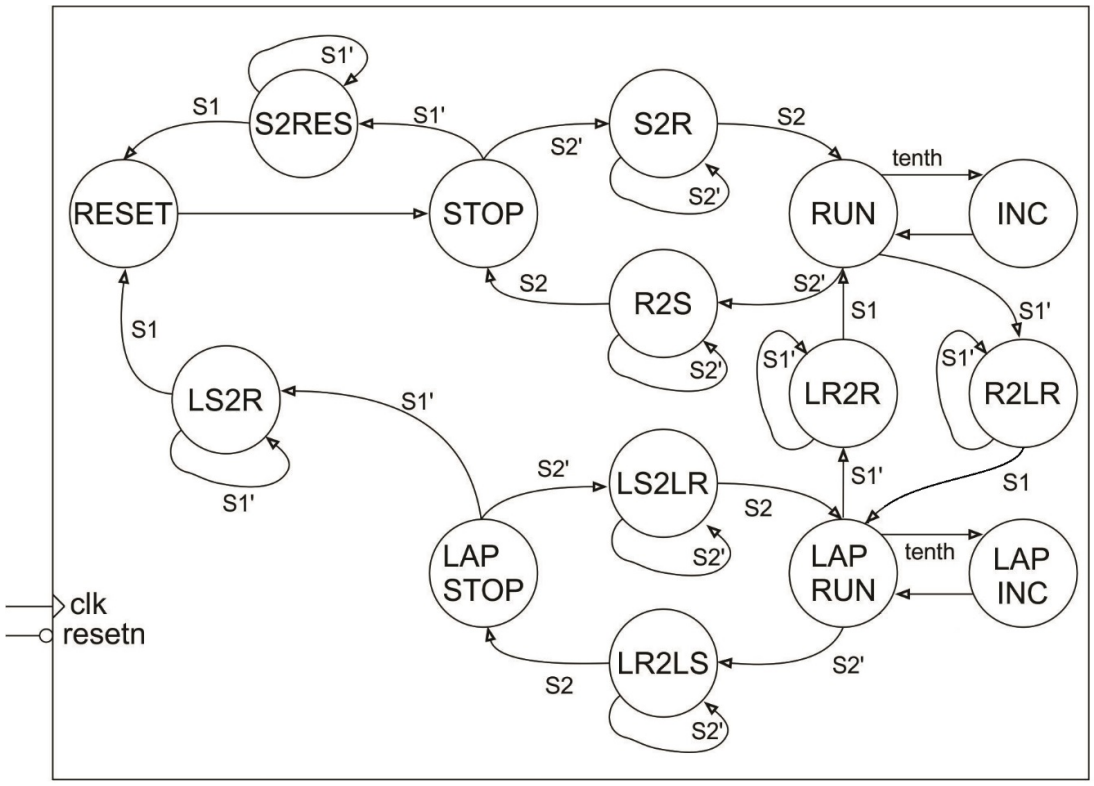


Figure 10.3: The control unit for the stopwatch needs to account for the time the user holds the button down.

RUN states at 50MHz while the S2 button is held down. When the user releases the S2 button there is a 50/50 chance that the FSM will end-up in the STOP or RUN state.

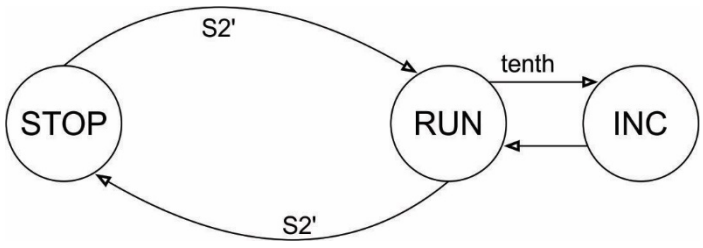


Figure 10.4: An improperly constructed FSM for the stopwatch.

Before you dive into writing the Verilog code for the control unit, you need to complete Table 10.1, the control word value for each state. You have already completed some of these control words in the previous lab. Most of the new states are the states with "2" in them. We will call these "transitional states" because they move the control unit between stopwatch modes. When writing the control words in Table 10.1 you must follow the following 2 rules.

1) Assign the cw[5] bit for transitional states the same value as the cw[5] value for the source

state. So for example in the R2LR state, set cw[5] to 0 because the cw[5] value in the source state, RUN, is 0.

2) Only have the timer counter counting up while the control unit is in the RUN or LAPRUN states. We don't want the timer counting up in intermediate states because these states do not have an "INC" associated with them, and as a consequence, the tenth pulse from the timer counter would most likely be missed.

Table 10.1: Control word table for the stopwatch finite state machine shown in Figure 10.3.

	cw[5] 2x1 mux	cw[4] lap register	cw[3] mod 10 reset	cw[2] mod10 count	cw[1:0] timer counter
	0 = mod10	1 = load	1 = reset	1 = count up	11 = load
	1 = register	0 = hold	0 = hold	0 = hold	10 = count up
					01 = not used
					00 = hold
RESET				0	
STOP					
S2RES					
S2R					
RUN	0				
R2LR	0				
R2S					
INC					
LAPRUN					10
LR2R					
LR2LS					
LAPINC					
LAPSTOP			0		
LS2R					
LS2LR					

After you complete the control word table, you are ready to write the Verilog for the control unit. This file will have the following main sections.

- Port description – This has been provided to you.
- Control word values – This is a set of localparam statements, one for each state. You will define the control word output for each state. This will include all the control words that you derived in the previous lab, plus some new control words for the intermediate states. For example, the following is the control word for the STOP state.

```
localparam STOP_CW = 6'b000000;
```

- State codes – Each state needs to be assigned a unique binary value. It does not matter what code you assign which state. These codes are mostly invisible. That said, you will want them handy when performing the simulation so that you know which state the control unit is in. For example, the following is the state code that I used for the STOP state. Note, your codes can be different.

```
STOP_STATE = 4'b0010,
```

- Reset logic – this has been provided to you.
- Output logic – This is an always/case statement that has one case for each state and simple associated the control word output with the appropriate control word for the state that the FSM is currently in. For example, the following is the case statement that I used to associate the STOP state control word with the STOP state code.

```
STOP_STATE: cw = STOP_CW;
```

- Next state logic – This is an always/case statement that has one case for each state. It embodies the logic in Figure 10.3 using if/then statements. Let's redraw Figure 10.3 by focusing on the states connected to the STOP state by transition arcs in Figure 10.5.

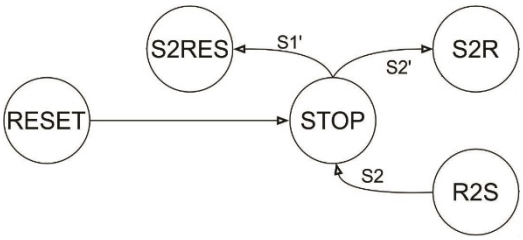


Figure 10.5: All the states and with a transition arc connected to the STOP state.

The next state logic for the STOP state embodies the 2 transition arcs leaving the STOP state in Figure 10.5. If the FSM is in the STOP state and the S1 input equals 1 then the FSM should go to the S2RES state. This logic and the transition to the state S2R is provided in the following code snippet. You must specify every possible next state, even when the next state does not change. Thus, I prefer to use a case statement and leave the default case to be when the next state is equal to the current state.

```
STOP_STATE:
        begin
                case({S2,S1})
                        2'b10: nextstate = S2RES_STATE;
                        2'b01: nextstate = S2R_STATE;
                        default: nextstate = STOP_STATE;
                endcase
        end
```

When writing code for the control unit, I want you to:

- Use the controlUnit.v file provided in the Canvas folder as the starting point.
- Provide meaningful names to the wires in the module.
- Properly tab-indent your code. You can use View -> Show White Space
  - Single level for wire declarations
  - Single level for component instantiations
  - Two levels for case statement
  - Three levels for case values

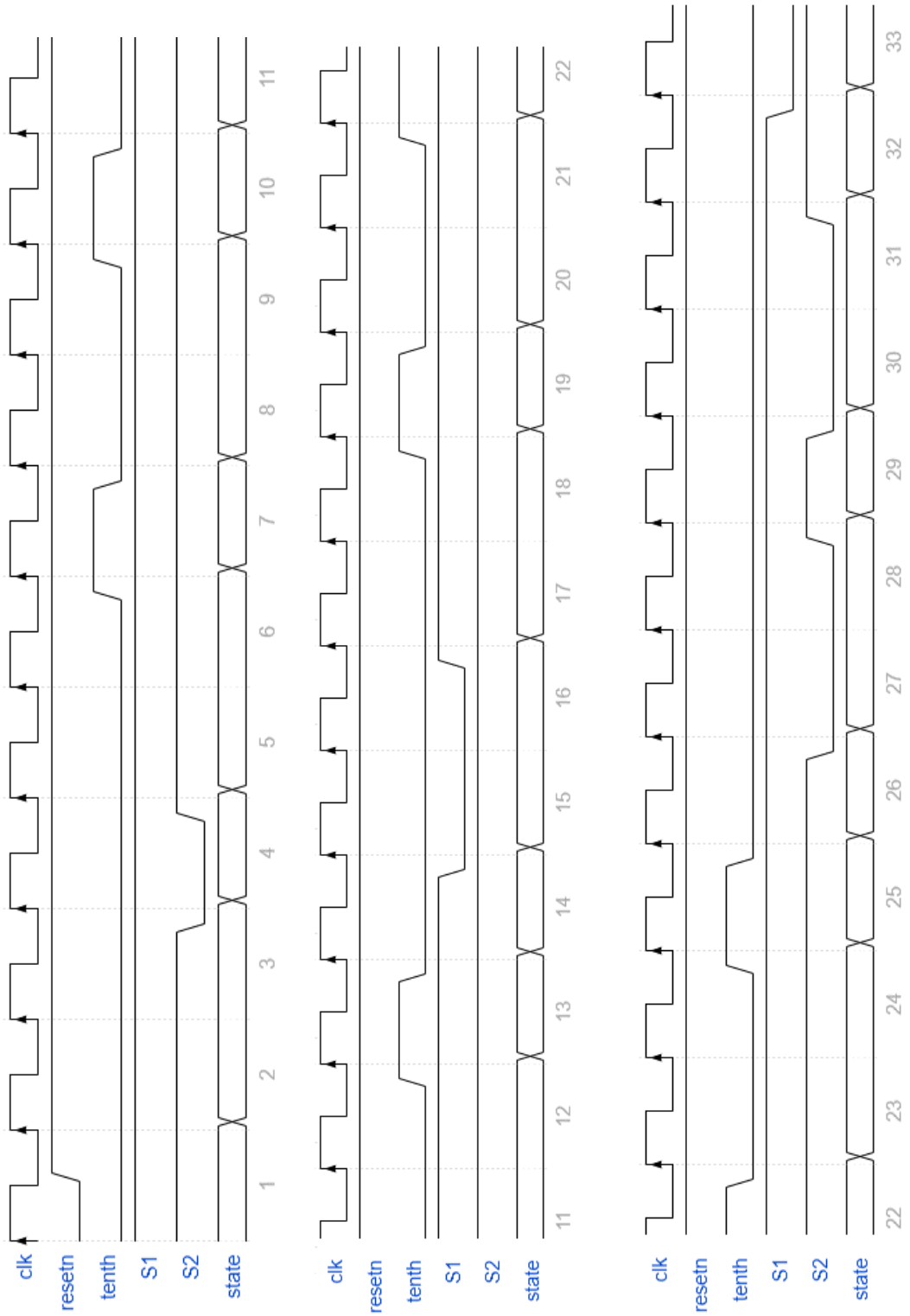
Compile the Verilog code for the control unit, look for and remove the errors. When the control unit compiles cleanly, it's time to simulate to check that it behaves correctly.

10.4 Control simulation

Before you download your completed control unit to the C5G boards, you are going to perform an extensive simulation to uncover as many bugs as possible. Trust me, errors are much, much easier to find in a simulation. The goal of this simulation is to cover every transition arc in Figure 10.3 so that you can be sure your code is working correct.

The timing diagrams shown in Figure 10.6 show the tenths, S1 and S2 signals as they are manipulated by the testbench. These signals will affect the state of the control unit according to Figure 10.3. Your task is to use these signals to determine what the state the control unit is in.

The goal of the testbench was to cover every transition arc in Figure 10.3. This goal was not achieved; one transition arc was not taken in the testbench, which one was it?


clk
resetn
tenth
S1
S2
state
1 2 3 4 5 6 7 8 9 10 11
11 12 13 14 15 16 17 18 19 20 21 22
22 23 24 25 26 27 28 29 30 31 32 33

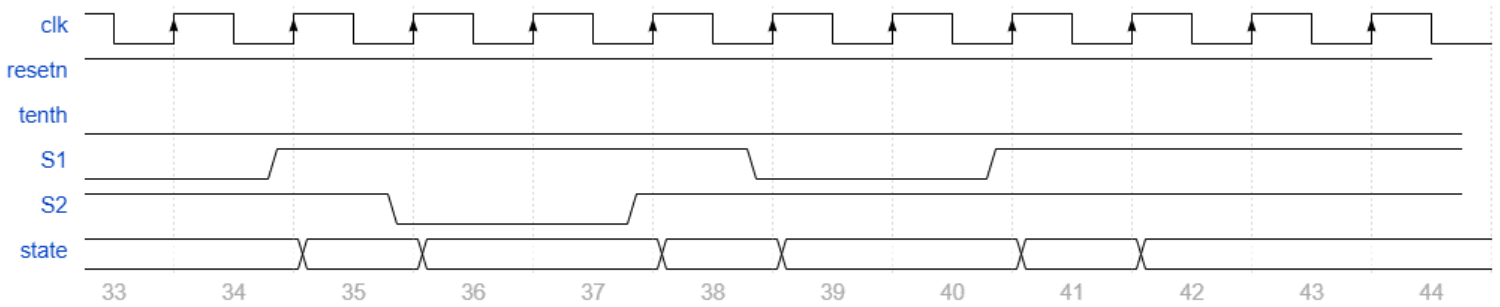


Figure 10.6: Timing diagram for the testbench for the testbench simulation

.

Now that you have an understanding of what the testbench should do, it's time to run the simulation. Complete the provided do file to simplify testing your control unit. Your timing diagram should have the following waveforms.

clk	default	green trace
resetn	default	green trace
cw	hex	yellow trace
tenth	default	orange trace
S2	default	orange trace
S1	default	orange trace
states		
	stop/stop2reset/reset/stop2run	red trace
	run/inc/run2lapRun/run2stop	yellow trace
	lapRun/lapInc/lapRun2run/lapRun2lapStop	orange trace
	lapStop/lapStop2run/lapStop2LapRun	green trace

When editing the do file for this lab, note the following.

- Correct the waveform names as needed. This might happen if you name a signal differently than I did.
- Create alias' for each state code. When you coded the control unit by assigning an arbitrary 4-bit value to each state. In Listing 10.1 you will associate each of these 4-bit values to a string and a color. The string should correspond to the name of the state and the color to something identifiable when the simulation is running. For example, you may remember that I assigned STOP_STATE = 4'b0010. This corresponds to the line "4'b0010 "STOP" -color red," in Listing 10.1

  Listing 10.1: Creating alias' for the binary codes of states in the do file uses requires knowing the binary code of each state, the name of each state and the color for each state.

```
radix define States {
                ...
        4'b0010 "STOP"    -color red,
                ...
        -default hex
        -defaultcolor white
}
```

- This will produce a much more meaningful representation like that shown in Figure 10.7 of the state when you run the simulation.

Run the entire simulation and compare the simulation output to Figure 10.6. Use the results of the simulation to either correct Figure 10.6 or to correct your controlUnit.v code.

10.5 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

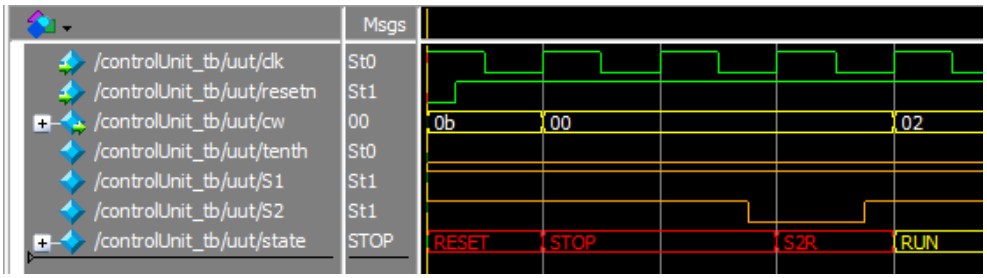


Figure 10.7: A small segment of the testbench simulation showing how Listing 10.1 encodes state names.

was derived. In addition to this submission, you will be expected to demonstrate your circuit at the beginning of your lab section next week.

Control Unit Design

- Completed Table 10.1.
- Completed Figure 10.6.
- Verilog code for the body of the control unit module (courier 8-point font single spaced), leave out header comments.

Control Unit Simulation:

- Produce a timing diagram of the testbench simulation using the format specifiers given here .
- Demonstrate your simulation and your do file works in ModelSim.

Laboratory 11

Stopwatch Datapath and Control

11.1 Objective

The objective of this lab is to unify the stopwatch datapath and control units into a working design that runs on the FPGA development board.

11.2 Discussion

From the previous lab, you should be familiar with the operation of our stopwatch. Briefly, our stopwatch allows a user to measure elapsed time and lap times of a competitive events. Our stopwatch measures time in increments of a tenth of a second, unit second and tens of seconds. Control input comes from 2 buttons called S1 and S2 according to the finite state machine shown in Figure 11.1.

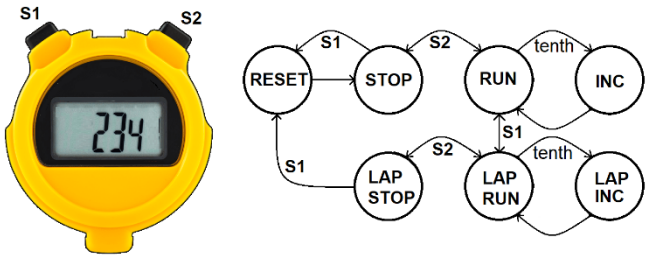


Figure 11.1: A digital stopwatch gets its input from 2 buttons and displays its output on a 7-segment display. The behavior of the stopwatch can be described by this finite state machine (FSM).

Figure 11.2 shows how you will implement the idea presented in Figure 11.1 using the FGPA development board.

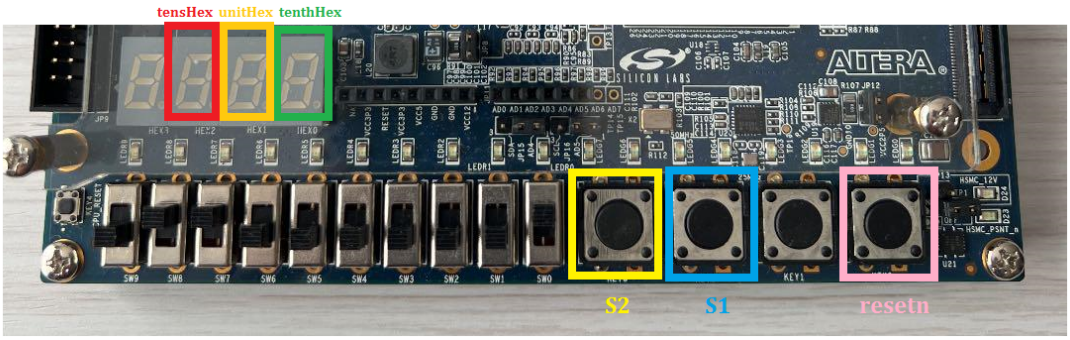


Figure 11.2: The stopwatch user interface as implemented on the FPGA development board.

Two of the buttons will act as the stopwatch buttons and 3 of the 7-segment displays will show the time. The reset signal is connected to a button and the 50MHz clock is on the FPGA development board but does not require any user interaction.

11.3 Stopwatch architecture

In the previous 2 labs you have created the datapath and control unit for the stopwatch. Using these 2 components as building blocks, the architecture for the stopwatch, shown in Figure 11.3, is almost trivial; the Verilog file for the stopwatch contains 2 component instantiations of the datapath and controlUnit.

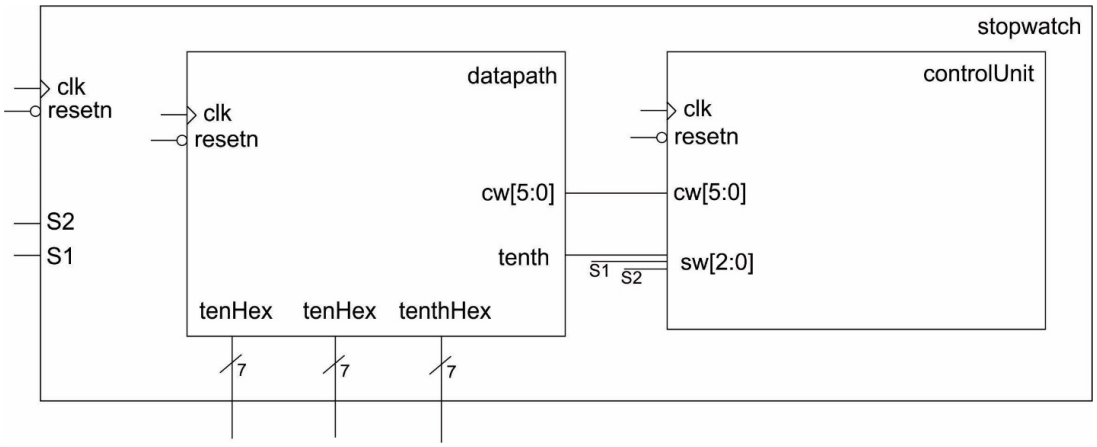


Figure 11.3: The architecture for the stopwatch consists of a datapath and controlUnit.

The only minor complication is combining the tenth output from the datapath with the S2 and S1 signals coming in from the buttons into a 3-bit signal sent to the status word input of the control unit.

Use the starter code provided on Canvas to complete the stopwatch module. While you are at it, download the stopwatch testbench provided on Canvas, and make the testbench the top-level entity.

11.4 Stopwatch Simulation

Before you download your completed control unit to the development boards, you are going to perform extensive simulations to uncover as many bugs as possible. Errors are much, much easier to find in a simulation. First you will need to understand what the testbench simulation is supposed to do.

The timing diagrams in Figure 11.4 show the S1 and S2 signals as they are manipulated by the testbench. The tenth signals will be asserted by your datapath but are included to help you. Your task is to fill in the symbolic name of the state that the FSM is in as well as the clkCount value, the mod10Counter outputs and the values displayed on the 7-segment displays.

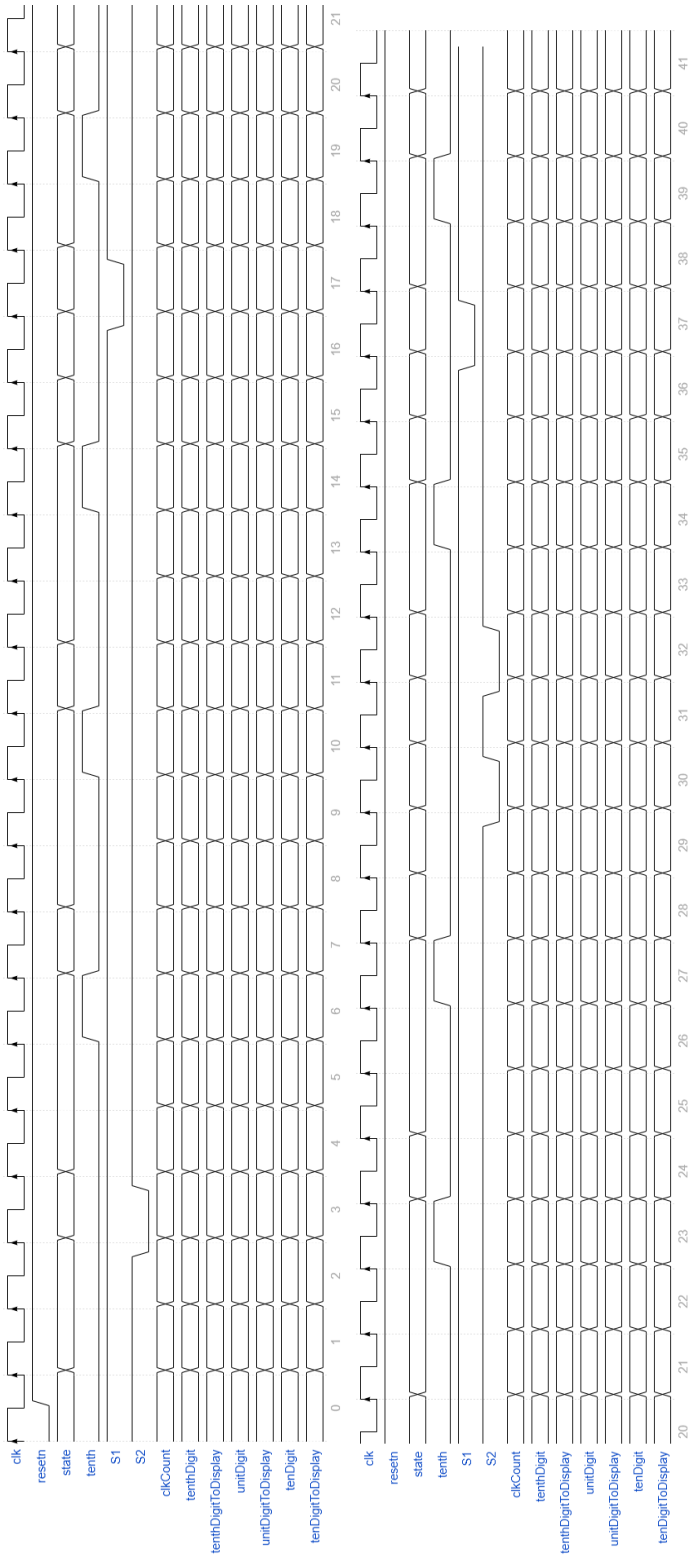


Figure 11.4: Timing diagram that you will run on the testbench.

The goal of the testbench was to cover every transition arc in the state diagram. This goal was not achieved; several transition arc was not taken in the testbench, which one was it? I'd suggest double checking this transition during the testing of the stopwatch when it is downloaded onto the FPGA development board.

Run the testbench using the provided do file and compare the output against Figure 11.4 You will probably need to modify the do file to make it work with your design. Address any problems in the stopwatch before proceeding.

11.5 Stopwatch Synthesis

When you created the datapath, you intentionally designed the timer counter to count up from 0 to 2 in order to expedite execution of the simulations. Before you synthesize the datapath, you need to undo this. I would suggest leaving the relevant constants in your code and just comment them out. Immediately following each commented constant, put the constant that you need for the datapath to operate correctly on the FPGA development board. I've summarized these changes in Listing 11.1.

Listing 11.1: Changes to the datapath that will allow it to run properly on the FPGA development board.

```
// parameter N = 4;
parameter N = 24;

// localparam    tenthSecondConstant    = 4'h000002;
localparam    tenthSecondConstant    = 24'h4c4b40;

//localparam    zero24 = 4'h000000;
localparam    zero24 = 24'h000000;
```

While you are at it, make sure that you remove the testbench as the top-level module and make the stopwatch the top-level module. Then run the analysis and elaboration tool to make sure that the changes in Listing 11.1 did not create any warnings or errors

The next step is to create the mapping of stopwatch module inputs and outputs to the pins of the FPGA and by extension the input and output devices on the FPGA development board. Use the inputs and outputs shown in Figure 11.2 and the information in the FPGA development board User Guide to complete the following pin assignment tables.

Table 11.1: Pin assignment for the stopwatch.

S2	Key[3]	Y16
S1	Key[2]	
resetn	Key[0]	
clk	CLOCK_50	R20

Segment	tenHex Hex2	unitHex Hex1	tenthHex Hex0
seg[6]			
seg[5]			

Segment	tenHex Hex2	unitHex Hex1	tenthHex Hex0
seg[4]	V20		
seg[3]			
seg[2]			V17
seg[1]			
seg[0]		AA18	

After making the pin assignment, download and test your design.

11.6 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.

Stopwatch Simulation

- Completed Figure 11.4. Please paste the images in landscape format into your solutions
- Show the screen shot of simulation timing diagram. Please compose the simulation in landscape format into your solutions and break the screen shot of the simulation into three parts,
  - from 0 to 400ps
  - from 400ps to 800ps
  - from 800ps to 1200ps

Stopwatch Synthesis

- Completed pin assignment from Table 11.1.
- Demo working stopwatch