

November 20, 2024

This document was prepared with L^AT_EX.

The laboratory exercises associated with Digital Design - A Datapath and Control Approach
© 2024 by Christopher Coulston are licensed under CC BY-NC-SA 4.0 For more infomation
about the Create Commons license see: <https://creativecommons.org/licenses/by-nc-sa/4.0/>



Contents

Contents	iv
Introduction to Verilog	ix
Outcomes and Objective	ix
FPGA: Creating a project in Quartus and running a testbench	ix
FPGA: Symbolic to Verilog , Timing Diagram, Truth Table	xii
FPGA: Verilog to Symbolic, Truth Table, Circuit Diagram	xiii
FPGA: Circuit Diagram to Verilog, Symbolic, Truth Table	xiii
Turn in	xiv
Hexadecimal to Seven-Segment Converter	xv
Outcomes and Objectives	xv
Verilog: Vectors	xv
Verilog: Always/Case statements	xvi
A Multiple Output Function	xvii
FPGA: Pin-Assignment	xviii
FPGA: Synthesizing a Verilog Module	xix
Hexadecimal to 7-segment Converter	xx
Testbench	xxii
Pin-Assignment and Synthesis	xxii
Turn in	xxiii
Rock Paper Scissors	xxv
Outcomes and Objectives	xxv
The Rock Paper Scissors Game	xxv
System Architecture	xxvii
Module: onesToDense	xxvii
Module: playToSeven	xxviii
Module: winLose	xxix
Module: rpsGame	xxxi
Testbench	xxxii
Pin-Assignment and Synthesis	xxxii
Turn in	xxxii

High Low Guessing Game	xxxv
Outcomes and Objectives	xxxv
The Guessing Game	xxxv
System Architecture	xxxvi
Module: 2:1 Mux	xxxvi
Module: Compare	xxxvii
Module: hexToSevenSeg	xxxix
Module: 2:4 Decoder	xxxix
Module: hiLowWin	xl
Module: LFSR	xl
Module: hiLow	xlii
Testbench	xliii
Pin-Assignment and Synthesis	xliii
Turn in	xliv
Debugging Tips	xlv
High Low Guessing Game With Hints	xlvii
Outcomes and Objectives	xlvii
The Guessing Game with Hints	xlvii
System Architecture	xlix
Module: 2:1 Mux	1
Module: Compare	1
Module: Add/Sub	1
Logic: hotWarmCold	1
Module: hiLow	liii
Testbench	liii
Pin-Assignment and Synthesis	liv
Turn in	lv
Debugging Tips	lvi
Calculator With Friendly Output	lix
Outcomes and Objectives	lix
Calculator with Friendly Output	lix
System Architecture	lx
Module: sigUnsign	lxii
Testbench	lxvi
Pin-Assignment and Synthesis	lxvii
Turn in	lxviii
Bonus: Ovf Logic	lxix
Cellular Automata	lxxi
Outcomes and Objectives	lxxi
1-dimensional cellular automata	lxxi
System Architecture	lxxv
Module: singleCell	lxxvi
Testbench	lxxvii
Pin-Assignment and Synthesis	lxxviii
Turn in	lxxviii
Debugging Tips	lxxix

Mod 10 Counter	lxxxii
Outcomes and Objectives	lxxxii
Module: <code>mod10Counter</code>	lxxxii
System Architecture	lxxxii
Testbench	lxxxiv
FPGA: Do file	lxxxvi
Turn in	lxxxvii
Stopwatch Datapath	lxxxix
Outcomes and Objectives	lxxxix
Stopwatch	lxxxix
Module: <code>stopWatchDatapath</code>	xc
Turn in	xcviii
Stopwatch Control Unit	xcix
Outcomes and Objectives	xcix
Stopwatch	xcix
Module: <code>stopWatchControl</code>	c
Timing Diagram	civ
Turn in	cvii
Stopwatch Datapath and Control	cix
Outcomes and Objectives	cix
Stopwatch	cix
System Architecture	cx
Turn in	cxiv

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

Objectives

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

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

- Arithmetic Statements
- Conditional Statements
- Basic Memory Element Behavior
 - Characteristics
 - Timing of basic memory element
 - Asynchronous set reset
- Sequential Logic Building Blocks
 - Analysis, design and use of a register in a digital design
 - Shift Register
 - Counter
 - Static RAM
 - Design a circuit that performs register transfer
- Finite State Machines
 - Hardware organization of a finite state machine
 - State diagram for a finite state machine
 - One's Hot Encoding
 - Design
 - Using a timing diagram to specify or verify the proper operation of a Finite State Machine
- Datapath and Control
 - Datapath and Control Architecture
 - Algorithmic Language
 - Design a control word table and specify the control word values for every state.
 - Design
 - Using a timing for a datapath and control circuit to specify or verify proper operation.

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

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

Introduction to Verilog

Outcomes and Objective

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

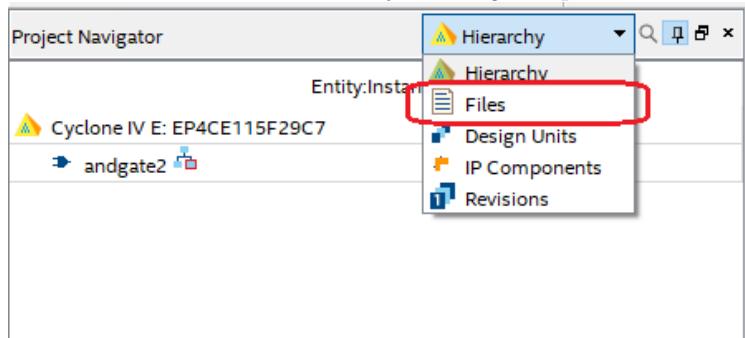
- Elementary Logical Functions
- Conversion between two different representations of a logic function
- Writing concurrent signal assignment statements for a logic function
- Writing a Verilog statement using primitive logic operations
- Creating a simulation timing diagram for a module

FPGA: Creating a project in Quartus and running a testbench

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

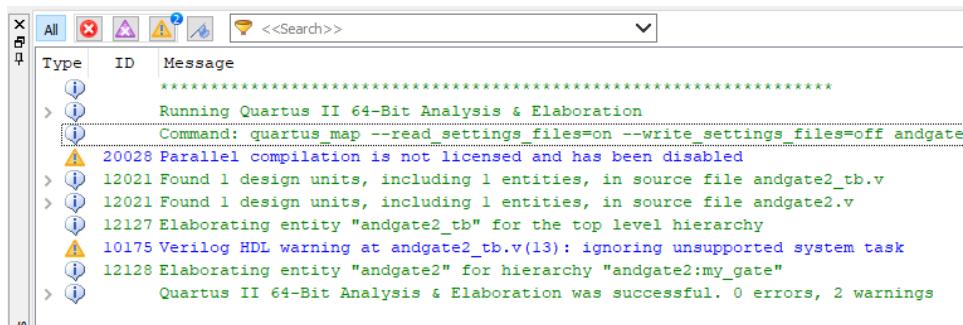
1. Select an appropriate working directory for your project. I would recommend selecting your network drive.
 - a. Create a new folder *lab1*,
 - b. Create another folder within *lab1* called *andgate2*,
 - c. Download *andgate2.v* and *andgate2.tb.v* from Canvas,
 - d. Save these files in *andgate2*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

5. In the **Project Type** page of the New Project Wizard pop-up:
 - a. Select the *Empty project* radio button,
 - b. click *Next*.
6. In the **Add Files** page of the New Project Wizard pop-up:
 - a. Click the ... button to the right of File name,
 - b. In the Select File pop-up, navigate to, and select, *andgate2.v* and *andgate2_tb.v*, click Open,
 - c. The file should appear in the window below,
 - d. Click *Next*
7. In the **Family & Device Settings** page of the New Project Wizard pop-up:
 - a. Device family, Family: Cyclone V
 - b. Package: FBGA
 - c. Pin Count: 672
 - d. Speed Grade: 7_H6
 - e. Select Specific device selected in ‘Available devices’ list
 - f. From the list of available devices, select: 5CGXFC5C6F27C7
 - g. Click *Next*
8. In the **EDA Tool Settings** page of the New Project Wizard pop-up:
 - a. In the Simulation row
 - i. Tool Name column: ModelSim-Altera
 - ii. Formats column: Verilog HDL
 - b. Leave other defaults alone
 - c. Click *Next*
9. In the **Summary** page of the New Project Wizard pop-up:
 - a. Review information,
 - b. Click *Finish*.
10. Back in the main Quartus II window, Click *Tools -> Options...*
11. In the Options pop-up:
 - a. Select *EDA Tool Options* from the Category menu,
 - b. If the last row, “ModelSim-Altera” is blank, click on the ... button at right and navigate to the *C:\intelFPGA_lite\18.1\modelsim_ase*, select the *win32aloem* folder, the click Select Folder. Note the software version in these instructions is 18.1 The version installed on your computer may be different. If so, the path should be the same with the exception of the version number.
 - c. Click *Ok*.
12. Click on the Files tab in the *Project Navigator* pane.

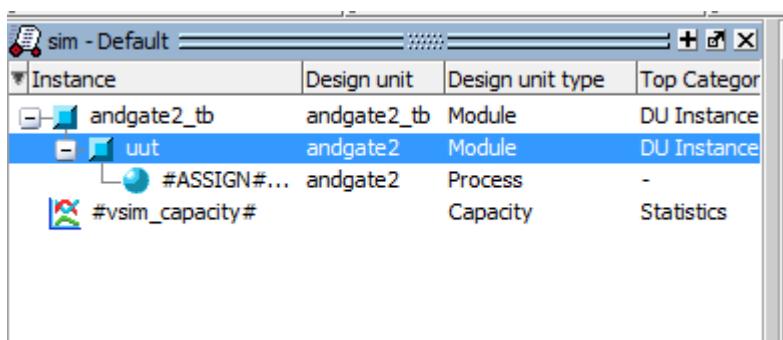


13. Right click on *andgate2_tb* in the *Project Navigator* pane and select Set as Top-Level entity.

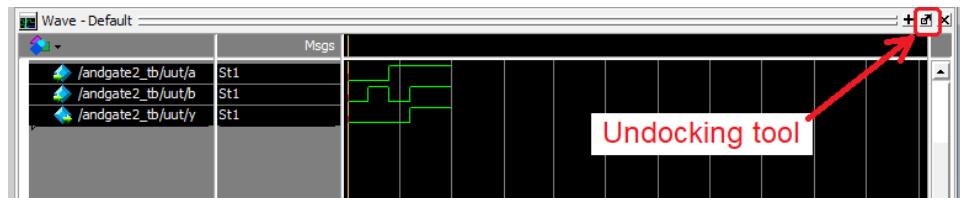
14. Double click on *andgate2*.
15. If you added the Verilog file in the correct directory and included it in the project, a Verilog file should pop up on the right.
16. In the main Quartus II window, click on *Processing -> Start -> Start Analysis & Elaboration*. This may take some time, so be patient.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



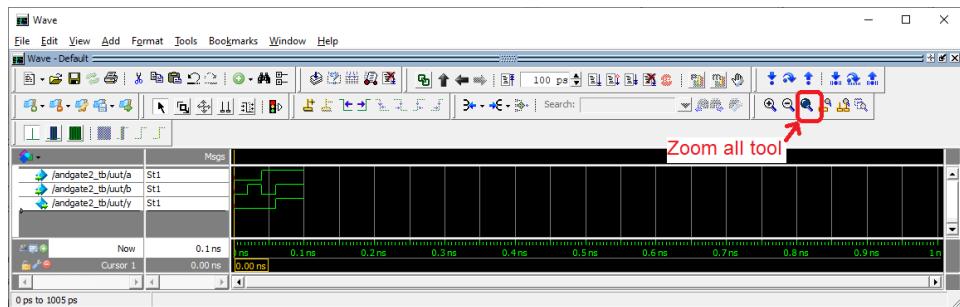
18. In the main Quartus II window, click *Tools -> Run Simulation Tool -> RTL Simulation*. The ModelSim program will launch. This may take a few moments, be patient. If you get a pop-up Nativelink Error window, then go back, check and fix the directory in step 11.
19. In ModelSim, click *Simulate ->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



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



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



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

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

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

FPGA: Symbolic to Verilog , Timing Diagram, Truth Table

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

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

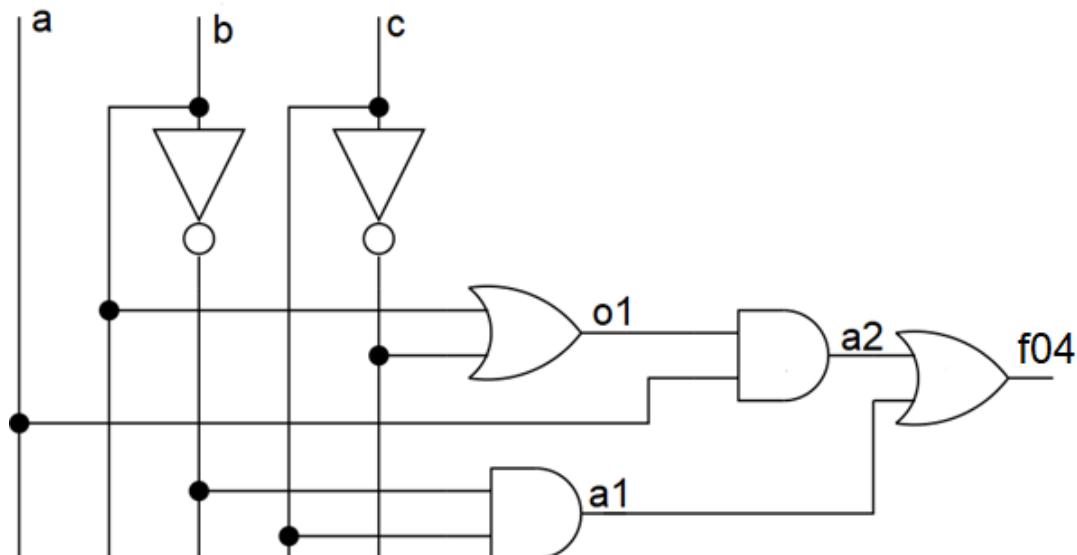
FPGA: Verilog to Symbolic, Truth Table, Circuit Diagram

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

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

FPGA: Circuit Diagram to Verilog, Symbolic, Truth Table

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



1. Create a new project folder within your *lab1* directory called *function04*.

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

Turn in

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

FPGA: Creating a project in Quartus and running a testbench

- Step 23 Timing diagram of AND gate

FPGA: Symbolic to Verilog , Timing Diagram, Truth Table

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

FPGA: Verilog to Symbolic, Truth Table, Circuit Diagram

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

FPGA: Circuit Diagram to Verilog, Symbolic, Truth Table

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

Hexadecimal to Seven-Segment Converter

Outcomes and Objectives

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

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

Verilog: Vectors

A vector is a collection of bits that are related to one another in some way. For example, the individual bits of a 4-bit number could be represented as a vector. There are three things that you will need to know about vectors in order to complete today's lab (and future labs), combining bits into a vector, defining a vector, and accessing the bits of a vector. These operations are illustrated in Figure 1.

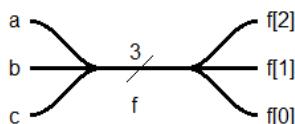


Figure 1: An illustration showing three bits combined into a vector, f , and then accessing the individual bits of f .

The operations shown in Figure 1 are similar to the operations in Listing 1. In this module, the line of code `f = {a,b,c};` combines the individual signals `a`, `b` and `c` into a 3-bit vector `f`. The left-most signal in the parenthesis list, `a`, becomes the MSB of the vector and the right-most signal, `c`, becomes LSB. Combining signals is more commonly called concatenation.

You can concatenate any arrangement of signals as long as the number of bits comes out the same as the signal on the left-hand-side of the = sign.

Listing 1: Verilog code which illustrates vector manipulations and declarations.

```
module unimportantModuleName();

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

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

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

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

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

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

Verilog: Always/Case statements

We will use the Verilog *always* statement to implement a function using its truth table. Listing 2 shows an always statement that uses the 3-bit value of a signal x to compute the value of f.

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

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

always @(*)
    case (x)
        3'b000: f = 3'b00;
        3'b001: f = 3'b01;
        3'b010: f = 3'b01;
        3'b011: f = 3'b10;
        3'b100: f = 3'b01;
        3'b101: f = 3'b10;
        3'b110: f = 3'b10;
        3'b111: f = 3'b11;
    endcase
```

For the time being, we will trust that the statement **always** @(*) allows the code between **case** and **endcase** to run continuously and concurrent with any other statements in the module.

Yes, this means that all the code between `case` and `endcase` acts like a single assign statement. A case statement uses the argument to case, `x` as a selector for one of the rows below. Every possible value of `x` must be present and when that value matches `x`, the action to the right of the colon is performed. When we use a case statement as shown in Listing 2 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`.

A Multiple Output Function

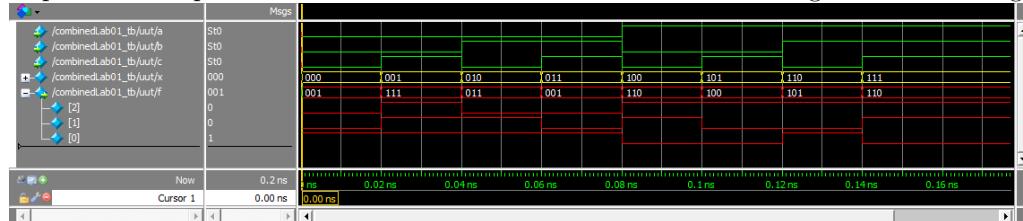
Let's explore vectors and the always statement by combining the three functions created in last weeks assignment into one function.

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

Table 1: The Truth Table for the combinedLab01 function. This function has a 3-bit input and 3-bits output.

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

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



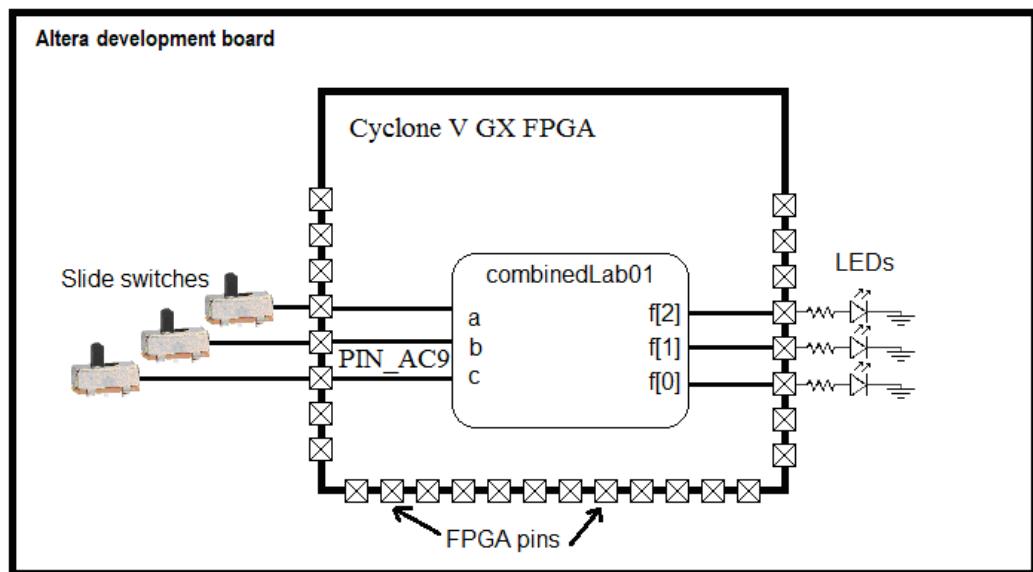
8. From the information in the timing diagram, produce a truth table. Compare the truth table generated from the data in the timing diagram to that you generated in Table 1.

FPGA: Pin-Assignment

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

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

Figure 2: A simple Verilog design synthesized and downloaded onto the development board.



The development board contains an Altera Cyclone V GX FPGA. This FPGA has many pins and they are identified by a lettered group and number. For example, in Figure 2 port c of the combinedLab01 module is mapped to pin AC9.

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

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

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

Use the information to complete the pin assignment in Table 2. We will use this assignment in the next section.

Table 2: Pin Assignment Table for combinedLab01.

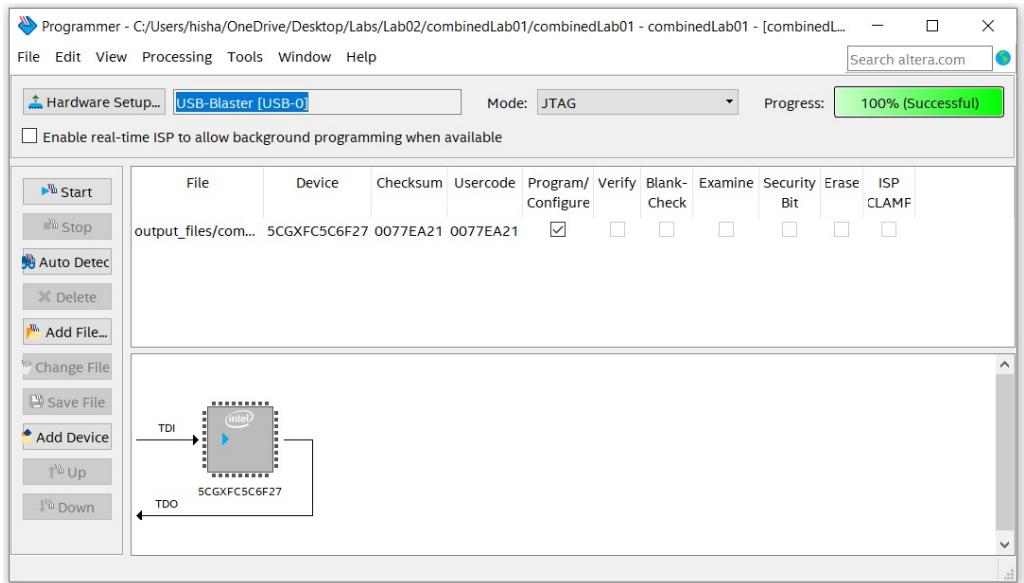
Port	a	b	c	f[2]	f[1]	f[0]
Signal name	SW[2]	SW[1]	SW[0]	LEDR[2]	LEDR[1]	LEDR[0]
FPGA Pin No.			PIN_AC9			

FPGA: Synthesizing a Verilog Module

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

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

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservation
a	Input				2.5 V (default)	8mA (default)				
b	Input				2.5 V (default)	8mA (default)				
c	Input				2.5 V (default)	8mA (default)				
f[2]	Output				2.5 V (default)	8mA (default)	2 (default)			
f[1]	Output				2.5 V (default)	8mA (default)	2 (default)			
f[0]	Output				2.5 V (default)	8mA (default)	2 (default)			
6. Double click in the Location cell for row c
7. Scroll down the list of pins to PIN_AC9
8. Complete the pin assignment for the other 5 inputs and outputs using the information contained in pin assignment table completed earlier.
9. Double check your pin assignments.
10. File -> Close. Note closing your file incorporates this assignment into the project.
11. Back in the Quartus window, Processing -> Start Compilation <Ctrl-L>
12. Tools -> Programmer
13. In the Programmer pop-up window click Add File...
14. In the Select Programming File pop-up, navigate to your project directory, then into the output files folder, the select combinedLab01.sof, click Open. You should see something like the following.



15. Connect the Altera Cyclone V GX FPGA to your computer through the USB port, connect the power supply, and push the red power-on button. Try not to be annoyed by the infernal blinking LEDs.
16. In the Programmer pop-up
 - a. Click Hardware Setup....
 - b. In the Hardware Setup select USB-Blaster [USB=0] from the Currently selected hardware pull-down
 - c. Click Close
17. Back in the Programmer window, the box next to Hardware Setup... should reflect your choice. Click Start,
18. The Development board should stop its infernal blinking and run your program. You may notice that the unused LEDs are dimly illuminated.
19. Move the slide switches up and down to verify that the input/output matches the values in Table 1. Use white silk screen printing on the development board to locate slide switches and LEDs you assigned in your pin-assignment.

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.

The pattern of segments to be illuminated for each digit is shown in Figure 3. For example, to display '4' output would be:

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

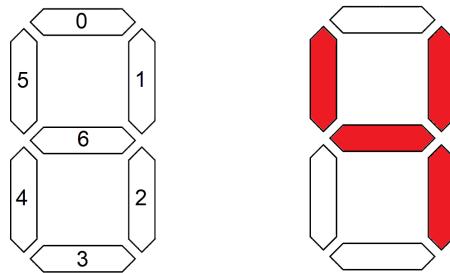


Figure 3: Left, the proper numbering of the segments. Right, illuminating segments to form the number 4.

Figure 4 shows the proper formatting for all the values between 0 – f.

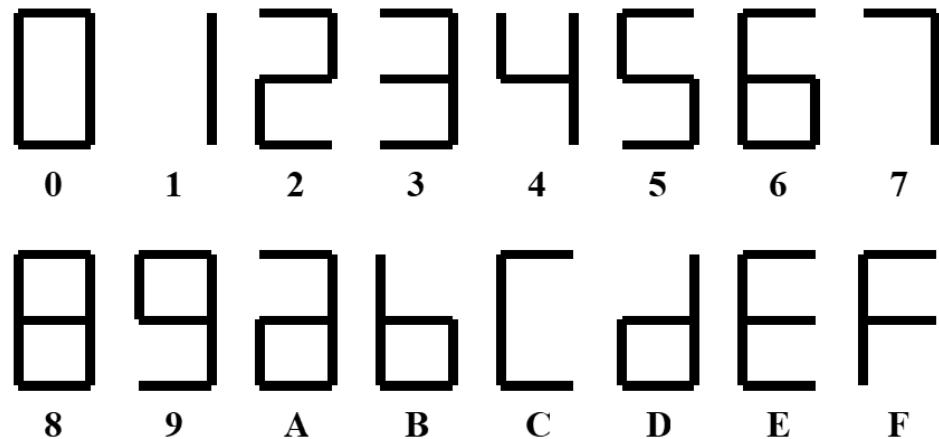


Figure 4: The proper arrangement of LEDs to form hexadecimal characters.

Use this information to complete the Table 3 to illuminate the active low led segments to generate proper hexadecimal characters.

Table 3: Truth table for the hexToSevenSeg component.

x	seg[6]	seg[5]	seg[4]	seg[3]	seg[2]	seg[1]	seg[0]
0000							
0001							
0010							
0011							
0100	0	0	1	1	0	0	1
0101							
0110							

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

Now that you have a complete description of the input/output behavior of the hexadecimal to seven segment converter, it's time to write the Verilog code. You will capture the behavior in Table 3 using an always/case statement. Work through the following steps to complete this task.

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

Testbench

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

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

Pin-Assignment and Synthesis

Before you can download your design to the FPGA, you need to assign the input and outputs of the *hexToSevenSeg* module to FPGA pins. Figure 5 shows the slide switches and 7-segment display that will use.

Use the C5G User manual to complete the pin-assignment in Table 4.

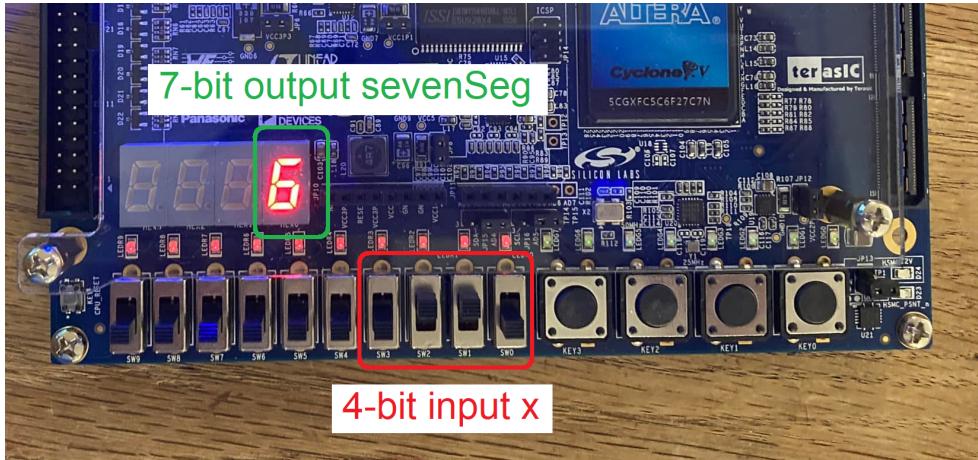


Figure 5: The input set to 4'b0110 displaying a 6 on the 7-segment display.

Table 4: Pin-assignment tables for the hexToSevenSeg module.

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

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

Use the instructions in Section 1 to combine the pin assignment with your hexToSevenSeg module. Synthesize your design, bask in the glow of another success as you demonstrate your circuit's functionality to a member of the lab team.

Turn in

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

Combine lab 1

- Truth Table for combinedLab01 function in Table 1
- Link Timing diagram for combinedLab01 function
- Pin assignment for combinedLab01 in Table 2

Hexadecimal to 7-segment

- Truth Table for hexToSevenSeg function in Table 3
- [Link](#) Verilog code for hexToSevenSeg function – just the always/case statement
- [Link](#) Timing diagram for hexToSevenSeg function
- Pin assignment tables for hexToSevenSeg in Tables 4
- Demonstrate working hexadecimal to seven segment module to a member of the lab team.

Rock Paper Scissors

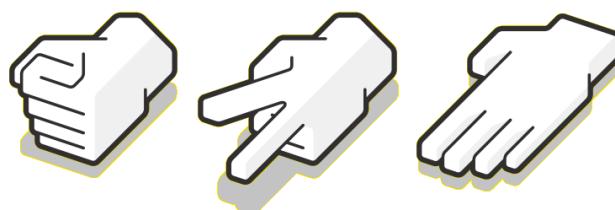
Outcomes and Objectives

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

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

The Rock Paper Scissors Game

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



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. Your goal in this lab is to create a digital version of rock, paper, scissors on the Altera Cyclone V Board using the inputs and outputs shown in Figure 6. 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).

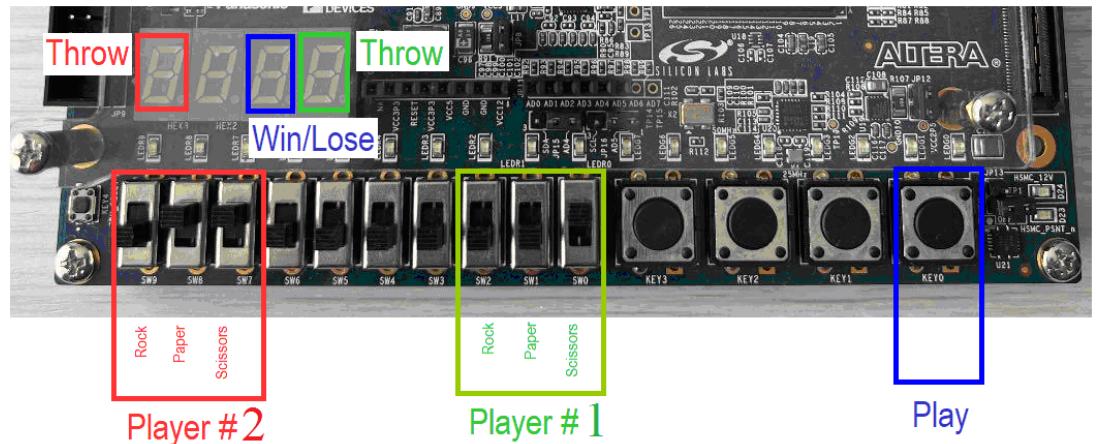


Figure 6: The input and output you should use to realize your digital system.

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 7. These patterns are supposed to vaguely resemble the objects.

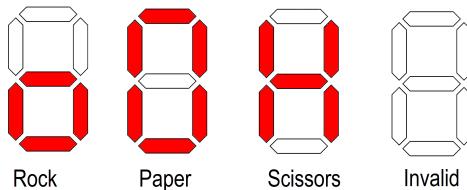


Figure 7: Illuminated patters for the different plays.

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 6. When the Throw button is un-pressed, the Win/Lose 7-segment display is blank.

Table 6: 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 Architecture

The system architecture shown in Figure 8 is your guide to building a functioning circuit. As such, we will take a moment to cover some important details of this diagram that will help you write your Verilog code later./

The names outside the FPGA square correspond to the labels in Figure 6. Each soft-square (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.

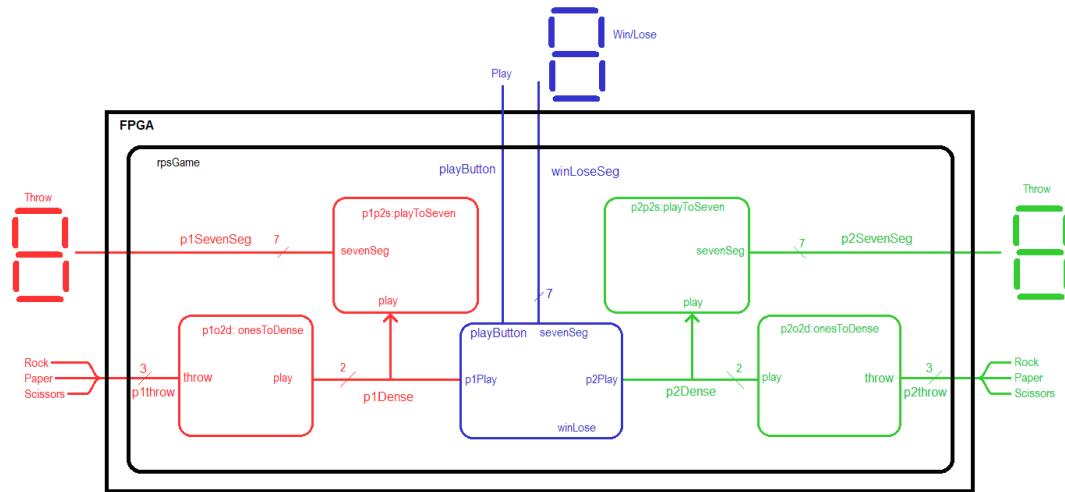


Figure 8: System architecture for the rock, paper, scissors system.

Module: onesToDense

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

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

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

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

In order to write the Verilog code for this module, complete the truth table in Table 7 for the onesToDense module.

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

Table 7: The truth table for the onesToDense module.

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

From this truth table, determine the canonical SOP expressions for play[1] and play[0] functions. Do this by writing the canonical SOP expression for the most significant bit of the play output, play[1], in the Table 7 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 7 truth table while ignoring the MSB.

play[1] =

play[0] =

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

- Use the module declaration: module onesToDense (throw, play);
- Use vectors for the throw input. The MSB should come from the rock slide-switch and the LSB from the scissors slide-switch.
- Use a vector for the play output.
- Make the input and output port types “wire”.
- You may want to break the input vector into its component pieces to correspond to the variable names used in Table 7. 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.

Module: playToSeven

The playToSeven module converts the player throw, represented in the dense coding, to a “graphical” form displayed on the 7-segment display. The symbols for each possible throw are shown in Figure 9.

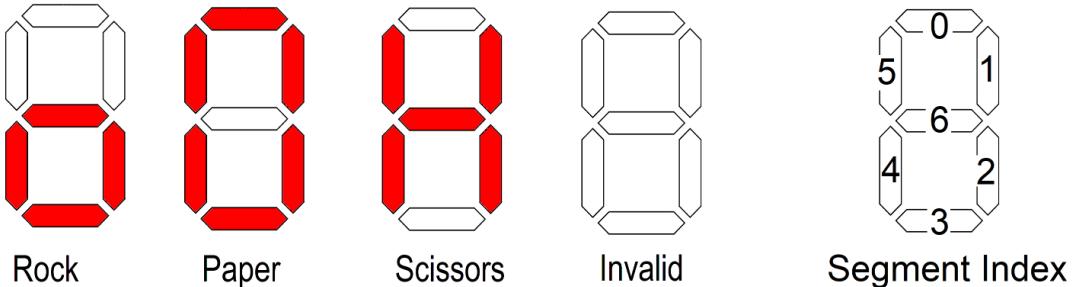


Figure 9: 7-segment display patterns for the different throws along with the 7-segment display segment indices.

Use the information in Figure 9 to determine the bit patterns needed to generate the four different symbols in Table 8. Remember that the LEDs in the segments are active low, meaning that a logic 0 output illuminates an LED segment. In Table 8 put a 0 or 1 in each of the numbered column so that each row produces the patterns for its throw. Remember that pPlay codes rock as 00, paper as 01, scissors as 10 and an invalid throw is coded as 11. In the sevenSeg column put the 7-bit code formed by concatenating the bits together. Use proper Verilog syntax to write this 7-bit vector.

Table 8: 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 8 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.

Module: winLose

The winLose module takes the throw from each player (in the dense coding), the push button, and determines what to display on the win/lose 7-segment display. The win/lose 7-segment display is blank when the button is not being pressed, otherwise it will show “1” when player 1 has a winning throw, “2” when player two has the winning throw, “d” when the players

have the same throw. The patterns are the same as those you have already created for the hexToSeven module and are shown in Figure 10 as a reminder.

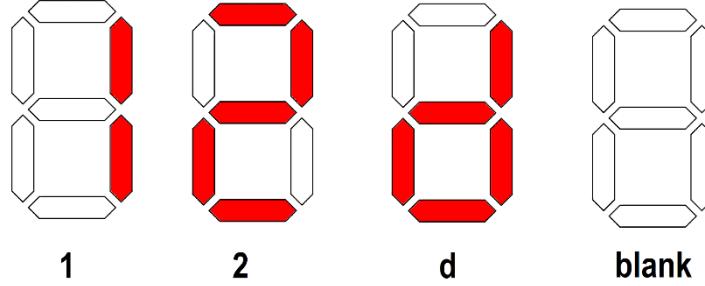


Figure 10: The illuminated patterns displayed on the win/lose 7-segment display.

You will use a case statement, based on the information in Table 9, 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 9.

Table 9: 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				
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 8.

- Use the module declaration: `module winLose(p1Play, p2Play, playButton, sevenSeg);`
- Use vectors for the p1Play and p2Play inputs.
- Use a vector for the sevenSeg output.
- Make the input port types “wire”. Make the output port types “reg”.

- You will use a case statement, embedded in an always statement to realize this module.
 - Use brackets to make the vector for the case statement. For example, if button was a 1-bit signal and play was a 2-bit signal, then

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

This code snippet will use the 3-bit value (button as MSB and play as least significant 2-bits) to select one of the rows. Note that the default case handles all the combinations where button is 1.

- Use a default case to handle all the situations where the button is not pressed. The default case catches any unspecified input combinations for the case statement. List the default as the last row in the case list.
- At the end of each “case” row, provide a comment that lists player 1’s throw, player 2’s throw and the output that is displayed on the 7-segment display. For example, in my program the first case row has a comment that looks like:
`// P1: Rock P2:Rock Draw`
- Use the hex2Seven module from lab 2 as the starting point for this module.

Module: rpsGame

The rpsGame module, “glues” together the modules shown in Figure 8 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 8 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 8.
- 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);`

Testbench

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

Pin-Assignment and Synthesis

Use the image of the FPGA Development Board in Figure 6 and the information in the C5G User Guide to determine the FPGA pins associated with the input and output devices used by the rpsGame module.

Table 10: 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.

Complete the pin-assignment in Quartus, compile your design and download to the FGPA development boards. Once you get your design working, demonstrate it to a member of the lab team.

Turn in

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

onesToDense Module

- Complete Table 7 truth table for oneToDense module.
- [link](#) Canonical SOP expressions for the play[1] and play[0] functions.
- [link](#) Verilog code for the entire module (courier 8-point font single spaced), leave out header comments.

playToSeven Module

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

winLose Module

- Complete Table 9 truth table for winLose module.
- [link](#) Verilog code for the module (courier 8-point font single spaced), leave out header comments.

rpsGame Module

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

Pin Assignment

- Completed pin assignment for 7-segment, slide switches and button in Table 10.
- Demonstrate your working design to a lab team member.

High Low Guessing Game

Outcomes and Objectives

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

- Analyzing a circuit with a combination of building blocks
- Designing glue logic to interface building blocks
- Definition and instantiation of Verilog generic modules
- Definition of Verilog modules

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 [0-15], inclusive. Once the *secret number* is decided, the guesser makes a *guess*, a number in the interval [0-15] 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*.

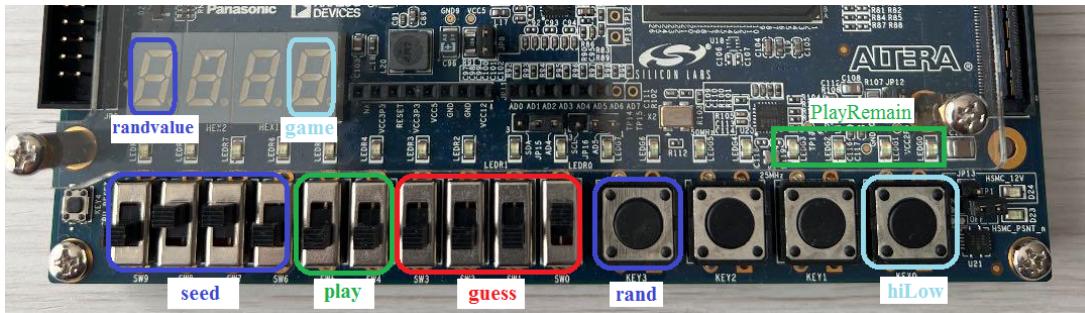


Figure 11: The input and output you should use to realize the High/Low Guessing game.

Your goal in this lab is to create a digital version of the guessing game using the development

board using the inputs and outputs shown in Figure 11. 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* \neq *secret number*
- ‘I’ when *guess* = *secret number*
- ‘L’ when *guess* \neq *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 Architecture

Use the system architecture shown in Figure 12 as your guide to this design. Please note that lines with the same name in different places are connected together. For example, the signal **randBut** connects the button input to the 2:1 mux in the upper left corner of the FPGA.

Module: 2:1 Mux

A 2:1 multiplexer, a mux for short, is a basic building block in many digital systems. The 2:1 mux shown in Figure 13 routes one of the two N-bit data inputs, *y*₀ or *y*₁ to the N-bit output, *F*, depending on the value of a 1-bit select signal, *s*. When *s* = 0, *F* = *y*₀ and when *s* = 1, *F* = *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 12 have their *y*₁ or *y*₀ data inputs denoted as 1 and 0 respectively. This is done to save space and increase clarity in the schematic.

The Verilog code for a 2:1 mux is provided to you on Canvas. When creating instances of the 2:1 mux, you will need to correctly order the signals in the module instantiation. To do this, follow the order shown in the module declaration shown in the top two lines in Listing 3.

Listing 3: Top, module definition for a 2:1 mux. Bottom, module instantiation of a 2:1 mux in Figure 12.

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

// Module instantiation for a 2:1 mux in the hiLow digital circuit
genericMux2x1 #(7) muxHex(7'b1111111, RandHex, randBut, randSeg);
```

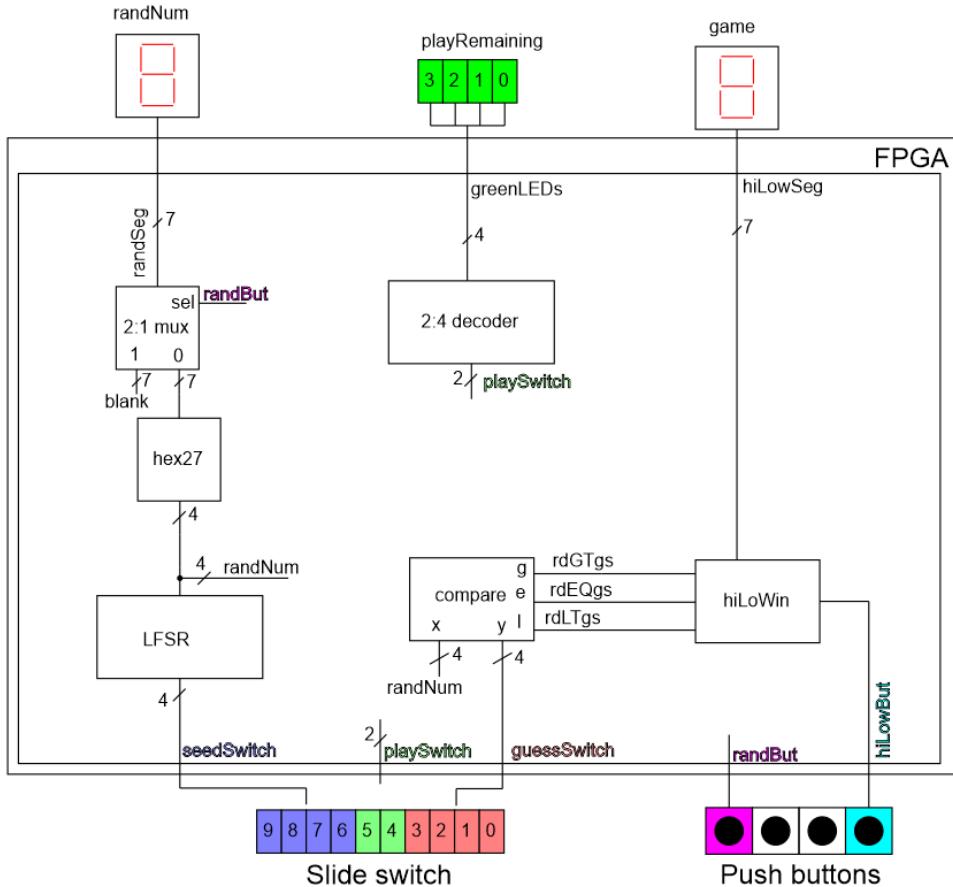


Figure 12: System architecture for the guessing game.

The signal width, N , shown in Figure 13 is a placeholder for an integer value that describes the width of the y_1 , y_0 , F signals. You can specify this width when you instantiate a `genericMux2x1` module using the `#()` specifier immediately after `genericMux2x1` as shown in the bottom line of code in Listing 1.

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

Module: Compare

A N -bit comparator is a basic building block in many digital systems. The N -bit comparator shown in Figure 15 checks the relative magnitude of the two N -bit inputs x and y and sets one

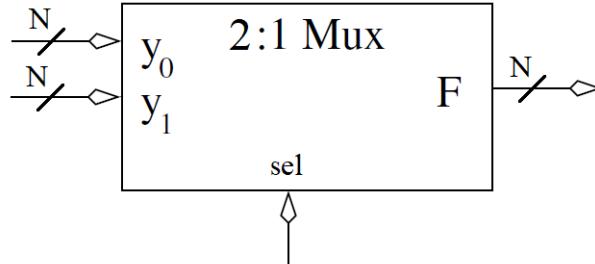


Figure 13: The schematic representation of a 2:1 mux.

Port Connectivity Checks: "genericMux2x1:muxMsbHex"				
	Port	Type	Severity	Details
1	y1	Input	Info	Stuck at VCC
2	f	Output	Warning	Output or bidir port (8 bits) is wider than the port expression (7 bits) it drives; bit(s) "[7..7]" have no fanouts
3	y1	Input	Warning	[Input port expression (7 bits)] is smaller than the input port (8 bits) it drives. Extra input bit(s) "y1[7..7]" will be connected to GND.
4	y0	Input	Warning	[Input port expression (7 bits)] is smaller than the input port (8 bits) it drives. Extra input bit(s) "y0[7..7]" will be connected to GND.

Figure 14: Forgetting the generic specifier in a 2:1 Mux will generate this report.

of the three outputs equal to 1, one's-hot output, depending on their relation to each other.

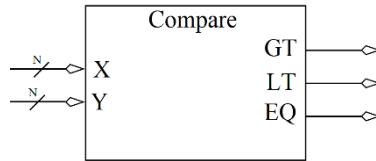


Figure 15: A schematic representation of a N-bit comparator.

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

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

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

Listing 4: Top, the module definition for the comparator. Bottom, module instantiation of a comparator in Figure 15. Remove the component instantiation line break in your code.

```
// Module definition for the comparator
module genericComparator(x, y, gt, eq, lt);

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

```
randGTguess , randEQguess , randLTguess );
```

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

Module: hexToSevenSeg

You should use the hexToSevenSeg module you developed in a previous lab. Note, the name of this module was shortened in Figure 12 to hex27 in order to save space and make the schematic more readable.

Module: 2:4 Decoder

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

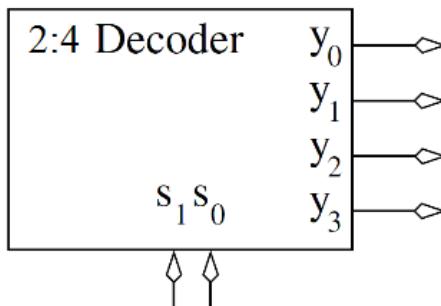


Figure 16: A 2:4 decoder.

The first few rows for the truth table for the 2:4 decoder are shown in Table 13.

Table 13: Partial truth table for the 2:4 decoder.

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

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

You should implement the 2:4 decoder in the hiLow module using an always/case statement similar to the one used to implement your hexToSevenSeg. You should put this Verilog code in the hiLow module as a (large) concurrent statement. **This means that you should not have a separate Verilog file for the 2:4 decoder.** Remember that the output type from an always/case statement must have the "reg" qualifier, not "wire".

Module: hiLowWin

The hiLowWin functionality converts the output from the comparator into the illuminated patterns shown in Figure 17 when the **hiLow** button is pressed. The “I” from “wIn” is needed because you cannot make a “W” on a 7-segment display.

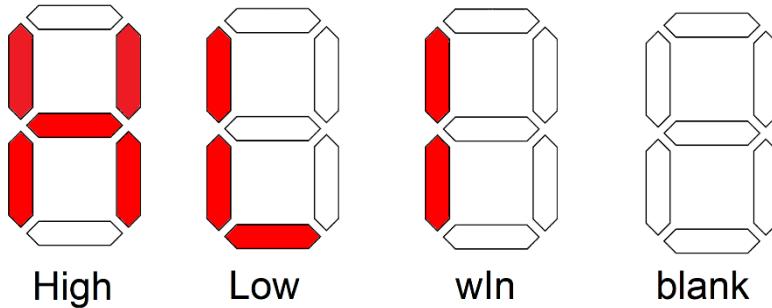


Figure 17: The illuminated patterns to inform the guesser about the magnitude of their guess.

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 5. Note that the code shown Listing 5 is incomplete.

Listing 5: Starter code for the hiLowWin module.

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

Module: LFSR

A linear feedback shift register (LFSR) is a digital circuit that generates a pseudo-random sequence of numbers starting from a seed value. Since we do not yet have storage devices in our class, we will implement a LFSR that performs a single iteration of the randomization step as shown in Figure 18.

Figure 18 shows the input bits $I_2 \dots I_0$ being shifted one bit to the left on their way to the outputs $O_3 \dots O_1$. The output O_0 is formed by computing $I_2 \wedge I_0$ where “ \wedge ” is the xor operation.

Let’s use Table 14 to understand what happens if the input in Figure 18 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 $1 \wedge 0$ which equals 1. The resulting output is 4'b1101, when interpreted as a decimal number,

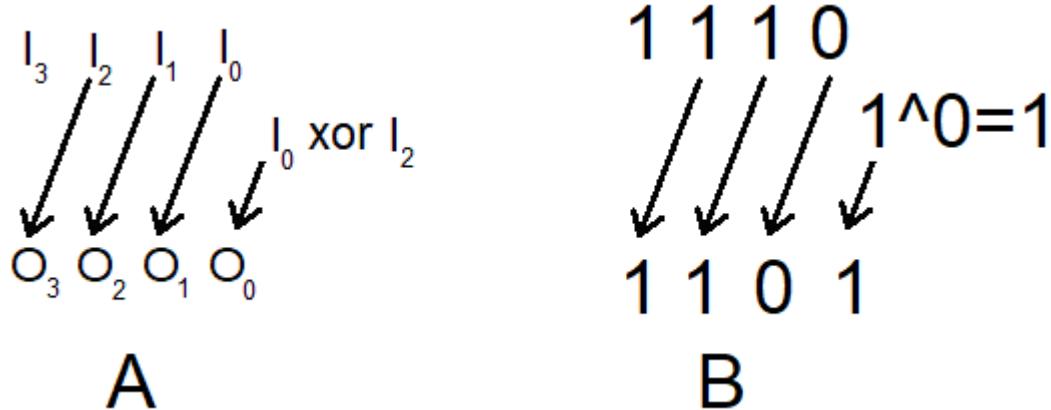


Figure 18: A) A schematic illustration of a 4-bit LFSR operation. B) The input 4b'1110 produced the output 4b'1101.

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 14: The first iteration of the LFSR shown in Figure 18 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 14 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 18 is 15. This means that if Table 2 had 15 rows and you filled them all in, you would get 14 on the 14th row. Can you figure out what number is excluded from the sequence?

For the lfsr module, you need to:

- Use the module declaration:

```
module lfsr(Seed, outputRand);
```
- Make the input and outputs vectors with wire type.
- Use 4 assign statements to give each bit of output a value.
- Complete the testbench for the lfsr module. Create timing diagram that asserts the four inputs listed in Table 14 waiting #20 between inputs. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Switch radix to unsigned decimal for input and output (right click on signal name in wave pane and select radix -> unsigned).

Module: hiLow

The hiLow module stitches together all the modules and contains all the signals shown in Figure 12. The module declaration is provided below to assist your pin assignment.

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

To complete this module, you will need to instantiate all the modules in Figure 12. To provide guidance on this process let's focus on the 2:1 mux from Figure 19. This 2:1 mux is reproduced in Figure 19.

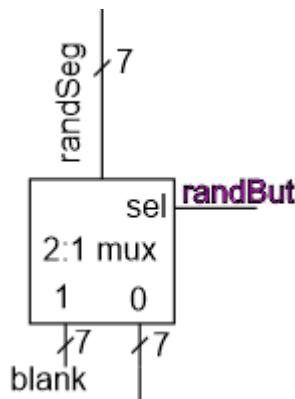


Figure 19: A small piece of hardware from Figure 12.

Let's write the Verilog code to instantiate the 2:1 mux. The first step that you need to take is to give EVERY signal in the system architecture a name or constant value. With respect to Figure 19, the output of the 2:1 mux is already named **randSeg** and the select line is named **randBut**. The data input **y1** will have a constant value **7b'1111111**, needed to produce a blank 7-segment display. The input **y0** is the output from a hexToSevenSeg module, a signal named **RandHex**.

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

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

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

- Define the width of the data input and data output of the mux (7-bits),
- Give the 2:1 mux instance a descriptive and unique name. For example, **muxHex**,
- Put the system architecture signals in their corresponding locations in the module
`genericMux2x1 #(7) muxHex(7\'textquoteshingle b1111111, RandHex, randBut, randSeg)`

Once you get the hang of it, you are just translating the system architecture of Figure 12 into words.

For the hiLow module, you should:

- Use the module declaration:

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

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

Testbench

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

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

Pin-Assignment and Synthesis

Use the image of the development board in Figure 11 and the information in the board User Guide to determine the FPGA pins associated with the input and output devices used by the hiLow module.

Table 15: Pin-Assignment for the High Low Guessing Game.

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

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

	seedSwitch	playSwitch	guessSwitch
slide[1]			
slide[0]			

randBut	Key[3]		
hiLowBut	Key[0]		

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

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

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

Turn in

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

Module: LFSR

- [Link](#) Verilog code for the body of the module (courier 8-point font single spaced), leave out header comments.
- A completed Table [14](#).
- [Link](#) Complete the testbench for the lfsr module. Create timing diagram that asserts the four inputs listed in Table [14](#) waiting #20 between inputs. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Switch radix to unsigned decimal for input and output (right click on signal name in wave pane and select radix -> unsigned).

Module: hiLow

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

Pin-Assignment and synthesis

- Completed pin assignment table for all the signals in Table [15](#).
- Demonstrate your completed circuit to a lab team member.

Debugging Tips

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

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

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

The screenshot shows the Quartus II Analysis & Elaboration window. The code editor displays a portion of a Verilog module:

```

    case (playSwitch)
      3'b000: greenLEDs = 8'b11111111;
      3'b001: greenLEDs = 8'b01111111;

```

The status bar at the bottom right indicates: "Quartus II 64-Bit Analysis & Elaboration was unsuccessful. 8 errors, 2 warnings".

The message list shows eight error messages of type 10137 Verilog HDL Procedural Assignment error, each pointing to line 65 of the file hiLow.v. The errors state that the object "greenLEDs" on the left-hand side of assignment must have a variable data type.

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

The screenshot shows the Quartus II Analysis & Elaboration window. The code editor displays a portion of a Verilog module:

```

//wire [6:0] randNum;
wire [6:0] warmThreshold, coldThreshold;
wire [6:0] msbRandHex, lsbRandHex;

```

The status bar at the bottom right indicates: "Quartus II 64-Bit Analysis & Elaboration was unsuccessful. 8 errors, 2 warnings".

The message list shows three warning messages of type 20028 Parallel compilation is not licensed and has been disabled, 10275 Verilog HDL Module Instantiation warning, and 10236 Verilog HDL Implicit Net warning.

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

The screenshot shows the Quartus II Analysis & Elaboration window. The code editor displays a portion of a Verilog module:

```


```

The status bar at the bottom right indicates: "Quartus II 64-Bit Analysis & Synthesis was unsuccessful. 1 error, 3 warnings".

The message list shows three error messages of type 12061 Can't synthesize current design -- Top partition does not contain any logic, 293001 Quartus II Full Compilation was unsuccessful, and 293001 Quartus II Full Compilation was unsuccessful.

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

Type	ID	Message
⚠	20028	Parallel compilation is not licensed and has been disabled
⚠	10275	Verilog HDL Module Instantiation warning at hiLow.v(94): ignored dangling comma in List of Port Connections
⚠	12241	1 hierarchies have connectivity warnings - see the Connectivity Checks report folder
> ⚠	13024	Output pins are stuck at VCC or GND
⚠	20028	Parallel compilation is not licensed and has been disabled
⚠	292013	Feature LogicLock is only available with a valid subscription license. You can purchase a software subscription to
⚠	15714	Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for details
⚠	332012	Synopsys Design Constraints File file not found: 'hiLow.sdc'. A Synopsys Design Constraints File is required by th
⚠	332068	No clocks defined in design.
⚠	20028	Parallel compilation is not licensed and has been disabled
⚠	332012	Synopsys Design Constraints File file not found: 'hiLow.sdc'. A Synopsys Design Constraints File is required by th
⚠	332068	No clocks defined in design.
⚠	332068	No clocks defined in design.
⚠	332068	No clocks defined in design.

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

High Low Guessing Game With Hints

Outcomes and Objectives

The outcome of this lab is to modify the high Low Guess circuit to add increased functionality. Through this process you will achieve the following learning objectives.

- Wire Logic
- Designing glue logic to interface building blocks
- Analyzing a circuit with a combination of building blocks
- Analyzing and designing a Verilog testbench
- Creating a pin assignment for a module

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, it's worth reviewing the guessing game because we will use some of the terms in the description of our enhanced functionality. The guessing 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 lab 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 20 are the same as last week's assignment with the exception of the **hotCold** button and **clue** 7-segment display.

The inputs and outputs include all the signals from last week's assignment with the addition of a **hotCold** button and **clue** 7-segment display.

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. This difference, called *difference* is compared against warmThreshold and coldThreshold as shown in Figure 21. This figure can be interpreted as follows:

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

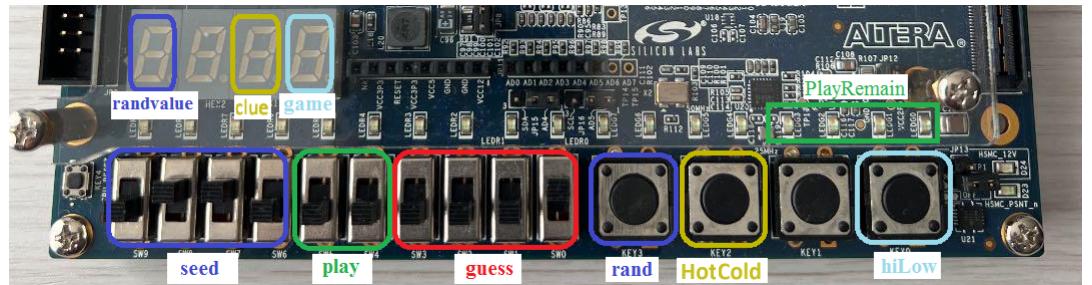


Figure 20: The input and output you should use to realize your digital system.

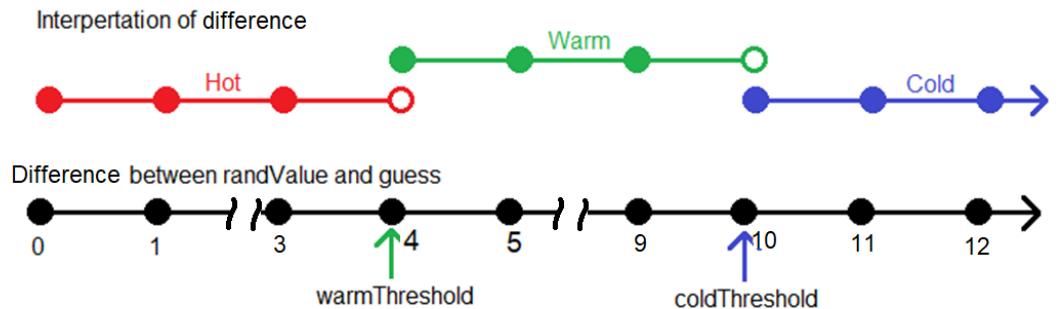


Figure 21: The interpretation of the quality of a guess in terms of thresholds.

Explore the relationship between guess, secret number and the quality of the guess by completing Table 19. Assume a 4-bit word size for guess and the secret number and use warmThreshold = 4 and ColdThreshold=10.

Table 19: 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

<i>guess</i>	<i>secret number</i>	<i>difference</i>	<i>Quality</i>
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 Cold, 'A' if the guess is wArm, 'H' if the guess is Hot.

System Architecture

You will use the system architecture shown in Figure 22 to design your circuit.

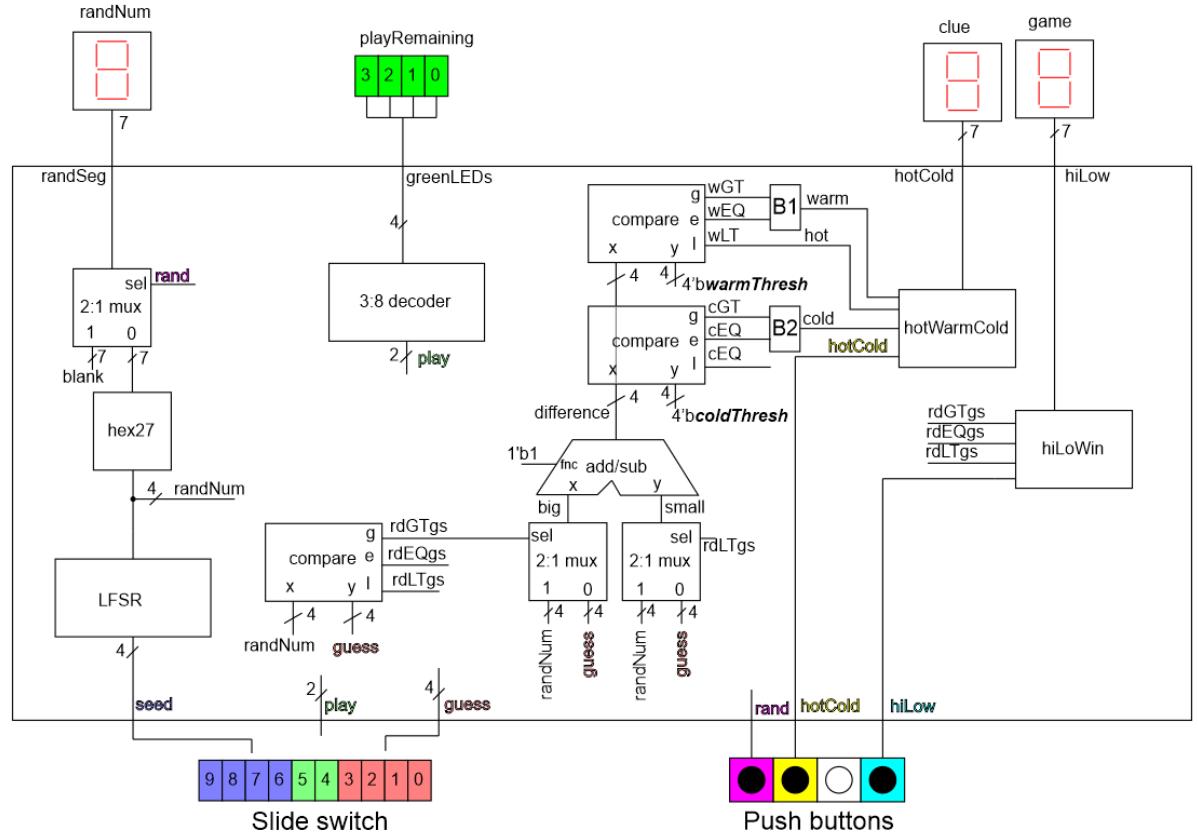


Figure 22: System architecture for the guessing game with hint.

You should use last week's lab as a starting point. The new circuitry is shown in the middle of Figure 22. Let's walk through the new circuitry to understand what it does to provide a hint about the quality of the user's guess.

Let's start by examining the output of the add/sub, *difference* which is the absolute value of the difference between the *randNum* and *guess*. It is computed by taking the difference between the larger of *randNum* and *guess* and subtracting the smaller of *randNum* and *guess*. The add/sub is hardwired to subtract because its fun input is hardwired to 1'b1, hence it is always computing the difference between its *x,y* inputs, *big-small*.

The 2:1 mux in front of the add/sub's *x* input passes *randNum* to its output, called *big*, when the comparator to its left outputs 1. This comparator outputs 1 when *randNum* is greater than *guess*. Hence when *randNum* is larger it is the *big* output.

The output of the 2:1 mux in front of the add/sub's *y* input, *small* follows a similar logical organization, but notice that the 1 and 0 inputs to this mux are the opposite of those on the *big* mux.

Now that we have *difference* computed correctly, let's see how it's used to inform the user of the quality of their guess. The *difference* is compared to the *warmThreshold* and *coldThreshold*

using a pair of comparators. Through some logic that you will design you will create three signals *hot*, *warm*, *cold* which are based on the relationship you examined in Figure 21.

Module: 2:1 Mux

This module was discussed in Lab 4.

Module: Compare

This module was discussed in Lab 4.

Module: Add/Sub

A N-bit adder subtractor is a basic building block in many digital systems. The N-bit adder subtractor shown adds its N-bit input *x* and *y* when *fnc* = 0 and subtracts *y* from *x* when *func*=1. 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.

The Verilog code for the N-bit adder subtractor is on Canvas. Since the adder subtractor uses full-adders in its construction, you will need to include the full adder module contained in the file *fullAdder.v* in your project. Listing 2 shows the module declaration for the genericAdderSubtractor. The module instantiation shown in Listing 2 corresponds to the system architecture shown in Figure 22. 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 6.

Listing 6: Top, module definition for an adder subtractor. Bottom, module instantiation of the adder subtractor in Figure 22.

```
// Module definition for the adder subtractor
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.

Logic: hotWarmCold

The goal of this section is to determine the logic inside the **hotWarmCold** logic block in Figure 22. To do this you will first need to form three signals, *hot*, *warm*, and *cold*. These three signals describe how close the *guess* is from *randNum*. We will call the comparator that compares *difference* and *warmThresh*, the warm comparator and call its three outputs *wGT*, *wEQ*

and wLT. The other comparator is the cold comparator and its outputs prefixed with lowercase “c”.

The values of *warmThresh* and *coldThresh* are set using the signal declaration and signal assignment statements shown in Listing 7.

Listing 7: The signal declaration and assignment for guess thresholds.

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

The warm and cold comparators generate a total of 6 signals, some of which are sent to the logic block B1 and B2 to form the warm and cold signals - the logic for the hot signal is given to you in Figure 22.

To understand the logic inside the B1 and B2 logic blocks let’s work through a set of examples in Table 20. In these examples you will let:

- *coldThresh* = 10
- *warmThresh* = 4
- *difference* = 9

as a first step, let’s determine the outputs of the warm comprarator. The x input o the waarm omprator is equal to 9 (the value of *difference*) and the y input equals 4 (the value of *warmThresh*). Since 9 is greater than 4, then:

- wGT = 1
- wEQ =0
- wLT =0

Using similr reasoning we ind that:

- cGT = 0
- cEQ =0
- cLT =1

Finally, since *difference* equals 9 and this is between the warm and cold thresholds, the quality of the guess is warm, so we set this output equal to 1 and hot and cold to 0.

Table 20: Complete the following table, let *warmThresh* = 4 and *coldThresh* = 10. Leave comprator outputs which re 0 blank.

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

Now, you need to use the values in Table 20 to determine the logic for each of the three outputs from the “discrete logic” block shown in Figure 22. To do this write an expression using AND and OR to describe when that output equals 1.

```
  cold =                                // write logic description
  warm =                                // write logic description
  hot = wLT;                            // given to you in Figure~\ref{fig:guessWithHintSystemArch}
```

For the hotWarmCold 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.

One you have the logic or the *hot/wrm* nd *cold* signals you can start building the *hotWrmcold* logic block. You will implement the *hotWarmCold* logic using an always/case statement that uses the 3-bit vector *hot*, *warm* and *cold* to form the 7-segment output shown in Figure 23 when the *hotCold* button is pressed. The 7-segment display should be blank when the *hotCold* button is unpressed.

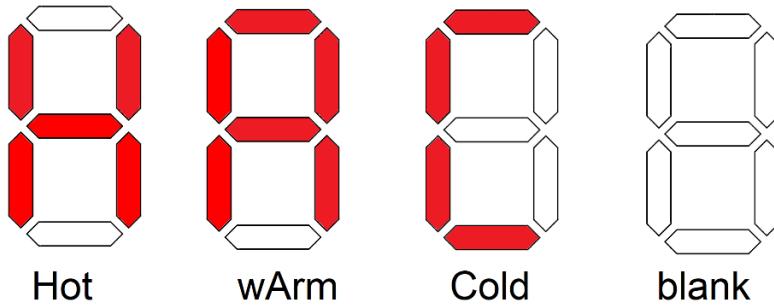


Figure 23: The illuminated patterns to inform the guesser about the magnitude of their guess.

For this block of code:

- Use an always/case statement.
- 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 8.

Listing 8: 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]
```

```
//*****
```

Module: hiLow

This is the top-level module in Figure 22, the outermost block. 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 9. If you got the previous lab working correctly, then you should use the bottom declaration in Listing 9.

Listing 9: The module declaration for the enhanced hiLow module if you did or did not get the previous lab working.

```
// if you did not get the previous lab working, then use this module declaration
module hiLow(seedSwitch, guessSwitch, hotColdBut, hotColdSeg);

// If you successfully completed previous lab, then use this module declaration (with no line break)
module hiLow(seedSwitch, playSwitch, guessSwitch, randBut, hotColdBut, hiLowBut, randMsbSeg, randLsbSeg,
greenLEDs, hotColdSeg, hiLowSeg);
```

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.

Testbench

The testbench 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 21 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 21 are borrowed from the system architecture in Figure 22. 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 21: Table : The values used in the hiLow testbench.

Test	seed	randNum	guess	big	small	Difference	Comment
1			4'b1111				
2			=14				
3	4'b1010	4'b0100	4'b1101 =				
4			4'b1000 =				
5			4'b0111 =				

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

When you figure out what the testbench will output, its time to run it. Use the testbench provided on Canvas. Produce a timing diagram with the following waves with the correct radix and color and order the traces from top to bottom as

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

When compete, your testbench should look like the timing diagram in Figure 24.

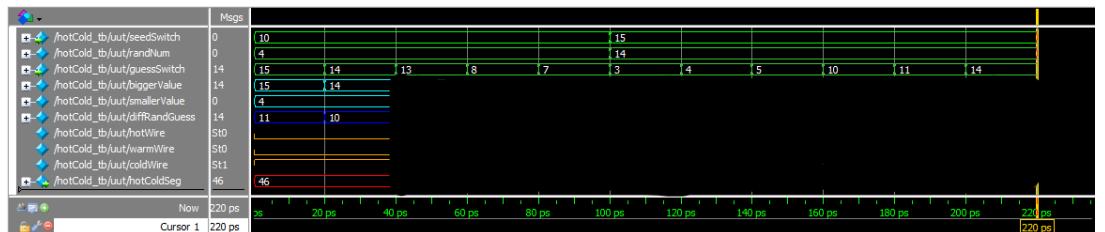


Figure 24: A partially obscured timing diagram generated by the testbench.

Pin-Assignment and Synthesis

Use the image of the C5G Development Board in Figure 20 and the information in the User Guide to determine the FPGA pins associated with the input and output devices used by the hiLow module.

Table 22: Pin-assignment for the High Low Guessing Game with Hints.

Segment	randSeg	hotColdSeg	hiLowSeg
seg[6]	AC22		
seg[5]			

Table 22: Pin-assignment for the High Low Guessing Game with Hints.

Segment	randSeg	hotColdSeg	hiLowSeg
seg[4]			
seg[3]			W18
seg[2]	AA23		
seg[1]			
seg[0]			V19

	seedSwitch	playSwitch	guessSwitch
slide[3]	AE19	N/A	
slide[2]		N/A	
slide[1]			AE10
slide[0]		W11	

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

G[3]	G[2]	G[1]	G[0]
E9			

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

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

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 19.

Discrete Logic block

- Complete Table 20
- [Link](#) Logic for hot, warm, and cold signals

Module: hiLow

- [Link](#) Verilog code for the body of the hiLow module (courier 8-point font single spaced), leave out header comments.
- Complete Table 21.
- [Link](#) Complete testbench timing diagram.

Pin-Assignment and Synthesis

- Complete pin-assignment table or ll the signals in Table 22.
- Demonstrate your completed circuit to a lab team member.

Debugging Tips

Even when your program executes successfully, you may get the warnings shown in Figure 25. 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.

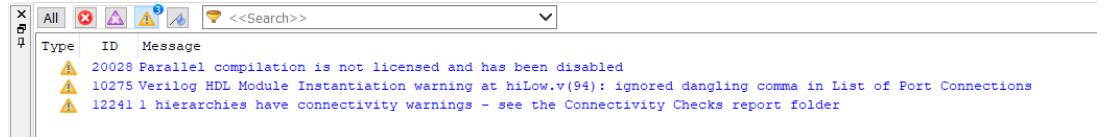


Figure 25: The messages console filtered by warnings.

The Connectivity Checks folder in the Compilation Report will help you 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 26, I selected the genericAdderSubtractor, note the fnc input is hardwired 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.

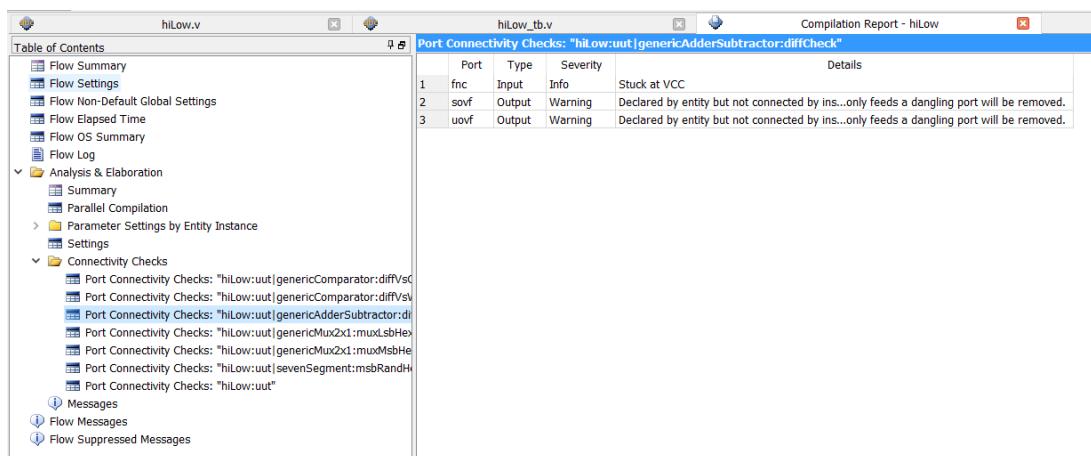


Figure 26: Connectivity Checks report for a working hiLow circuit.

Calculator With Friendly Output

Outcomes and Objectives

The outcome of this lab is to instantiate a calculator with signed decimal output making it easy for anyone to use the circuit. Through this process you will achieve the following learning objectives.

- Wire Logic
- Designing glue logic to interface building blocks
- Analyzing a circuit with a combination of building blocks
- Writing a Verilog statement using an Always/CaseZ statement
- Synthesizing a module on the FPGA development board

Calculator with Friendly Output

This week you are going to build a calculator that can add or subtract 4-bit values using the input and output shown in Figure 27 and display the results as decimal, base-10, values, not as hexadecimal values.

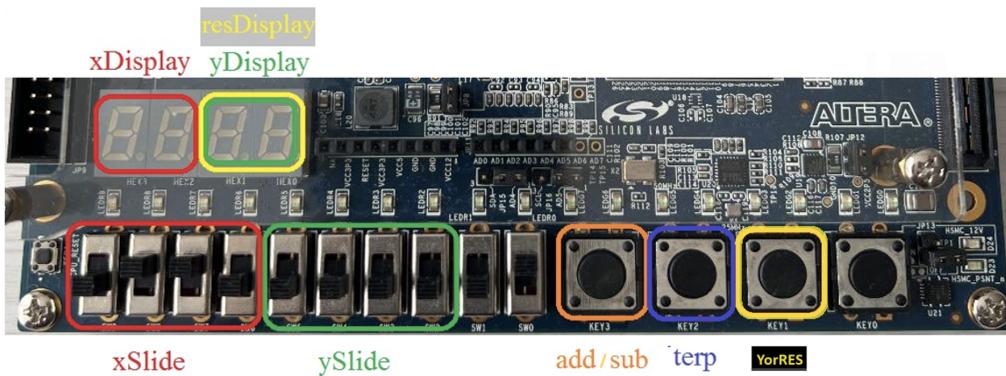


Figure 27: The input and output of the calculator digital circuit.

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 27. 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 four 7-segment displays on board, the same 7-segment displays of operand Y will be used to show the operation result (yellow) when the **yOrResult** button is pressed.

System Architecture

The system architecture shown in Figure 28 shows the adder subtractor processing the **xSlide** and **ySlide** inputs. The 4-bit x, y and result values are processed by the **sigUnsig** module before being displayed on the 7-segment displays. It is now time to turn our attention to this module.

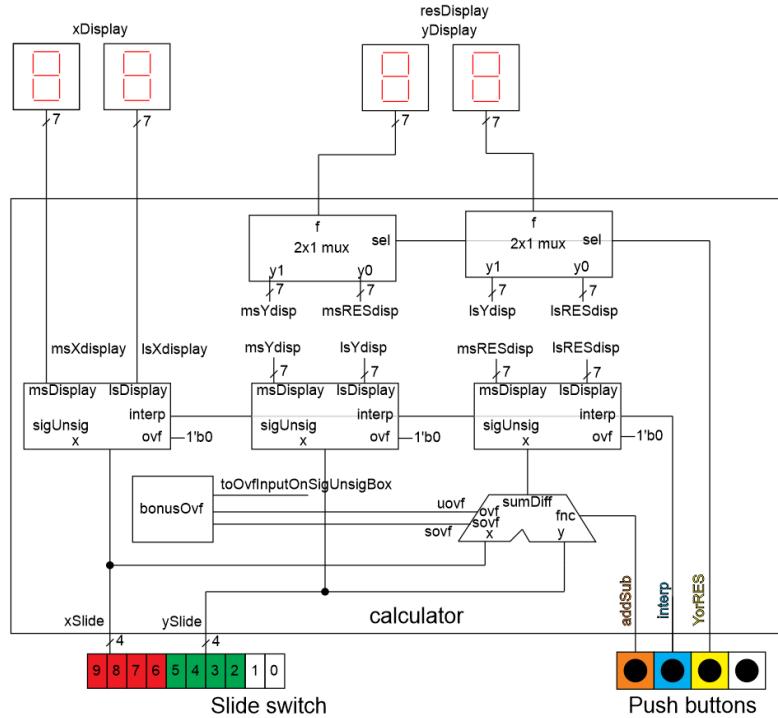


Figure 28: The system architecture of the calculator.

Module: sigUnsign

The significant design problem in today's lab comes in this section, building the `sigUnsign` module that shows up three times in Figure 28. This module takes in a 4-bit value and displays a 2-digit signed or unsigned representation on a pair of 7-segment displays. The `sigUnsign` module declaration is shown in Listing 10.

Listing 10: 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 `msDisplay` is the most significant (ms) symbol being displayed and `lsDisplay` is the least significant (ls) symbol being display. The term "symbol" is used because more than one type of information can be displayed depending on the values of the inputs. Let's explore this.

For example, let `x = 4'b1100`.

- If `interp = 1'b0` then `x` is interpreted as unsigned and its value is 12. Then the `msDisplay` should show "1" and `lsDisplay` "2".
- If `interp = 1'b1` then `x` is interpreted as 2's complement and its value is -4. Then the `msDisplay` should show "-" and `lsDisplay` "4".
- In the previous two cases we assumed, without stating it, that `ovf = 1'b0`. If `ovf = 1'b1` then the operation which generated `x` overflowed and the value of `x` is invalid. In this case both display's should show "X". Since we are working with 7-segments, our "X" looks much more like "H" :(

Not complete Table 26 by filling in the values of `msDisplay` and `lsDisplay` for a signed and unsigned interpretation, assuming `ovf=0`. If the interpreted value is positive and a single digit then assign `msDisplay` blank. If the interpreted value is negative then assign `msDisplay` "-". If the interpreted value is greater than 10, assign `msDisplay` "1".

Table 26: 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	blank	0	blank	0
4'b0001				
4'b0010				
4'b0011				
4'b0100				
4'b0101				
4'b0110				
4'b0111				
4'b1000				
4'b1001				
4'b1010				

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

Take a moment and look at the patterns in Table 26. You should make the following important observations.

- `msDisplay` is assigned one of four values
 - `blank` when the interpretation of `x` is an unsigned or signed value less than 10.
 - 1 when the interpretation of `x` is an unsigned value greater than 10.
 - - when the interpretation of `x` is a **signed value less than 0**.
 - X (the invalid character) when the `ovf = 1`.
- `lsDisplay` is assigned one of four values,
 - `x` (the value of the `x` input) when the interpretation of `x` is a unsigned or signed value less than 10.
 - `x-10` when the interpretation of `x` is an unsigned value greater than 10.
 - `0-x` when the interpretation of `x` is a **signed value less than 0**.
 - X (the invalid character) when the `ovf = 1`.

Why are we taking the 2's complement of `x`?

Please take a moment and reflect on pair of rows where you are asked to interpret `x` as a **signed value less than 0**. Under a signed (2's complement) interpretation, if the most significant bit of `x` is 1 then the value of `x` is less than 0. In this case the `msDisplay` 7-segment display should be illuminated with a “-” to indicate negative. The `lsDisplay` needs to show the negation of `x` because the negation of a negative number is a positive number and we can use a `hexToSevenSeg` module to display positive numbers. This is a complex but important observation.

If you follow the above reasoning, there is a need to form the 2's complement of `x` in certain input situations. You will form the negation of `x` by subtracting `x` from 0, that is compute `0-x`. You will do this by putting `4'b0000` on the `x` input of an `addSub`, put the `sigUnsign` input `x` on the `y` input of an `addSub`, and hardwire the `fnc` input to `1'b1` so that the `addSub` is hardwired to always subtract.

Formalized the observations in a more algorithmic syntax by completing Listing 11. Do this by filling the values of `msDisplay` and `lsDisplay` for the different input conditions. The values for these two signals are given in the two lists above. Note that this code is NOT to be used in your actual code for this lab.

Listing 11: Logic that determines the output of the 4:1 muxes in Figure 30.

```

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 that you know what should be displayed on `msDisplay` and `lsDisplay`, let's look at how we can form these symbols on the 7-segment displays. In order to do this, you need Figure 29, the bit-order of the segments controlling the illumination of the segments. 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".

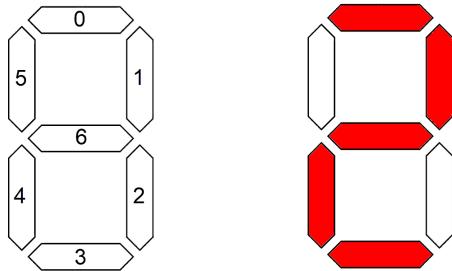
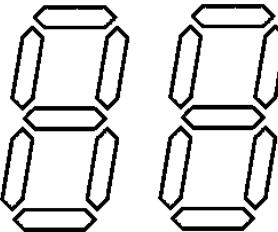
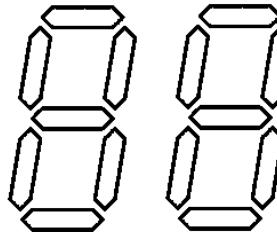
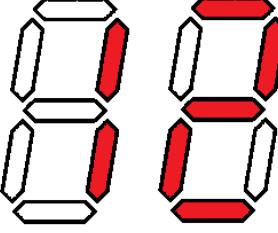
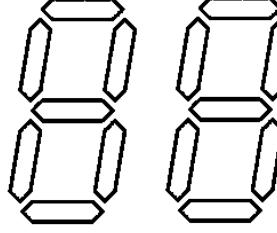
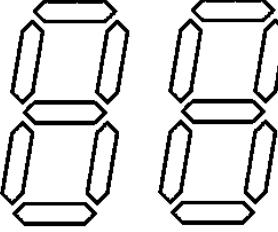
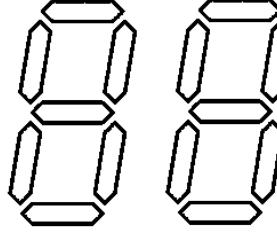


Figure 29: The logical arrangements of the segments in a 7-segment display.

Test your understanding of the `signUnsign` output by completing Table 27. Do this by coloring 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 to the right of `msDisplay=7'b` and `lsDisplay=7'b`.

Table 27: 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		4'b0111 interp = 0 ovf = 0	
	msDisplay = 7'b lsDisplay = 7'b		msDisplay = 7'b lsDisplay = 7'b
4'b1100 interp = 0 ovf = 0		4'b1000 interp = 1 ovf = 0	
	msDisplay = 7'b1111001 = 7'h79 lsDisplay = 7'b0100100 = 7'h24		msDisplay = 7'b lsDisplay = 7'b
4'b1100 interp = 1 ovf = 0		4'b1010 interp = 1 ovf = 1	
	msDisplay = 7'b lsDisplay = 7'b		msDisplay = 7'b lsDisplay = 7'b

Now we are ready to put the pieces of the sigUnsig module together. The building blocks in Figure 30 are captured in the organization described by Listing 11, along with some extra hardware.

Complete Figure 30 by adding the following:

- Connect the inputs of the 4:1 mux using the logic described in Listing 11.

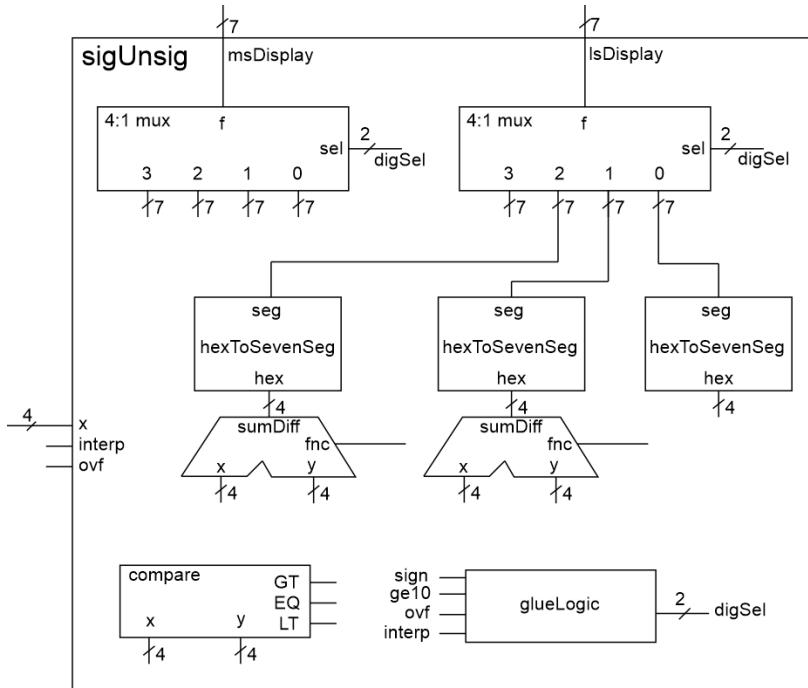


Figure 30: The internal architecture of the signUnsig module.

- Connect the inputs of the adder subtractors using the logic described in Listing 11.
- Wire the inputs of the comparator to generate the signal `xGE10` which is logic 1 when `x` is greater than or equal to 10.
- Wire the input of the rightmost hexToSevenSeg .

All that remains is to define the contents of the `glueLogic` box in Figure 30.

`glueLogic always/casez statement`

the `glueLogic` box chooses which input of the 4 mux inputs to route to the output. This is the logic that you formalized in Listing 11. Note the signal `sign` which equals 1 when `x` represents a negative value when interpreted as a signed value.

Now complete the truth table in Table 28 for the `glueLogic` box in Figure 30.

Table 28: 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	
0	1	1	x	

It would make sense to use an always case statement to realize the logic in Listing 11. However, an always case statement requires each of the 16 difference cases to be explicitly

enumerated. However, the truth table in Listing 11 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 11, 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 12.

Listing 12: 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
```

sigUnsig Verilog code

The Verilog code for the sigUnsig 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 10.
- Use the module definitions for
 - genericMux4x1 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 localparam to give names to the 7-bit constant patterns (fill in the values for x).
 - localparam [6:0] displayBlank = 7'bxxxxxxxx;
 - localparam [6:0] displayOne = 7'bxxxxxxxx;
 - localparam [6:0] displayMinus = 7'bxxxxxxxx;
 - localparam [6:0] displayX = 7'bxxxxxxxx;
- 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

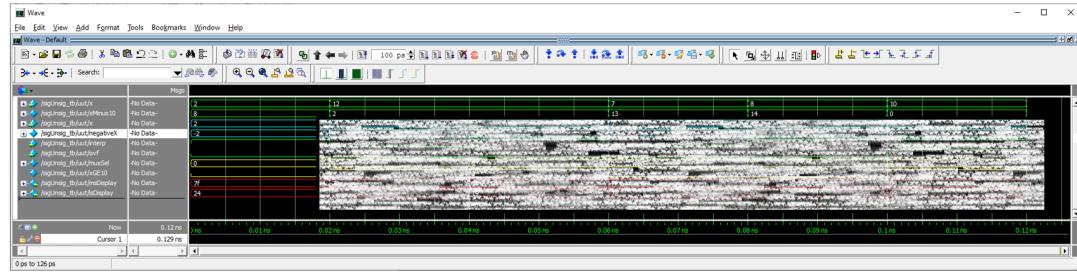
Testbench

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 and Synthesis

Use the image of the FPGA Development Board in Figure 27 and the information in the C5G User Guide to determine the FPGA pins associated with the input and output devices used by the devices used by the `calculator` module.

Table 29: Pin Assignment for the calculator.

Segment	msXdisplay	lsXdisplay	msYorRESdisplay	lsYorRESdisplay
seg[6]	AC22			
seg[5]		W21		
seg[4]			AE25	
seg[3]				W18
seg[2]				
seg[1]				
seg[0]				

	x	y
slide[3]	AE19	

	x	y
slide[2]		W11
slide[1]		
slide[0]		

YorRES	Key[1]	P12
interp	Key[2]	
addSub	Key[3]	

Complete the pin-assignment in Quartus, compile your design and download to the FGPA development boards. Once you get your design working, demonstrate it to a member of the lab team.

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 26.
- Complete Table 27.
- Complete the code in Listing 11.
- Complete Figure 30, including:
 - Constant values on inputs of 4:1 mux
 - Constant value on the input of the right-most hexToSevenSeg
 - Value on the input of the adder subtractors
 - Values on the input of the comparator
- Complete Table 27.
- Complete Table 28.
- **Verilog code for the body of the sigUnsig module** (courier 8-point font single spaced), leave out header comments.

Testbench

- Complete testbench and timing diagram from Section .

Pin-Assignment and Synthesis

- Complete the pin assignment in table 29.
- Demonstrate your circuit to a member of the lab team.

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.

Cellular Automata

Outcomes and Objectives

The outcome of this lab is to instantiate a 1-dimensional cellular automata using D-Flip Flops and combinational logic. Through this process you will achieve the following learning objectives.

- Basic Memory Element Behavior
- Timing of basic memory element
- Definition of Verilog modules
- Synthesizing a module on the FPGA development board

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.

Theory: Cellular Automata

As a testbench for this idea, Stephen Wolfram¹ 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 31, note that cells which are alive are shaded and cells which are dead, white.

Let's examine the next state of the 1-DCA shown in Figure 31 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.

Complete the unfilled entries in Table 32 using the initial state shown in Figure 31 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.

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

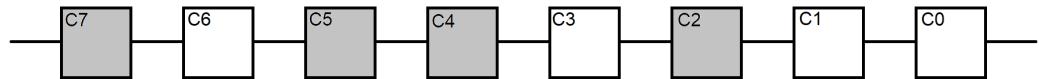


Figure 31: A 1-DCA consisting of 8 processing elements. Alive cells are shaded grey, dead white.

Table 32: 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 9-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 32. 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.

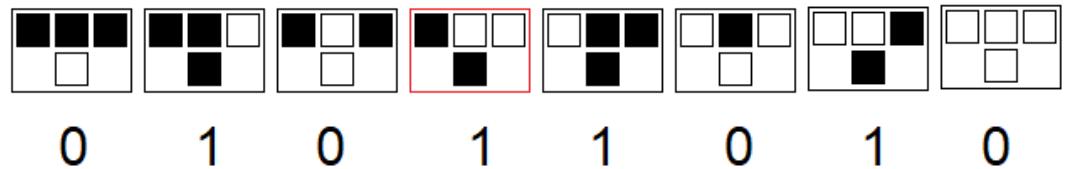


Figure 32: The next state of the center cell depends on the current state of a cell and its neighbors.

As an example, let's look at the 4th 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 32, 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 32. 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 33. Some of you may recognize this figure as the Sierpiński Triangle², an elementary fractal.

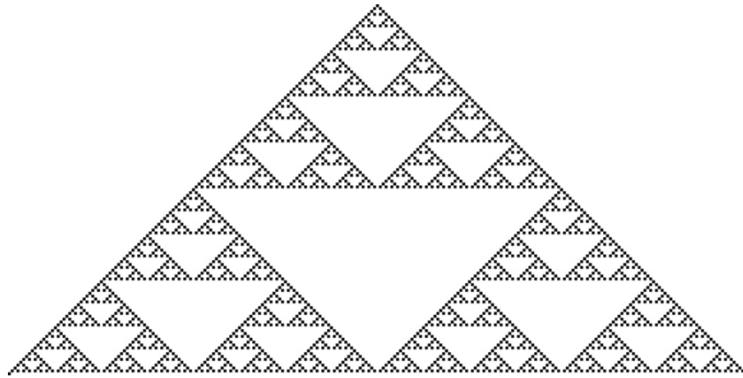


Figure 33: The evolution of a single alive cell (shown at top) under Rule 90.

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.

Implementation: Cellular Automata

You will use the inputs and outputs shown in Figure 34 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

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

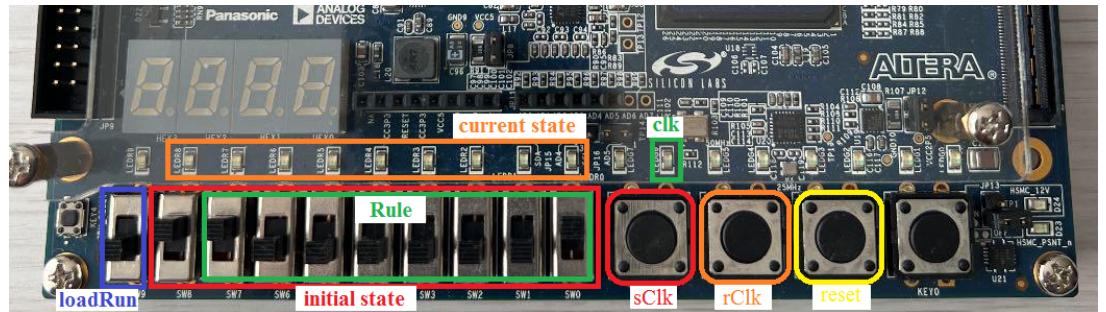


Figure 34: The input and output of the 1D cellular automata.

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

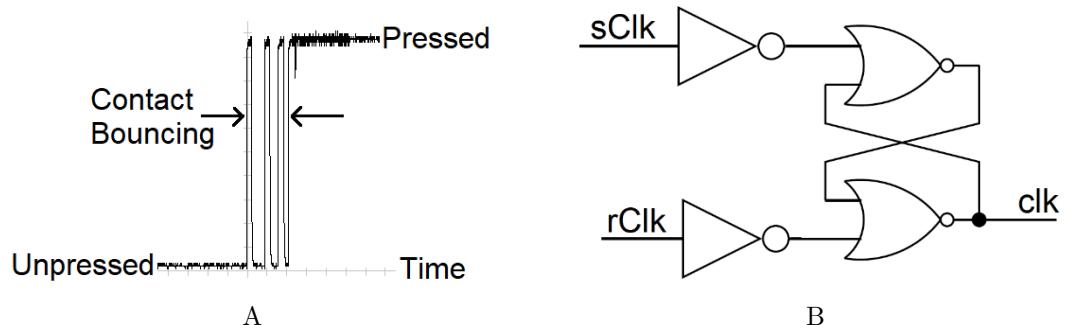


Figure 35: A) Switch bouncing makes generating a single clock edge problematic. B) The organization of the SR latch to run the clock requires inverters in the sClk and rClk inputs and an SR latch to help remove signal bouncing.

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 35B. 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...

System Architecture

The system architecture shown in Figure 36 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.

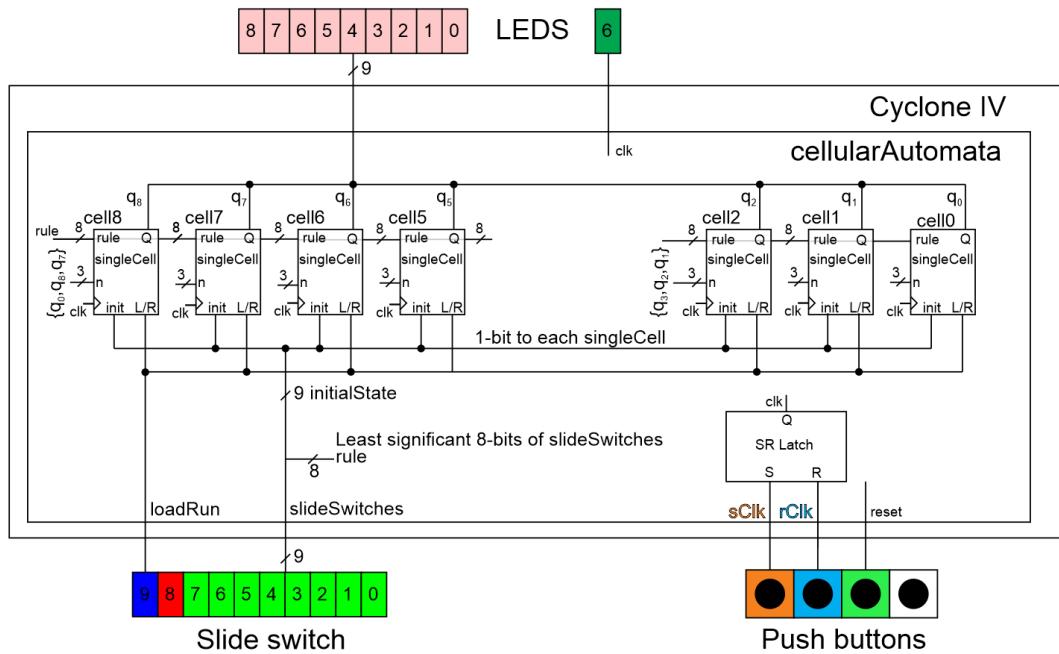


Figure 36: The system architecture of the cellularAutomata. Due to tight spacing, only the n inputs to cell8 and cell2 are shown.

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

- 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 36
 - Make cell 0 have cell 8 as its “right” neighbor in Figure 36
- Use cross-couples NORs and a pair of inverters 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.

Module: singleCell

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 36 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 37. For example, the output `currentState` in this figure was called `Q` in Figure 36. You should be able to decipher the rest of the signal abbreviations used in Figure 36.

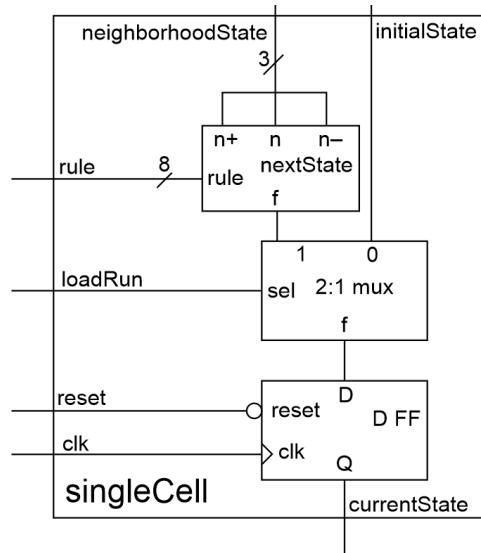


Figure 37: The architecture and module interface for the `singleCell` module.

The most complex portion of logic in the `singleCell` module is the box labeled “`nextState`”. This circuit has 11-bits of input. 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 33 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 33 for Rule 90 using the information in Figure 32. In the “Rule bit” column in Table 33, generalize the `nextState` column to the bit values in the 8-bit rule vector that is passed into the `singleCell` module.

Table 33: The input/output relationship for the `nextState` functionality in Figure 37.

$\{n+, n, n-\}$	<code>nextState</code>	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

Testbench

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

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

Do not use 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 38.

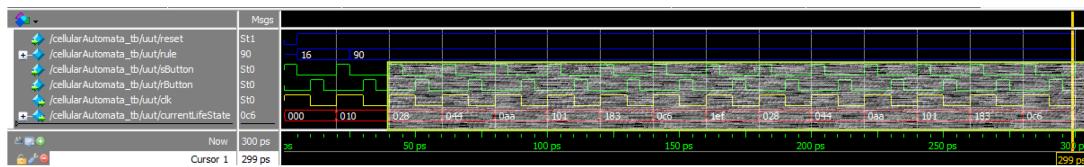


Figure 38: Partial timing diagram for the cellular automata.

Pin-Assignment and Synthesis

Use the image of the FPGA Board in Figure 31 and the information in the C5G User Guide to determine the FPGA pins associated with the input and output devices used by the cellular automata module.

Table 34: Pin Assignment for the 1-D cellular automata.

slide[9]	AE19		Clk LEDG6	
slide[8]			led [8] LEDR8	
slide[7]			led [7]	K8
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	

sClk	Key[3]	
rClk	Key[2]	Y15
reset	Key[1]	

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

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

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 32.
- Link Verilog code for the body of the cellularAutomata module (courier 8-point font single spaced), leave out header comments.

singleCell Module

- Complete Table 33.
- [Link](#) Verilog code for the body of the singleCell module (courier 8-point font single spaced), leave out header comments.

Testbench

- Complete testbench and timing diagram from Section .

Pin-Assignment and Synthesis

- Complete the pin assignment in table 34.
- Demonstrate your completed circuit to a lab team member.

Debugging Tips

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 “q₈q₇q₆q₅” is the output from the 4 most significant bits of the CA
- The column labeled “q₄” is the output from the middle bit of the CA
- The column labeled “q₃q₂q₁q₀” is the output from the 4 least significant bits of the CA

Rule 0: 0000 0000	q ₈ q ₇ q ₆ q ₅	q ₄	q ₃ q ₂ q ₁ q 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

Mod 10 Counter

Outcomes and Objectives

The outcome of this lab is to isimulate a mod10 counter that will be used as a building block in the implementation of a digital stopwatch. Through this process you will achieve the following learning objectives.

- Designing glue logic to interface building blocks
- Design a circuit that performs register transfer
- Creating a simulation timing diagram for a module
- Creating a Do file to automate waveform setup

Module: mod10Counter

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 “ $x \text{ mod } 10$ ” you get the integer remainder after division by 10. For example, $12 \text{ mod } 10$ is equal to 2 because $12/10 = 1$ with a remainder of 2. Note that “ $x \text{ mod } 10$ ” 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 39. 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.

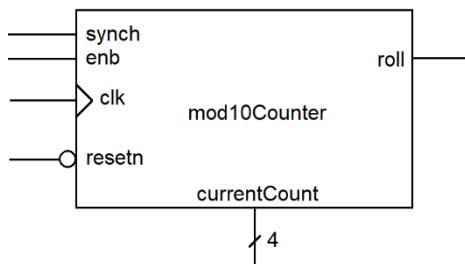


Figure 39: The high-level interface for the mod 10 counter.

Table 37 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 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 37: 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 38 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 38: The truth table for the roll output from the mod10Counter.

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

System Architecture

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 40.

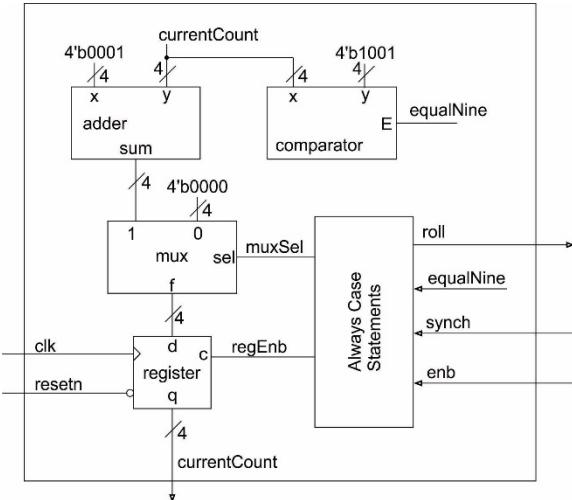


Figure 40: The internal organization of the mod10counter module.

You have been provided with the adder, comparator, mux, and register shown in Figure 40. 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 37 and Table 38 along with the hardware organization in Figure 40 to fill in Table 39 .

Table 39: 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`.

Complete the coding of the `mod10Counter` instantiating the the modules in Figure 40 and including the always/case statement as a CSA. Once you complete writing this code, you will need to demonstrate its proper functioning by simulating the design. You will not be synthesizing your design this week, so it's imperative that you carefully examine the output because you will be using this module in the coming weeks as an element in a larger design.

Testbench

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 41 using the value from the testbench.

TESTBENCH

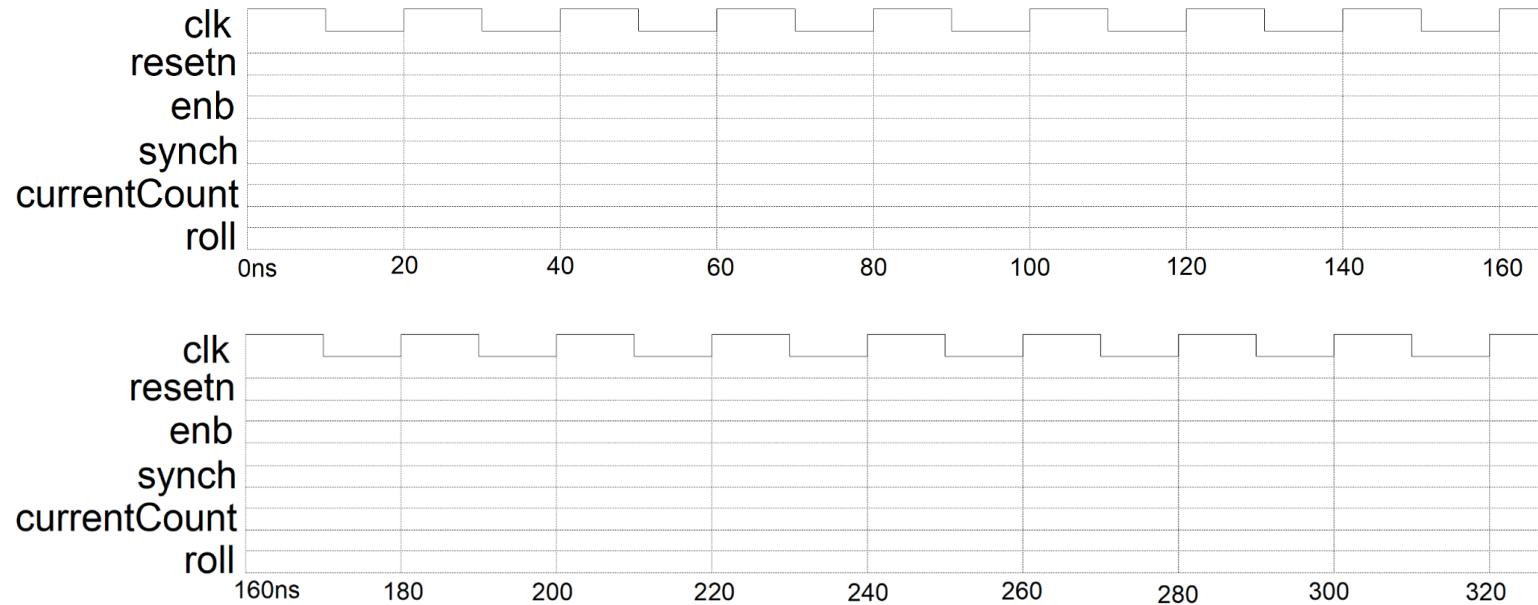


Figure 41: A timing diagram for you to fill out based on the testbench code for the mod10Counter. Note the diagram was too wide to be readable in one line, so it has been broken at 160ns to make it easier to you to read.

FPGA: Do file

You have been required to run simulation for almost every lab. This is not busy-work, all practicing digital designers are expected to demonstrate proper functionality before modules are incorporated into larger design. You may have experienced times when you had to run a simulation multiple times to get your design correct. It would surprise me if you took joy at setting up all the signals, colors and radix in a timing diagram only to find an error and have to repeat all this work again. A Do file is a script the sets up the waveforms for a simulation including their order, colors and radices.

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 13: 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:

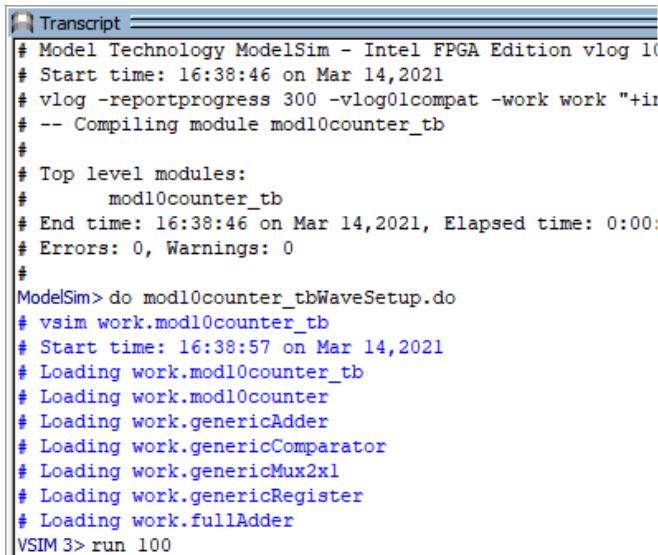
```
<projectDirectory>\mod10Counter\simulation\modelsim
```

Where `<projectDirectory>` is the path of your project. If this folder does not exist, 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 42.

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



```
# Model Technology ModelSim - Intel FPGA Edition vlog 10.1
# Start time: 16:38:46 on Mar 14, 2021
# vlog -reportprogress 300 -vlog01compat -work work "+incdir+."
# -- Compiling module mod10counter_tb
#
# Top level modules:
#     mod10counter_tb
# End time: 16:38:46 on Mar 14, 2021, Elapsed time: 0:00
# Errors: 0, Warnings: 0
#
ModelSim> do mod10counter_tbWaveSetup.do
# vsim work.mod10counter_tb
# Start time: 16:38:57 on Mar 14, 2021
# Loading work.mod10counter_tb
# Loading work.mod10counter
# Loading work.genericAdder
# Loading work.genericComparator
# Loading work.genericMux2x1
# Loading work.genericRegister
# Loading work.fullAdder
VSIM 3> run 100
```

Figure 42: The console output when the mod10counter do file is run.

choice, the command/filename will be completed up to the ambiguity and the console will provide a list of candidate filenames.

You can issue a variety of commands in the console window. A very popular command is “run <time>” to simulate some amount of time.

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

signal	radix	signal color
clk	default	green
reset	default	green
enb	default	gold
synch	default	gold
roll	default	yellow
currentCount	unsigned	greenyellow

You need to look at the values produced by the mod10counter and compare them against the values in Table 37. 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 student errors are due to incorrect wiring or modules – wrong names or wrong signal order.

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 the truth table in Table 39.
- Complete hand-drawn timing diagram in Figure 41.
- **Verilog code** for the body of the mod10counter module (courier 8-point font single spaced), leave out header comments.

Testbench

- Complete **simulation timing diagram** of the `mod10counte`.
- The contents of your **Do File**.

Stopwatch Datapath

Outcomes and Objectives

The outcome of this lab is to design and simulate a datapath to store and manipulate the information need to run a stopwatch. Through this process you will achieve the following learning objectives.

- Design a circuit that performs register transfer
- Datapath and Control Architecture
- Design a control word table and specify the control word values for every state.
- Creating a Do file to automate waveform setup

Stopwatch

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 43.

To make sense out of the FSM shown in Figure 43, 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 43.

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

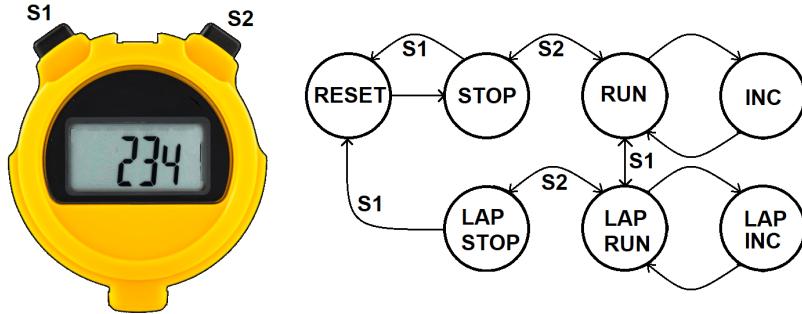


Figure 43: A digital stopwatch gets its input from 2 buttons and displays its output on a 7-segment display. The behavior of the stopwatch can be described by this finite state machine (FSM).

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 calmly 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 10th of a second. This is managed by the counter/-comparator at the top of Figure 44 which asserts the **tenth** signal every 10th of a second. This signal tells the FSM in Figure 43 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 44 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 10th second timer and hold the displayed time.
- RUN – Run the 10th second timer and display the stored time.
- INC – Increment the stored time
- LAP RUN – Run the 10th second timer and hold the displayed time.
- LAP INC - Increment the stored time.
- LAP STOP – Stop the 10th second timer and hold the displayed time.

Module: stopWatchDatapath

The datapath for the stopwatch is shown in Figure 44. The behavior of the datapath to perform the functions of a stopwatch follows.

The internal time is formed by the counter/comparator combination at the top of Figure 44. The **clk** input from the Development Board runs at 50MHz. This will be the rate at which

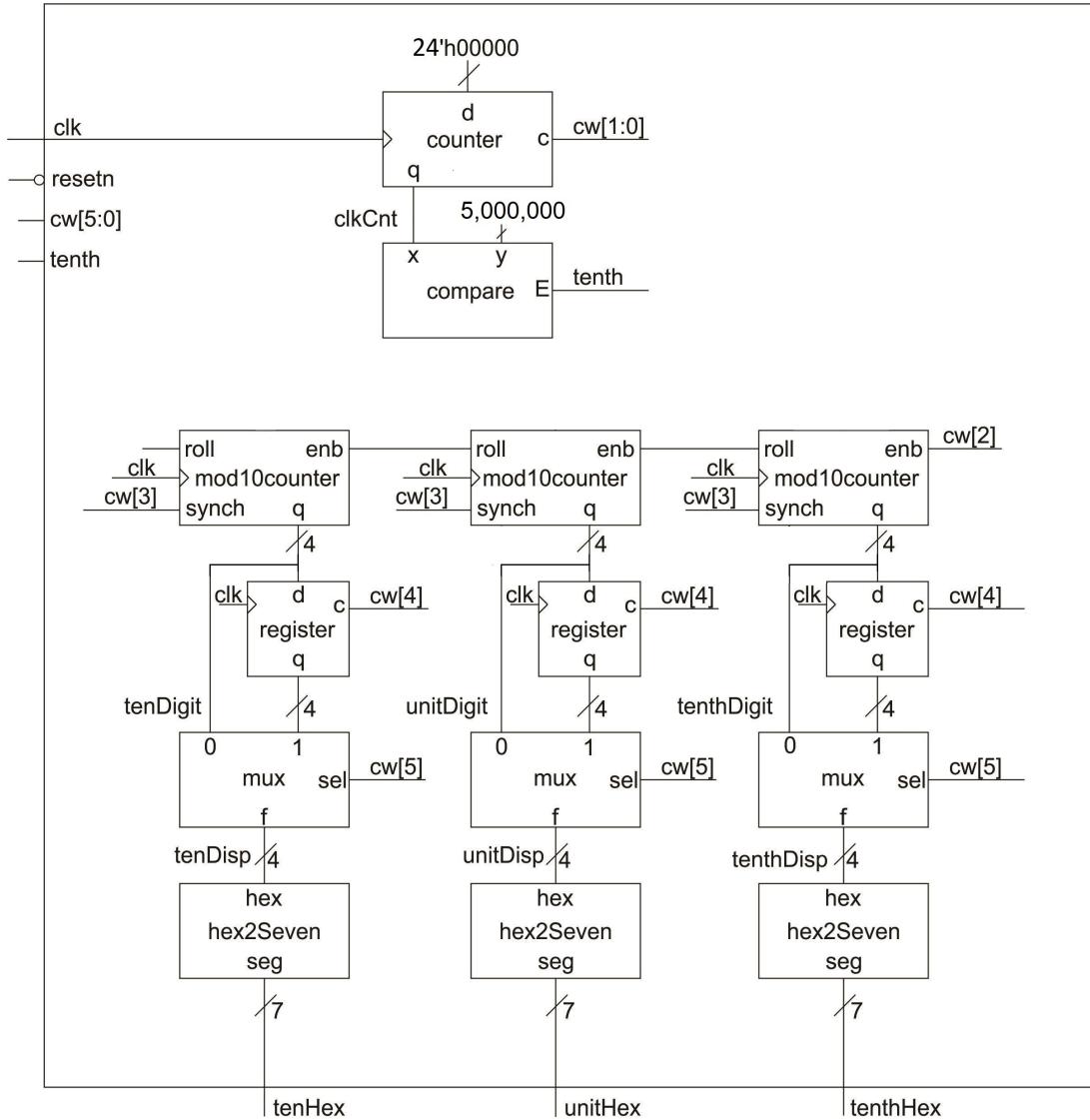


Figure 44: The datapath for the stopwatch has a 6-bit control word and displays the time on three 7-segment displays.

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 43. 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 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 40: Control word table for the datapath shown in Figure 44

	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:

- 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

Testbench

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 44 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, you will have the following genericCounter instantiation in your datapath - the use of the parameter value `#(N)` as the width of the counter is legal syntax in Verilog as long as `N` is defined somewhere.

```
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.

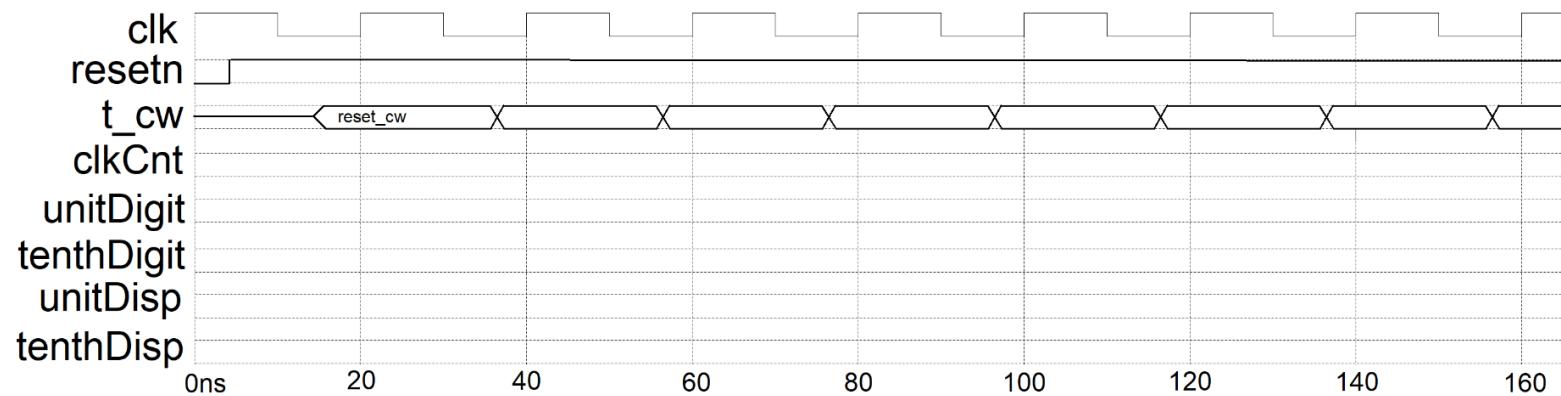
Next modify the control words defined in the datapathLab09_tb.v file using the values from Table 40.

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.

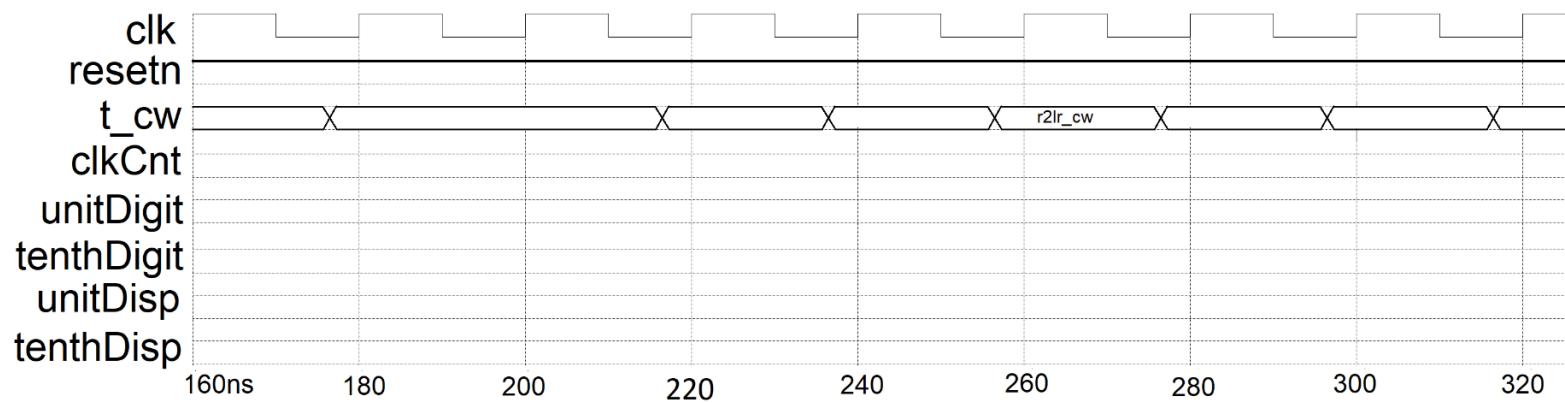
To do this complete the entries in the following timing diagram figures. Use the code in the testbench to figure out the values for the control word and how long they are held. The control word does not necessarily change every 20ns. When it does not, you can just rewrite the control word or edit the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

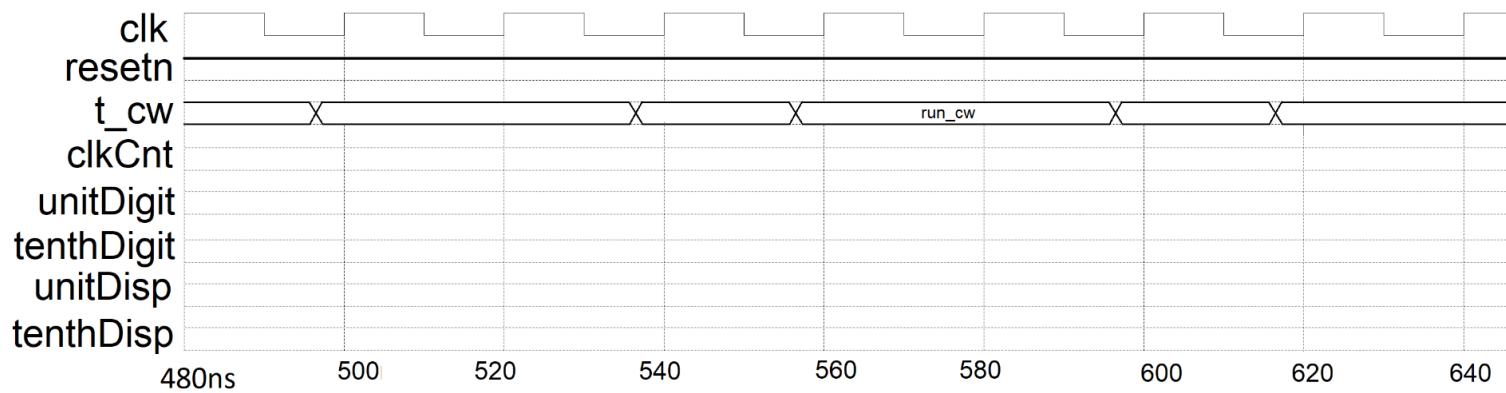
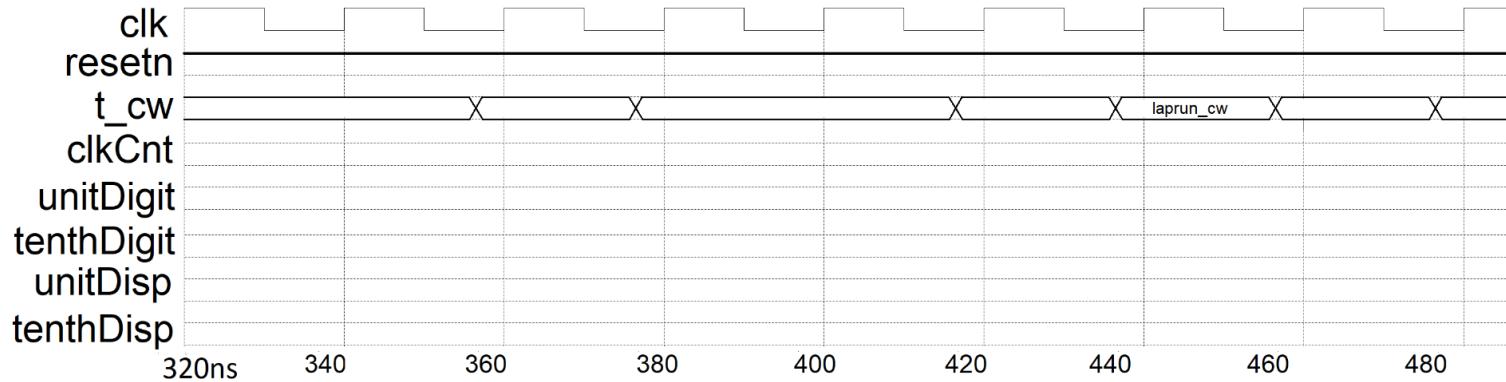
xciv



STOPWATCH DATAPATH



MODULE: STOPWATCHDATAPATH



xcv

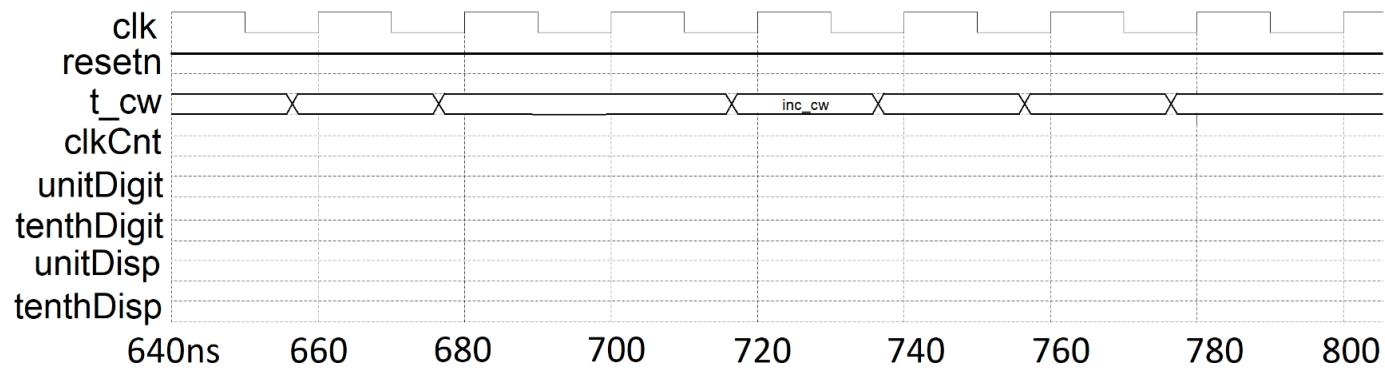


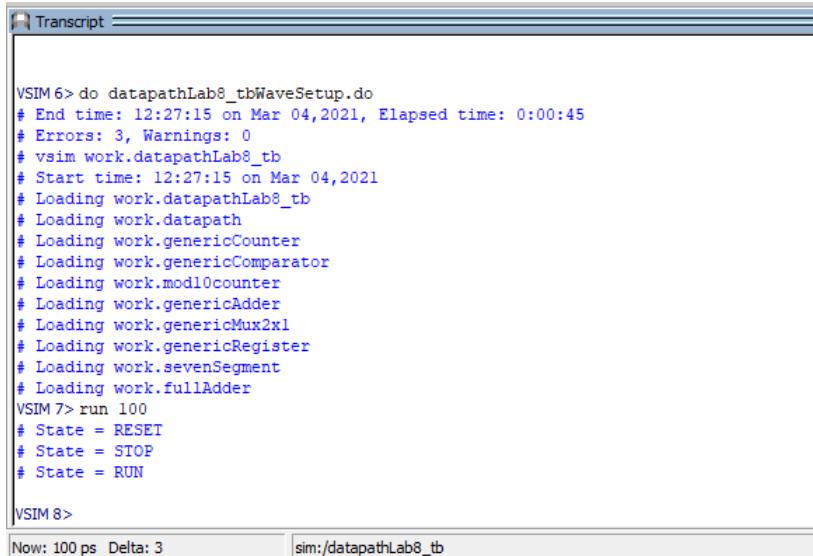
Figure 45: Complete the timing diagram using the control words found in the testbench.

Run the testbench for the datapathLab09.tb module using the do file provided. Produce a timing diagram ordered from top to bottom as:

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

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



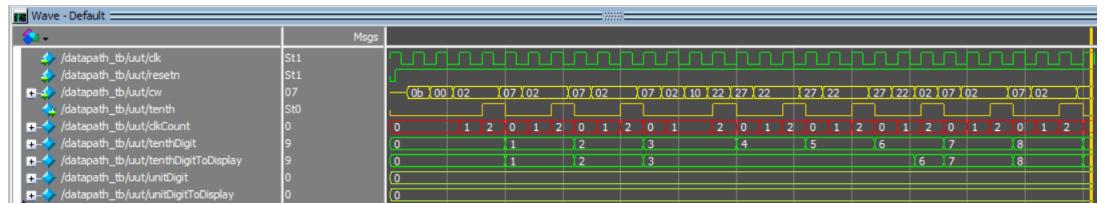
```

VSIM 6> do datapathLab8_tbWaveSetup.do
# End time: 12:27:15 on Mar 04, 2021, Elapsed time: 0:00:45
# Errors: 3, Warnings: 0
# vsim work.datapathLab8_tb
# Start time: 12:27:15 on Mar 04, 2021
# Loading work.datapathLab8_tb
# Loading work.datapath
# Loading work.genericCounter
# Loading work.genericComparator
# Loading work.modl0counter
# Loading work.genericAdder
# Loading work.genericMux2x1
# Loading work.genericRegister
# Loading work.sevenSegment
# Loading work.fullAdder
VSIM 7> run 100
# State = RESET
# State = STOP
# State = RUN
VSIM 8>

```

- You can type “run <time>” in this area (as shown) to simulate some amount of time.
- 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 errors are due to incorrect wiring of modules – wrong signal names or wrong signal order.



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 40.
- **Verilog code** for the body of the datapath module (courier 8-point font single spaced), leave out header comments.

Testbench

- Complete timing diagram from Figure 45.
- A copy of your **simulation timing** diagram. Use the signal color and order specified.
- Demonstrate that your simulation to a member of the lab team.

Stopwatch Control Unit

Outcomes and Objectives

The outcome of this lab is to design and simulate a control unit to control the datapath that stores and manipulates the information need to run a stopwatch. Through this process you will achieve the following learning objectives.

- Using a timing diagram to specify or verify the proper operation of a Finite State Machine
- Design a control word table and specify the control word values for every state.
- Creating a Do file to automate waveform setup
- Definition of Finite State Machines in Verilog

Stopwatch

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 46.

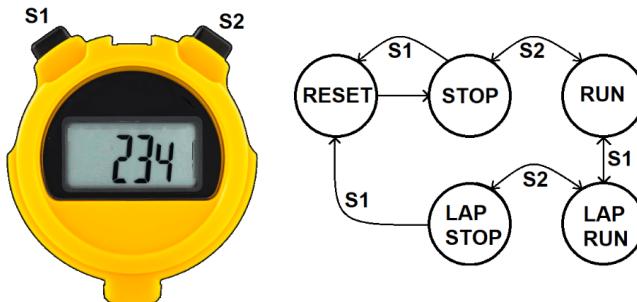


Figure 46: A digital stopwatch gets its input from 2 buttons and displays its output on a 7-segment display. The behavior of the stopwatch can be described by this finite state machine (FSM).

There are two reasons that the FSM shown in Figure 46 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 47 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 46. 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.

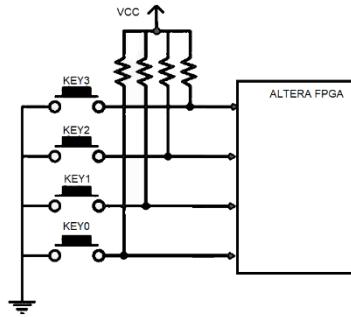


Figure 47: FPGA development board buttons used to operate the stopwatch.

To summarize the operation of the buttons shown in Figure 47. 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.

Module: `stopWatchControl`

The design of the stopwatch controller starts with the state diagram, then on to the control word values for each state, and then finally to the Verilog code needed to implement the state diagram and control word. Each of these steps is outlined below.

State diagram for `stopWatchControl`

The state diagram for the stopwatch is shown in Figure 48. 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.

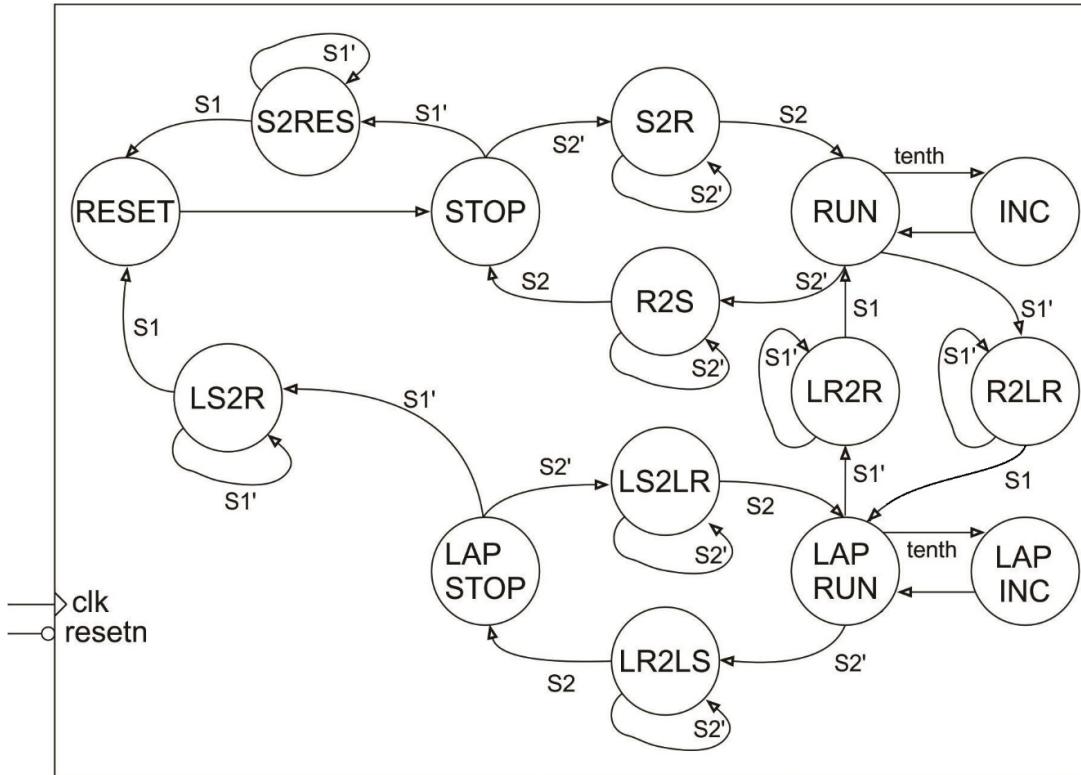


Figure 48: The control unit for the stopwatch needs to account for the time the user holds the button down.

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 49. 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 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.

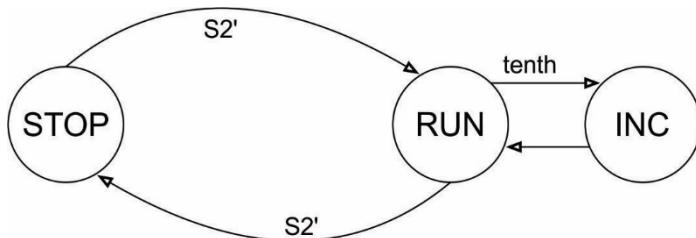


Figure 49: An improperly constructed FSM for the stopwatch.

Control word for stopWatchControl

Before you dive into writing the Verilog code for the control unit, you need to complete Table 41, 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 41 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 41: Control word table for the stopwatch finite state machine shown in Figure 48.

	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					

Verilog code for stopWatchControl

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.

```
TOP_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 associates 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 48 using if/then statements. Let's redraw Figure 48 by focusing on the states connected to the STOP state by transition arcs in Figure 50.

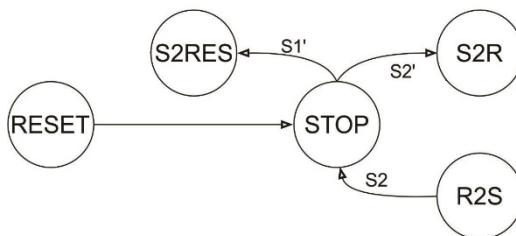


Figure 50: All the states and with a transition arc connected to the STOP state.

The next state logic for the STOP state embodies the 2 transition arcs leaving the STOP state in Figure 50. 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.

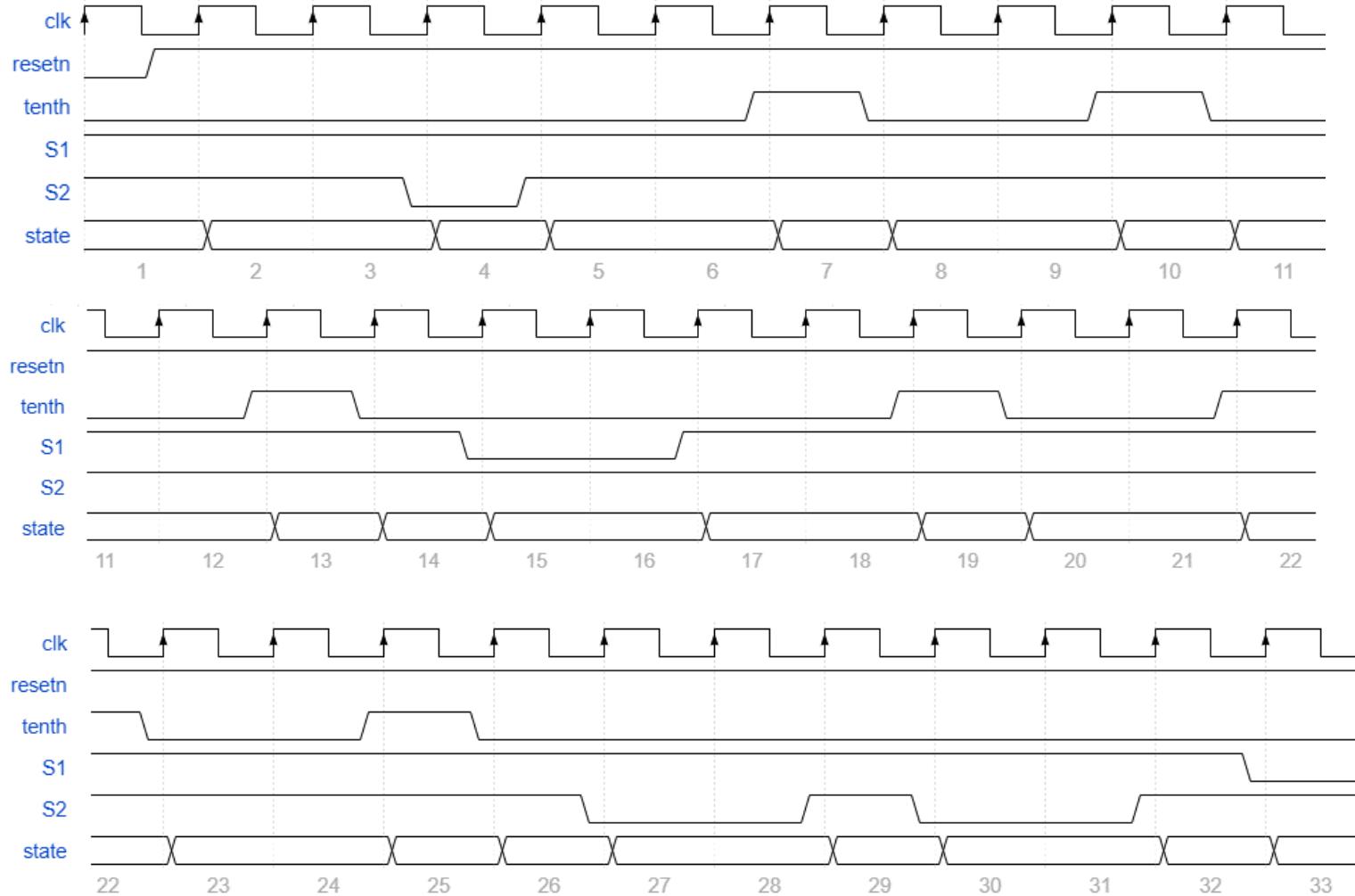
Timing Diagram

This lab will terminate with an extensive simulation of the stopwatch to uncover as many bugs as possible. Errors are much, much easier to find in a simulation. The goal of this simulation is to cover every transition arc in Figure 48 so that you can be sure your code is working correct.

The timing diagrams shown in Figure 51 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 48. 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 48. This goal was not achieved; one transition arc was not taken in the testbench, which one was it?

TIMING DIAGRAM



cv

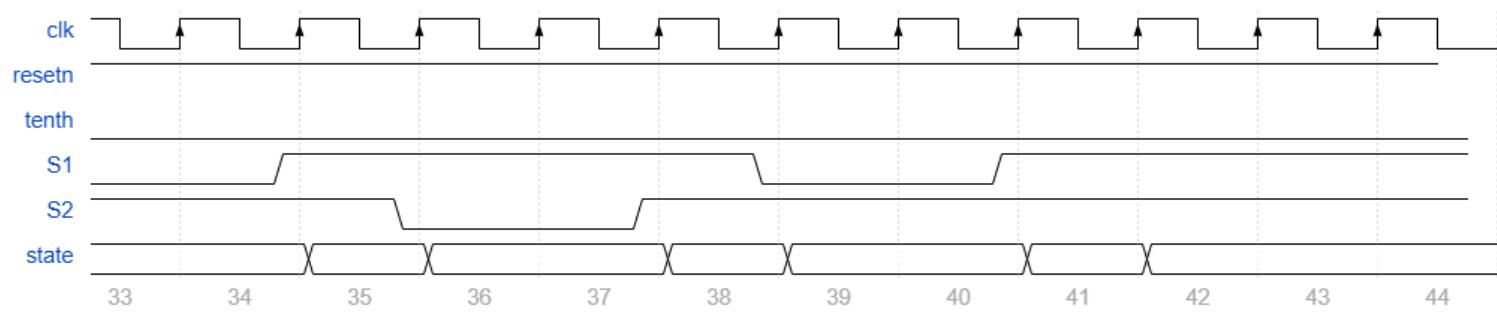


Figure 51: Timing diagram for the testbench for the testbench simulation

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.

signal radix	trace color
clk	default
resetn	default
cw	hex
tenth	default
S2	default
S1	default
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 14 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 14

Listing 14: 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 52 of the state when you run the simulation.

Run the entire simulation and compare the simulation output to Figure 51. Use the results of the simulation to either correct Figure 51 or to correct your controlUnit.v code.

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.



Figure 52: A small segment of the testbench simulation showing how Listing 14 encodes state names.

Control Unit Design

- Completed Table 41.
- Completed Figure 51.
- Verilog code for the control unit in Section . Only include Verilog code for the body of the control unit module (courier 8-point font single spaced) - leave out header comments.

Control Unit Simulation

- Timing diagram of control unit simulation .
- Demonstrate your simulation to a member of the lab team.

Stopwatch Datapath and Control

Outcomes and Objectives

The outcome of this lab is complete the implementation of the stopwatch on the FPGA development board. Through this process you will achieve the following learning objectives.

- Datapath and Control Architecture
- Using a timing for a datapath and control circuit to specify or verify proper operation.
- Synthesizing a module on the FPGA development board

Stopwatch

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 53.

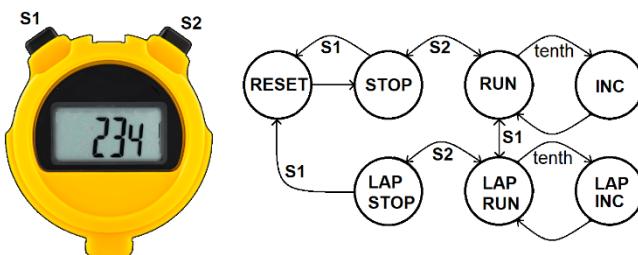


Figure 53: A digital stopwatch gets its input from 2 buttons and displays its output on a 7-segment display. The behavior of the stopwatch can be described by this finite state machine (FSM).

Figure 54 shows how you will implement the idea presented in Figure 53 using 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

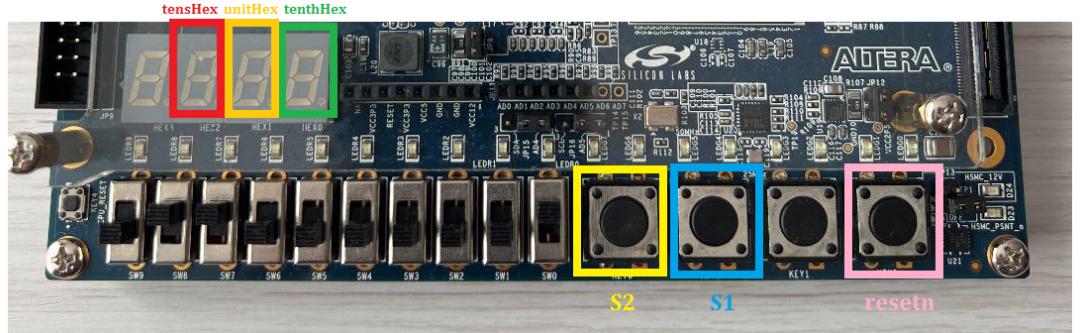


Figure 54: The stopwatch user interface as implemented on the FPGA development board.

development board but does not require any user interaction.

System 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 55, is almost trivial; the Verilog file for the stopwatch contains 2 component instantiations of the datapath and controlUnit.

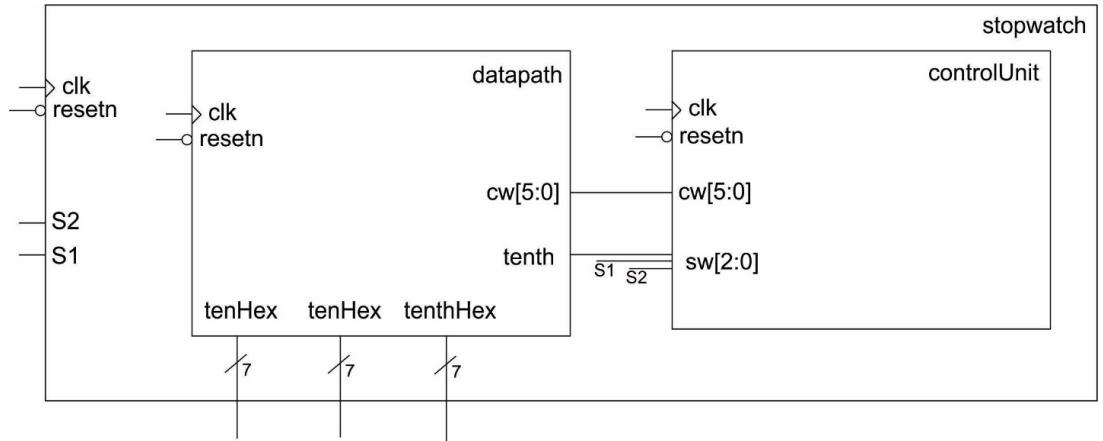


Figure 55: The architecture for the stopwatch consists of a datapath and controlUnit.

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.

Testbench

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 56 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.

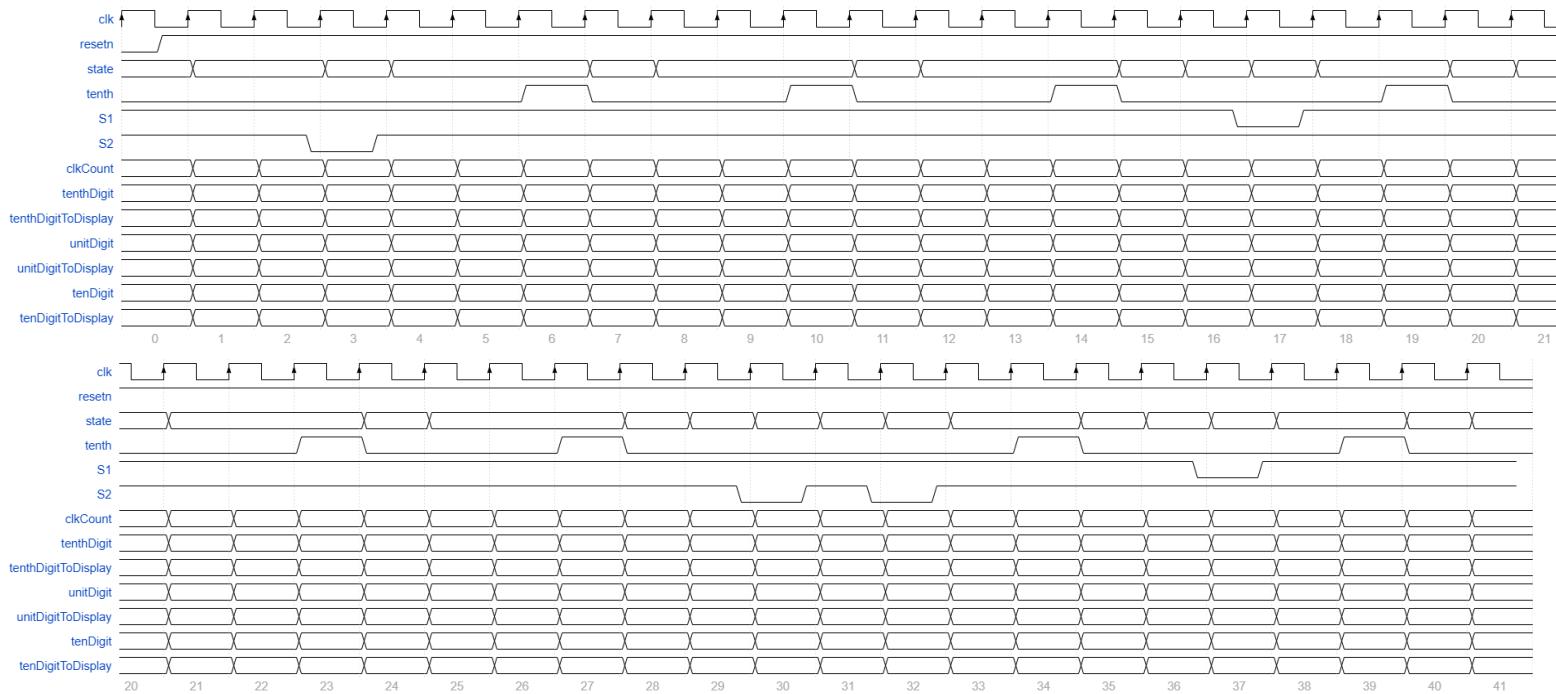


Figure 56: Timing diagram that you will run on the testbench.

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 56. You will probably need to modify the do file to make it work with your design. Address any problems in the stopwatch before proceeding.

Pin-Assignment and 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 15.

Listing 15: 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 15 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 54 and the information in the FPGA development board User Guide to complete the following pin assignment tables.

Table 42: 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]			
seg[4]	V20		

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

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

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.

Testbench

- Completed Figure 56. Please paste the images in landscape format into your solutions
- 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

Pin-Assignment and Synthesis

- Completed pin assignment from Table 42.
- Demonstrate your working stopwatch to a member of the lab team.