

---

# Contents

---

<b>Contents</b>	<b>1</b>
<b>1 Introduction to Verilog</b>	<b>5</b>
1.1 Outcomes and Objective . . . . .	5
1.2 FPGA: Creating a project in Quartus and running a testbench . . . . .	5
1.3 FPGA: Symbolic to Verilog , Timing Diagram, Truth Table . . . . .	9
1.4 FPGA: Verilog to Symbolic, Truth Table, Circuit Diagram . . . . .	9
1.5 FPGA: Circuit Diagram to Verilog, Symbolic, Truth Table . . . . .	9
1.6 Turn in . . . . .	10
<b>2 Hexadecimal to Seven-Segment Converter</b>	<b>13</b>
2.1 Outcomes and Objectives . . . . .	13
2.2 Verilog: Vectors . . . . .	13
2.3 Verilog: Always/Case statements . . . . .	14
2.4 A Multiple Output Function . . . . .	15
2.5 FPGA: Pin-Assignment . . . . .	16
2.6 FPGA: Synthesizing a Verilog Module . . . . .	17
2.7 Hexadecimal to 7-segment Converter . . . . .	18
2.8 Testbench . . . . .	20
2.9 Pin-Assignment and Synthesis . . . . .	21
2.10 Turn in . . . . .	21
<b>3 Rock Paper Scissors</b>	<b>23</b>
3.1 Outcomes and Objectives . . . . .	23
3.2 The Rock Paper Scissors Game . . . . .	23
3.3 System Architecture . . . . .	25
3.4 Module: onesToDense . . . . .	25
3.5 Module: playToSeven . . . . .	27
3.6 Module: winLose . . . . .	28
3.7 Module: rpsGame . . . . .	29
3.8 Testbench . . . . .	30
3.9 Pin-Assignment and Synthesis . . . . .	30
3.10 Turn in . . . . .	31

<b>4 High Low Guessing Game</b>	<b>33</b>
4.1 Outcomes and Objectives . . . . .	33
4.2 The Guessing Game . . . . .	33
4.3 System Architecture . . . . .	34
4.4 Module: 2:1 Mux . . . . .	34
4.5 Module: Compare . . . . .	36
4.6 Module: hexToSevenSeg . . . . .	37
4.7 Module: 2:4 Decoder . . . . .	37
4.8 Module: hiLowWin . . . . .	38
4.9 Module: LFSR . . . . .	38
4.10 Module: hiLow . . . . .	40
4.11 Testbench . . . . .	41
4.12 Pin-Assignment and Synthesis . . . . .	41
4.13 Turn in . . . . .	42
4.14 Debugging Tips . . . . .	43
<b>5 High Low Guessing Game With Hints</b>	<b>45</b>
5.1 Outcomes and Objectives . . . . .	45
5.2 The Guessing Game with Hints . . . . .	45
5.3 System Architecture . . . . .	47
5.4 Module: 2:1 Mux . . . . .	48
5.5 Module: Compare . . . . .	48
5.6 Module: Add/Sub . . . . .	48
5.7 Logic: hotWarmCold . . . . .	49
5.8 Module: hiLow . . . . .	51
5.9 Testbench . . . . .	51
5.10 Pin-Assignment and Synthesis . . . . .	52
5.11 Turn in . . . . .	53
5.12 Debugging Tips . . . . .	54
<b>6 Calculator With Friendly Output</b>	<b>57</b>
6.1 Outcomes and Objectives . . . . .	57
6.2 Calculator with Friendly Output . . . . .	57
6.3 System Architecture . . . . .	58
6.4 Module: sigUnsign . . . . .	59
6.5 Testbench . . . . .	64
6.6 Pin-Assignment and Synthesis . . . . .	65
6.7 Turn in . . . . .	66
6.8 Bonus: Ovf Logic . . . . .	67

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 Elements
  - Characteristics
  - Timing
  - Asynchronous set reset
- Sequential Logic Building Blocks
  - Analysis, design and use of a register in a digital design
  - Shift Register
  - Counter
  - Static RAM
  - Register Transfer
- Finite State Machines
  - Hardware organization of a finite state machine
  - State diagram for a finite state machine
  - One's Hot Encoding
  - Design
  - Timing
- Datapath and Control
  - Datapath and Control Architecture
  - Algorithmic Language
  - Design
  - Timing

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

---

## Laboratory 1

---

# Introduction to Verilog

---

### 1.1 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.

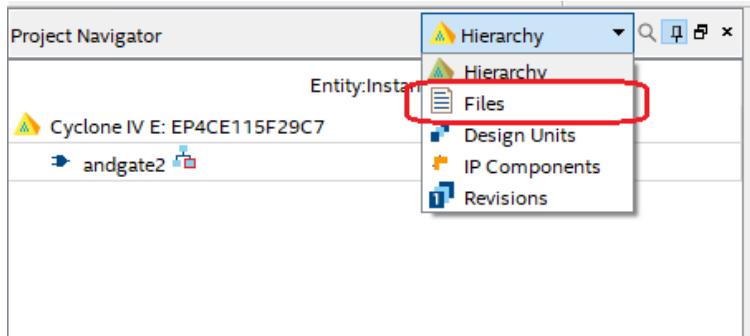
- Elementary Logical Functions
- Conversion between two different representations of a logic function
- Writing concurrent signal assignment statements for a logic function
- Writing a Verilog statement using primitive logic operations
- Creating a simulation timing diagram for a module

### 1.2 FPGA: Creating a project in Quartus and running a testbench

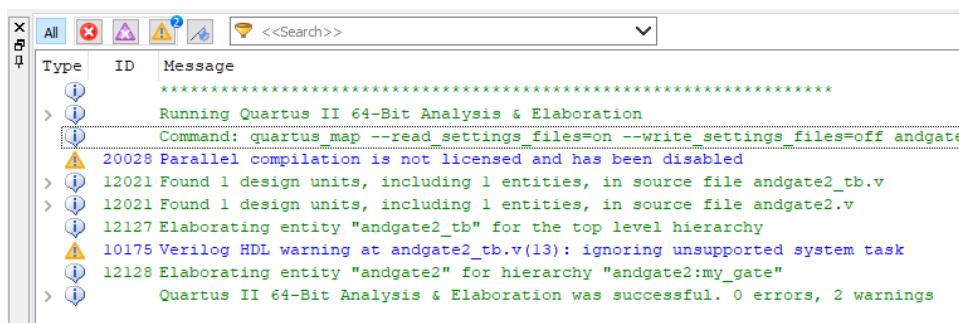
In this section you will apply inputs to a 2-input AND gate and observing the output on a timing diagram. Since all this activity will take place in the memory of a computer and not on actual hardware, it is called a simulation. To start this process, you will first have to create a project and add files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

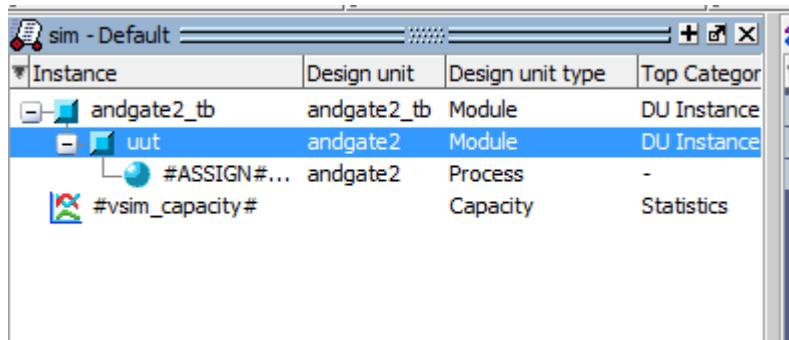
- in step 1,
- Select the andgate2 folder, click Select Folder,
  - In the “What is the name of this project” field type *andgate2*
  - click *Next*.
- In the **Project Type** page of the New Project Wizard pop-up:
    - Select the *Empty project* radio button,
    - click *Next*.
  - In the **Add Files** page of the New Project Wizard pop-up:
    - Click the ... button to the right of File name,
    - In the Select File pop-up, navigate to, and select, *andgate2.v* and *andgate2\_tb.v*, click Open,
    - The file should appear in the window below,
    - Click *Next*
  - In the **Family & Device Settings** page of the New Project Wizard pop-up:
    - Device family, Family: Cyclone V
    - Package: FBGA
    - Pin Count: 672
    - Speed Grade: 7\_H6
    - Select Specific device selected in ‘Available devices’ list
    - From the list of available devices, select: 5CGXFC5C6F27C7
    - Click Next
  - In the **EDA Tool Settings** page of the New Project Wizard pop-up:
    - In the Simulation row
      - Tool Name column: ModelSim-Altera
      - Formats column: Verilog HDL
    - Leave other defaults alone
    - Click Next
  - In the **Summary** page of the New Project Wizard pop-up:
    - Review information,
    - Click Finish.
  - Back in the main Quartus II window, Click *Tools -> Options...*
  - In the Options pop-up:
    - Select *EDA Tool Options* from the Category menu,
    - 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.
    - Click Ok.
  - Click on the Files tab in the *Project Navigator* pane.



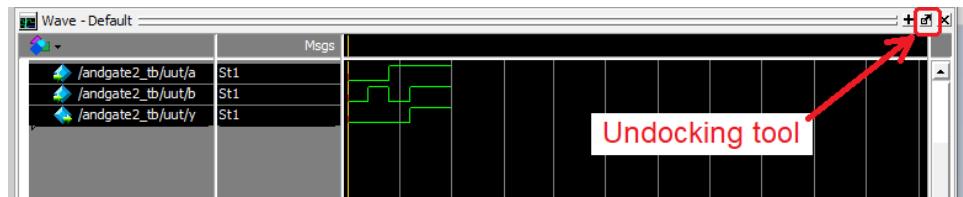
13. Right click on *andgate2\_tb* in the *Project Navigator* pane and select Set as Top-Level entity.
14. Double click on *andgate2*.
15. If you added the Verilog file in the correct directory and included it in the project, a Verilog file should pop up on the right.
16. In the main Quartus II window, 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 *andgate2\_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.



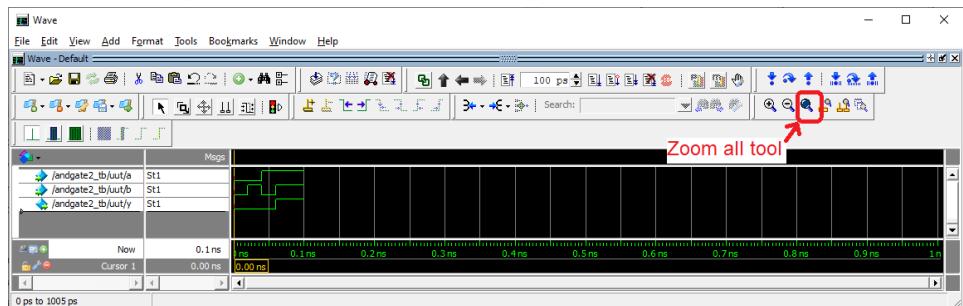
18. In the main Quartus II window, click *Tools -> Run Simulation Tool -> RTL Simulation*. The ModelSim program will launch. This may take a few moments, be patient. If you get a pop-up Nativelink Error window, then go back, check and fix the directory in step 11.
19. In ModelSim, click Simulate -> Start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

### 1.3 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.

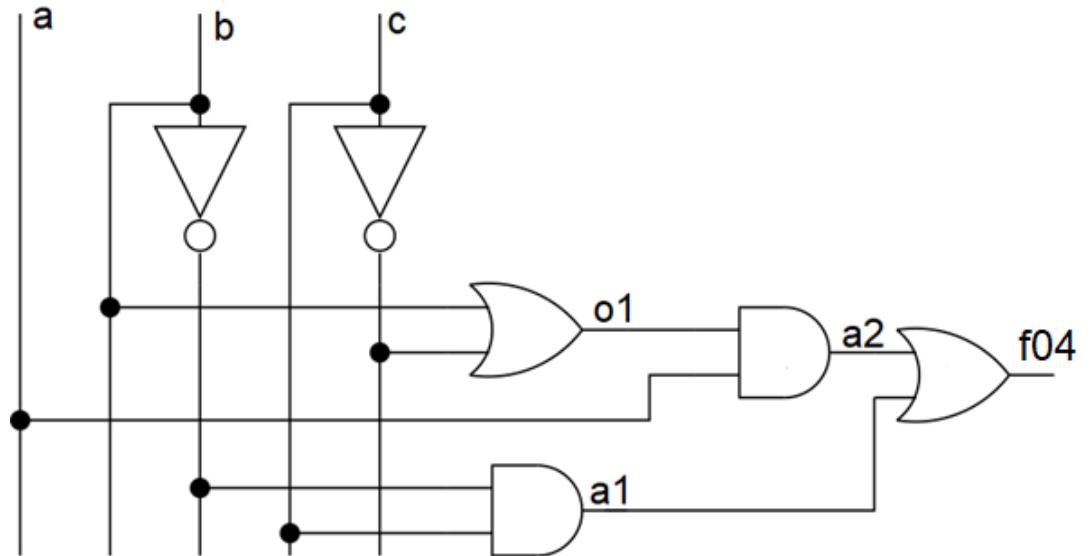
### 1.4 FPGA: Verilog to Symbolic, Truth Table, Circuit Diagram

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*.

### 1.5 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.

## 1.6 Turn in

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

### 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  $f04$



---

## Laboratory 2

# Hexadecimal to Seven-Segment Converter

---

### 2.1 Outcomes and Objectives

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

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

### 2.2 Verilog: Vectors

A vector is a collection of bits that are realted to one another in some way. For example, the individual bits of a 4-bit number could be represented as a vector. There are three things that you will need to know about vectors in order to complete today's lab (and future labs), combining bits into a vector, defining a vector, and accessing the bits of a vector. These operations are illustrated in Figure 2.1.

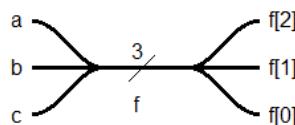


Figure 2.1: An illustration showing three bits combined into a vector,  $f$ , and then accessing the individual bits of  $f$ .

The operations shown in Figure 2.1 are similar to the operations in Listing 2.1. In this

module, the line of code `assign f = {a,b,c};` combines the individual signals `a`, `b` and `c` into a 3-bit vector `f`. The left-most signal in the parenthesis list, `a`, becomes the MSB of the vector and the right-most signal, `c`, becomes LSB. Combining signals is more commonly called concatenation. You can concatenate any arrangement of signals as long as the number of bits comes out the same as the signal on the left-hand-side of the `=` sign.

Listing 2.1: Verilog code which illustrates vector manipulations and declarations.

```
module unimportantModuleName();

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

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

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

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

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

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

### 2.3 Verilog: Always/Case statements

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

Listing 2.2: A 3-input, 2-output function realized with an always statement. Can you figure out how the output was computed?

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

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

```
3'b111: f = 3'b11;
endcase
```

For the time being, we will trust that the statement always `@(*)` allows the code between `case` and `endcase` to run continuously and concurrent with any other statements in the module. Yes, this means that all the code between `case` and `endcase` acts like a single assign statement. A case statement uses the argument to case, `x` as a selector for one of the rows below. Every possible value of `x` must be present and when that value matches `x`, the action to the right of the colon is performed. When we use a case statement as shown in Listing 2.2 you must make the output type `reg`.

All signals are either `wire` or `reg` type. A wire is a signal that has a value provided to it by some active element. This active element might be a gate or the output of a module. If a signal does not have an explicit gate or module driving its value, it needs to be typed `reg`.

## 2.4 A Multiple Output Function

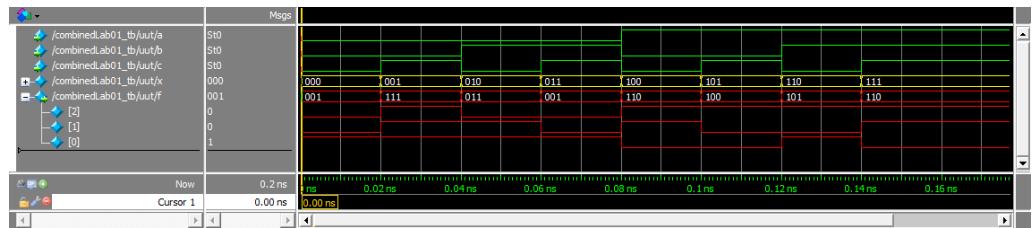
Let's explore vectors and the always statement by combining the three functions created in last weeks assignment into one function.

1. Go back to your Lab 01 solutions and extract the truth tables for function `f04`, `f03`, and `f02`. Put these values into the truth table shown in Table 2.1.

Table 2.1: The Truth Table for the combinedLab01 function. This function has a 3-bit input and 3-bits output.

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

1. Create a new project folder within your `lab2` directory called `combinedLab01`.
2. Download `combinedLab01.v` and `combinedLab01_tb.v` from Canvas to the project directory.
3. Create a project for these two files using the steps from last week's lab.
4. Modify `combinedLab01.v` so that `combinedLab01` outputs the values given in Table 2.1.
5. Modify `combinedLab01_tb.v` so that `combinedLab01` is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0 and going to 1,1,1.
6. Perform simulation using with the test bench using the steps from last week's lab.
7. Capture the output waveform from Simulink. It should look something like the following.



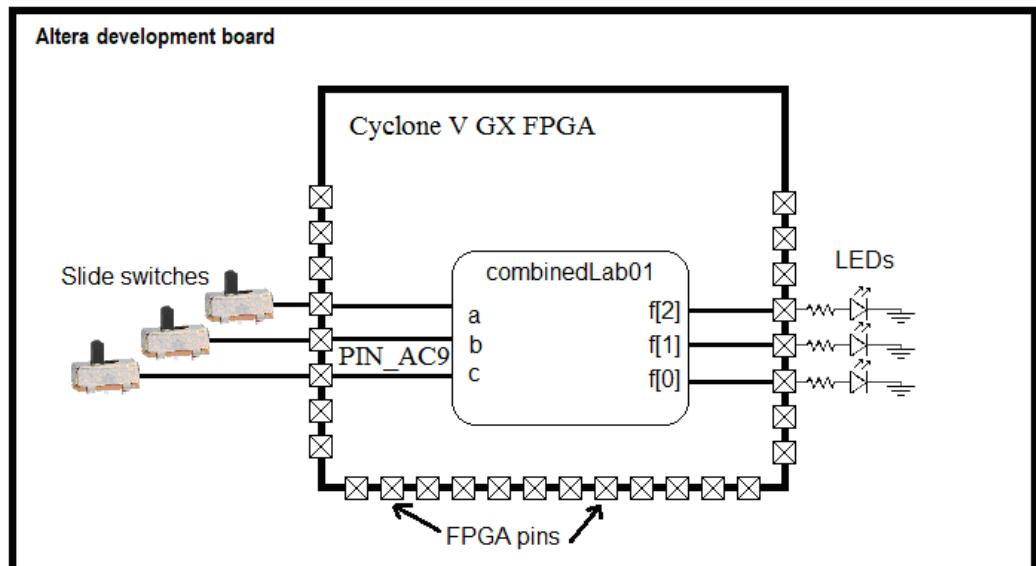
8. From the information in the timing diagram, produce a truth table. Compare the truth table generated from the data in the timing diagram to that you generated in Table 2.1.

## 2.5 FPGA: Pin-Assignment

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

The Figure 2.2 shows a Verilog module called *combinedLab01* synthesized and downloaded into an Altera FPGA on the development board. Note that ports a, b and c are connected to FPGA pins that are driven by slide switches. Ports f[2], f[1] and f[0] are connected to FPGA pins that drive LEDs. In this way, a user can provide input to the *combinedLab01* module by moving the slide switches and observe the circuit's output on the LEDs.

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



The development board contains an Altera Cyclone V GX FPGA. This FPGA has many pins and they are identified by a lettered group and number. For example, in Figure 2.2 port c of the combinedLab01 module is mapped to pin AC9.

You will need to be able to figure out the remaining pin assignments on your own. To do this open up the C5G User Manual posted on the class Canvas page. Start with Figure 3-9 on page 30 which shows that the slide switches in one of two positions (up or down). In the up position, they assert a logic 1 and down they assert a logic 0. On the next page, 31 of the C5G User Manual look at Table 3-3. This table defines the relationship between the different slide switches and the FPGA pins each is connected to. For example, slide switch SW[0] is connected to PIN\_AC9.

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

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

Use the information to complete the pin assignment in Table 2.2. We will use this assignment in the next section.

Table 2.2: Pin Assignment Table for combinedLab01.

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

## 2.6 FPGA: Synthesizing a Verilog Module

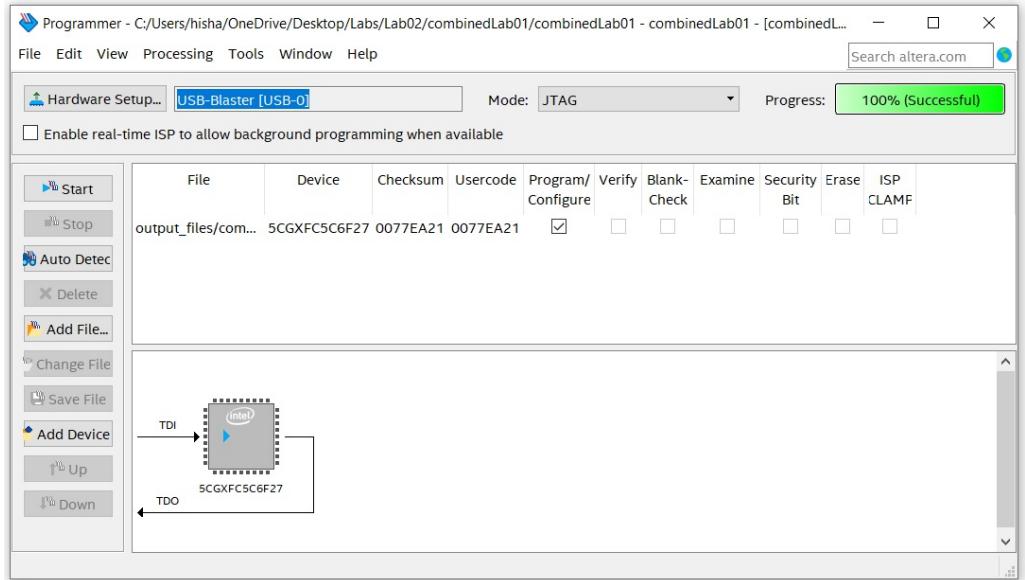
It's time to complete the synthesis process by downloading the combinedLab01 Verilog file along with its pin-assignment to the FPGA. Work through the follow these steps to accomplish this.

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

PLL/DLL Output											
Named:	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)		
	<<new node>>										

6. Double click in the Location cell for row c
7. Scroll down the list of pins to PIN\_AC9
8. Complete the pin assignment for the other 5 inputs and outputs using the information contained in pin assignment table completed earlier.

9. Double check your pin assignments.
10. File -> Close. Note closing your file incorporates this assignment into the project.
11. Back in the Quartus window, Processing -> Start Compilation <Ctrl-L>
12. Tools -> Programmer
13. In the Programmer pop-up window click Add File...
14. In the Select Programming File pop-up, navigate to your project directory, then into the output files folder, the select combinedLab01.sof, click Open. You should see something like the following.



15. Connect the Altera Cyclone V GX FPGA to your computer through the USB port, connect the power supply, and push the red power-on button. Try not to be annoyed by the infernal blinking LEDs.
16. In the Programmer pop-up
  - a. Click Hardware Setup....
  - b. In the Hardware Setup select USB-Blaster [USB=0] from the Currently selected hardware pull-down
  - c. Click Close
17. Back in the Programmer window, the box next to Hardware Setup... should reflect your choice. Click Start,
18. The Development board should stop its infernal blinking and run your program. You may notice that the unused LEDs are dimly illuminated.
19. Move the slide switches up and down to verify that the input/output matches the values in Table 2.1. Use white silk screen printing on the development board to locate slide switches and LEDs you assigned in your pin-assignment.

## 2.7 Hexadecimal to 7-segment Converter

While working on the previous problem, you probably noticed that the development Board has four 7-segment display. These figure 8 shaped blocks above the slide switches are the devices which light up numbers on some cash registers. We will be using these 7-segment displays for a variety of purposes during the term, so it would be a good idea.

The hexadecimal-to-seven-segment-decoder is a combinational circuit that converts a hexadecimal number to an appropriate code that drives a 7-segment display the corresponding value. **BEWARE**, the LEDs in the 7-segment displays on the Development Board are active low, asserting a logic 0 on the pin attached to a segment will cause that segment to illuminate.

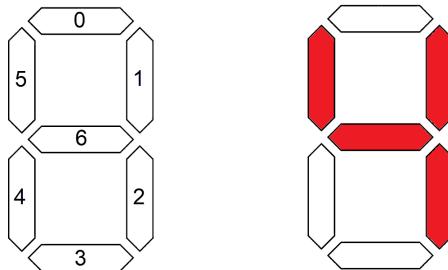


Figure 2.3: Left, the proper numbering of the segments. Right, illuminating segments to form the number 4.

The pattern of segments to be illuminated for each digit is shown in Figure 2.3. For example, to display '4' output would be:

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

Figure 2.4 shows the proper formatting for all the values between 0 – f.

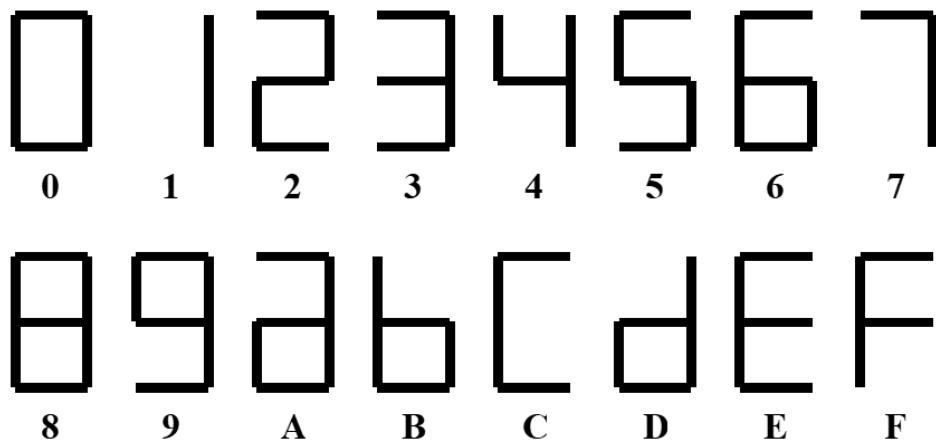


Figure 2.4: The proper arrangement of LEDs to form hexadecimal characters.

Use this information to complete the Table 2.3 to illuminate the active low led segments to generate proper hexadecimal characters.

Table 2.3: Truth table for the hexToSevenSeg component.

x	seg[6]	seg[5]	seg[4]	seg[3]	seg[2]	seg[1]	seg[0]
0000							
0001							
0010							
0011							
0100	0	0	1	1	0	0	1
0101							
0110							
0111							
1000							
1001							
1010							
1011							
1100							
1101							
1110							
1111							

Now that you have a complete description of the input/output behavior of the hexadecimal to seven segment converter. its time to write the Verilog code. You will capture the behavior in Table 2.3 using an always/case statement. Work through the following steps to complete this task.

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

## 2.8 Testbench

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

1. Modify *hexToSevenSeg\_tb.v* so that *hexToSevenSeg* is run through every combination of inputs. Assert the inputs in increasing binary numbering order starting from 0,0,0,0 and going to 1,1,1,1.
2. Perform simulation using this test bench as described in previous steps. You will need to “run 100” several times to go through all the inputs.
3. Save this waveform as an image as done in the previous section. If the waveform is missing, you can add it back in using View -> Waveform.
4. Compare From the information in the timing diagram, produce a truth table for in Table 2.3. Fix any errors in the always/case statement before proceeding to synthesis in the next step. *hexToSevenSeg*.

## 2.9 Pin-Assignment and Synthesis

Before you can download your design to the FPGA, you need to assign the input and outputs of the hexToSevenSeg module to FPGA pins. Figure 2.5 shows the slide switches and 7-segment display that will use.

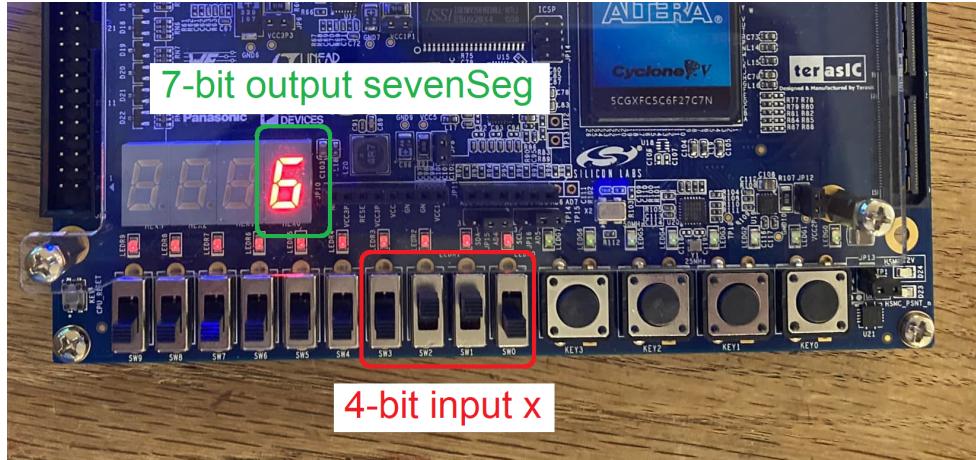


Figure 2.5: The input set to 4'b0110 displaying a 6 on the 7-segment display.

Use the C5G User manual to complete the pin-assignment in Table 2.4.

Table 2.4: Pin-assignment tables for the hexToSevenSeg module.

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

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

Use the instructions in Section 2.6 to combine the pin assignment with your hexToSevenSeg module. Synthesize your design, bask in the glow of another success as you demonstrate your circuit's functionality to a member of the lab team.

## 2.10 Turn in

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

**Combine lab 1**

- Truth Table for combinedLab01 function in Table 2.1
- [Link](#) Timing diagram for combinedLab01 function
- Pin assignment for combinedLab01 in Table 2.2

**Hexadecimal to 7-segment**

- Truth Table for hexToSevenSeg function in Table 2.3
- [Link](#) Verilog code for hexToSevenSeg function – just the always/case statement
- [Link](#) Timing diagram for hexToSevenSeg function
- Pin assignment tables for hexToSevenSeg in Tables 2.4
- Demonstrate working hexadecimal to seven segment module to a member of the lab team.

---

## Laboratory 3

# Rock Paper Scissors

---

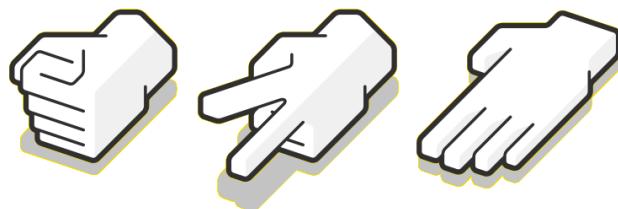
### 3.1 Outcomes and Objectives

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

### 3.2 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 3.1. Each player will have access to three slide switches and one 7-segment display. Each of the three switches represents one of the three plays and the 7-segment will display the throw when the Play button is pressed. The Win/Lose 7-segment display will show “1” when player 1 wins, “2” when player 2 wins, and “d” when the game is a draw (a tie).

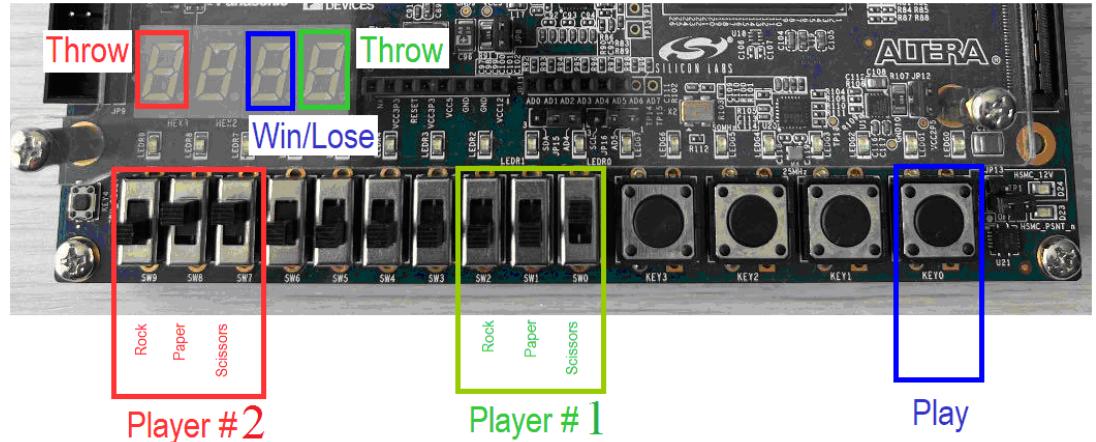


Figure 3.1: The input and output you should use to realize your digital system.

A player will move one of the three slide-switches into the up position to indicate their play. Moving more than one slide-switch, or no slide switch into the up position is in an invalid play. An invalid play always loses to a valid play. If each player throws an invalid play, the game is a draw.

While each player is making their choice of play, their Throw 7-segment display will reflect their choice as shown in Figure 3.2. These patterns are supposed to vaguely resemble the objects.

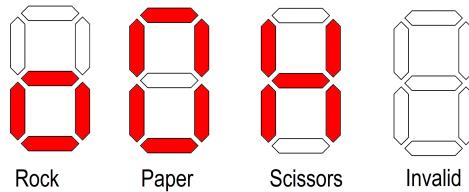


Figure 3.2: Illuminated patters for the different plays.

When the Throw button is pressed, the Win/Lose 7-segment will display “1”, “2”, or “d” depending on the outcome of the game as shown in Table 3.1. When the Throw button is un-pressed, the Win/Lose 7-segment display is blank.

Table 3.1: The output for every combination of player 1 (P1) and player 2 (P2) throws.

P1 \ P2	Rock	Paper	Scissors	Invalid
Rock	d	2	1	1
Paper	1	d	2	1
Scissors	2	1	d	1
Invalid	2	2	2	d

### 3.3 System Architecture

The system architecture shown in Figure 3.3 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 3.1. Each soft-square (a square with rounded corners) is a Verilog module. Names inside soft-squares, that are adjacent to lines outside the soft-square, are the port names for that module. The instance name and module name of a module are separated by a “:” and usually located along the top edge of the soft-square. Red soft-squares are associated with player 1 and green soft-squares associated with player 2. The names on lines inside the rpsGame soft-square are the signals names you should use in the rpsGame module to connect the 5 modules together. Lines that are slashed with a number denote bit vectors.

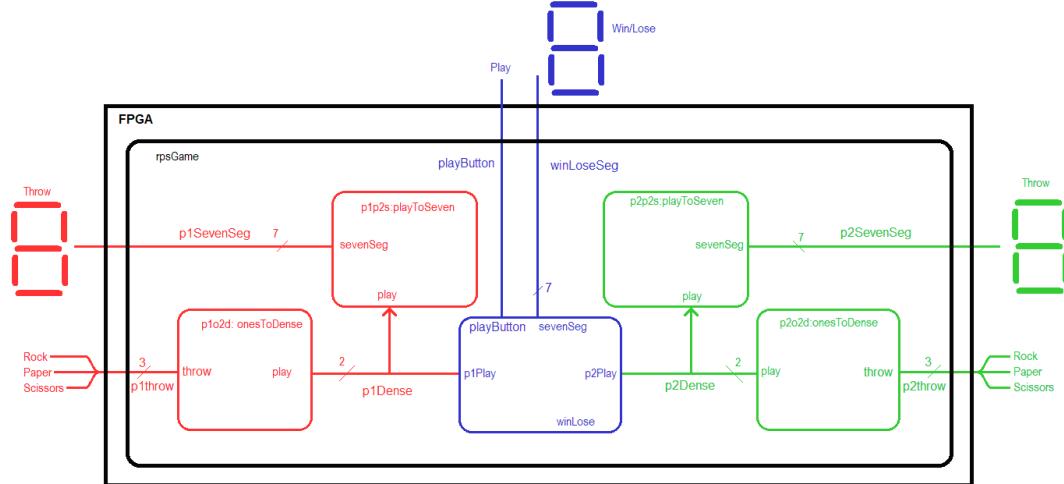


Figure 3.3: System architecture for the rock, paper, scissors system.

### 3.4 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 3.2 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 3.2: 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 3.2 truth table while ignoring the LSB. Then proceed to write the canonical SOP expression for the LSB of the play output, play[0], in the Table 3.2 truth table while ignoring the MSB.

$$\begin{aligned} \text{play}[1] = \\ \text{play}[0] = \end{aligned}$$

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 3.2. This will require a wire declaration and two assign statements to give each variable 1-bit from the input vector.

- Use assign statements to realize the AND, OR, NOT logic derived for your canonical SOP expressions.
- Use function04 from lab 1 as a starting point for this module.

### 3.5 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 3.4.

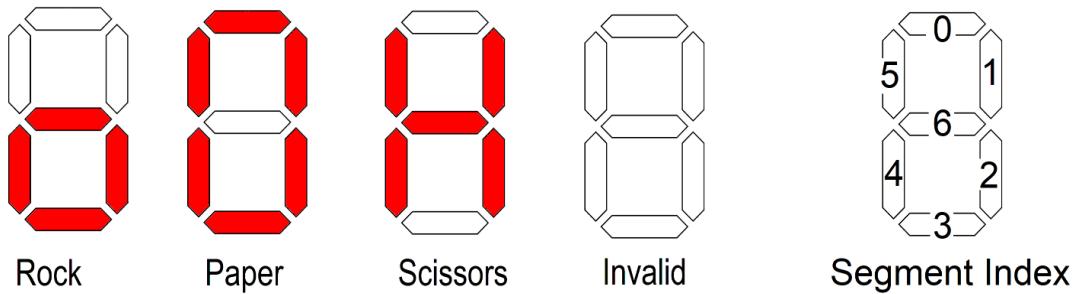


Figure 3.4: 7-segment display patterns for the different throws along with the 7-segment display segment indices.

Use the information in Figure 3.4 to determine the bit patterns needed to generate the four different symbols in Table 3.3. Remember that the LEDs in the segments are active low, meaning that a logic 0 output illuminates an LED segment. In Table 3.3 put a 0 or 1 in each of the numbered column so that each row produces the patterns for its throw. Remember that 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 3.3: The 7-segment display LEDs to produce the throw patterns.

pPlay	6	5	4	3	2	1	0	sevenSeg	Note
00									Rock
01									Paper
10									Scissors
11								7'b1111111	Invalid

Incorporate the following into your playToSeven Verilog module:

- Use the module declaration: `module playToSeven (pPlay, sevenSeg);`
- Use a vector for the input pPlay and a vector for the sevenSeg output.
- Make the input port type “wire”. Make the output port type “reg”.
- Use a case statement, embedded in an always statement to realize this module. Enumerate all combinations of the input; do not use a default case
- Use the information in Table 3.3 to assign values to the output.

- Put comments at the end of each case row describing, in words, what the output should look like on the 7-segment display.
- Use the hex2Seven module from lab 2 as the starting point for this module.

### 3.6 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 3.5 as a reminder.

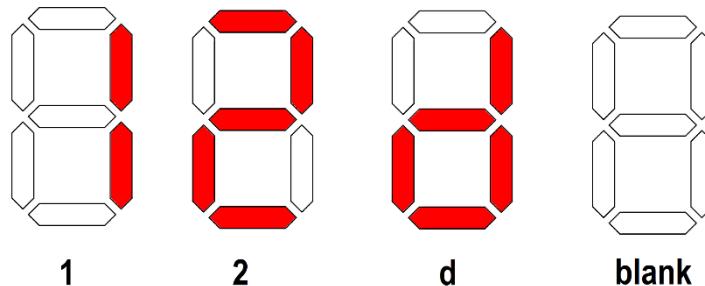


Figure 3.5: The illuminated patterns displayed on the win/lose 7-segment display.

You will use a case statement, based on the information in Table 3.4, to realize this function. In the sevenSeg column put the 7-bit code needed to illuminate the winLose 7-segment display to indicate the outcome of the game. In the Note column put either “1”, “2”, “Draw”, or “Blank” depending on the outcome of the game and button press. Use the bit order given in Figure 3.4.

Table 3.4: Abbreviated truth table for the winLose module.

button	p1Play	p2Play	sevenSeg	Note
0				
0				
0				
0				
0				
0				
0				
0	10 (scissors)	00 (rock)	7'b0100100	2
0				
0				
0				
0				

button	p1Play	p2Play	sevenSeg	Note
0				
0				
0				
1	xx (don't care)	xx (don't care)	7'b1111111	Blank

Incorporate the following into your winLose Verilog module: Use the module and port names given in Figure 3.3.

- Use the module declaration: `module winLose(p1Play, p2Play, playButton, sevenSeg);`
- Use vectors for the p1Play and p2Play inputs.
- Use a vector for the sevenSeg output.
- Make the input port types “wire”. Make the output port types “reg”.
- You will use a case statement, embedded in an always statement to realize this module.
  - Use brackets to make the vector for the case statement. For example, if button was a 1-bit signal and play was a 2-bit signal, then

```

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

```

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

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

// P1: Rock P2:Rock Draw

- Use the hex2Seven module from lab 2 as the starting point for this module.

### 3.7 Module: rpsGame

The rpsGame module, “glues” together the modules shown in Figure 3.3 and serves as the top-level entity. The Verilog code for this module consists of 5 instantiation statements; one of them is given as the last bullet point item in the list below. For this module, I want you to:

- Use the module declaration:

```
module rpsGame(p1Throw, p1SevenSeg, p2Throw, p2SevenSeg, playButton, winLoseSeg);
```

- Make the p1Throw and p2Throw inputs vectors with the MSB coming from the rock slide-switch input and scissors slide-switch as the LSB. You will need to keep this consistent with the pin assignment that you will complete next.
- The playButton input is not a vector.
- Use a vector for the winLoseSeg output.

- Make the input and output port types “wire”.
- You need to create 2 internal vectors. Look carefully at Figure 3.3 and find wires that begin and end inside the rpsGame module. These are the vectors.
- Name the module instances using the names provided in Figure 3.3.
- When you instantiate a module
  - The first term is the name of the module you are instantiating
  - The second term is the instance name of the module
  - The remaining term is the parenthesis list of signal in and out of the module. The order of the signals in the instantiation must be the same as those in the module declaration. Pay special attention to this!
  - For example, in my program I had an instantiation that looked like:  
`onesToDense p1o2d(p1Throw, p1Dense);`

### 3.8 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.

### 3.9 Pin-Assignment and Synthesis

Use the image of the FPGA Development Board in Figure 3.1 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 3.5: Pin assignment tables for the Rock Paper Scissor game.

Segment	Player 1 Throw	Player 2 Throw	Win/Lose
seg[6]	PIN_Y18		
seg[5]		PIN_AC23	
seg[4]			
seg[3]			
seg[2]			
seg[1]			
seg[0]			PIN_AA18

	Player 1 Slide Switch	Player 2 Slide Switch
slide[2]		
slide[1]		
slide[0]	PIN_AC9	

Play Button	Key[0]	
-------------	--------	--

Note, each push-button provides a high logic level when it is not pressed, and provides a low logic level when pressed.

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.

### 3.10 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 3.2 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 3.3.
- [link](#) Verilog code for the module (courier 8-point font single spaced), leave out header comments.

#### **winLose Module**

- Complete Table 3.4 truth table for winLose module.
- [link](#) Verilog code for the module (courier 8-point font single spaced), leave out header comments.

#### **rpsGame Module**

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

#### **Pin Assignment**

- Completed pin assignment for 7-segment, slide switches and button in Table 3.5.
- Demonstrate your working design to a lab team member.



---

## Laboratory 4

# High Low Guessing Game

---

### 4.1 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

### 4.2 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*.

Your goal in this lab is to create a digital version of the guessing game using the development board using the inputs and outputs shown in Figure 4.1. In this case, the FPGA will play the role of secret keeper. You will enter a seed value using the **seed** slide switches. The seed value will be “randomized” into a 4-bit *secret number* using a linear feedback shift generator (more about this later). Pressing the **rand** button reveals the 4-bit *secret number* as a 1-digit hexadecimal value on the **randValue** 7-segment displays. Obviously, the guesser should not press the **rand** button during regular game play.

The player will make their guess about the secret number on the **guess** slide switches. This *guess* is compared to the *secret number* and the outcome is displayed on the **game** 7-segment display when the **hiLow** button is pressed. The **game** 7-segment display will show:

- ‘H’ when *guess*  $\neq$  *secret number*

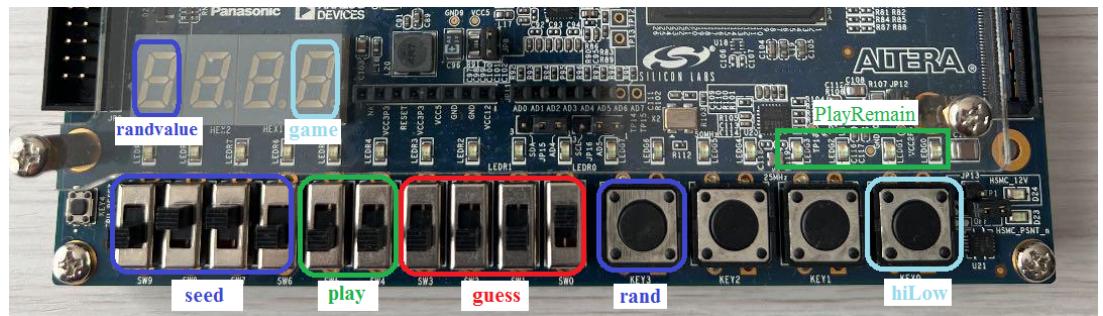


Figure 4.1: The input and output you should use to realize the High/Low Guessing game.

- ‘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.

### 4.3 System Architecture

Use the system architecture shown in Figure 4.2 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.

### 4.4 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 4.3 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 4.2 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 4.1.

Listing 4.1: Top, module definition for a 2:1 mux. Bottom, module instantiation of a 2:1 mux in Figure 4.2.

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

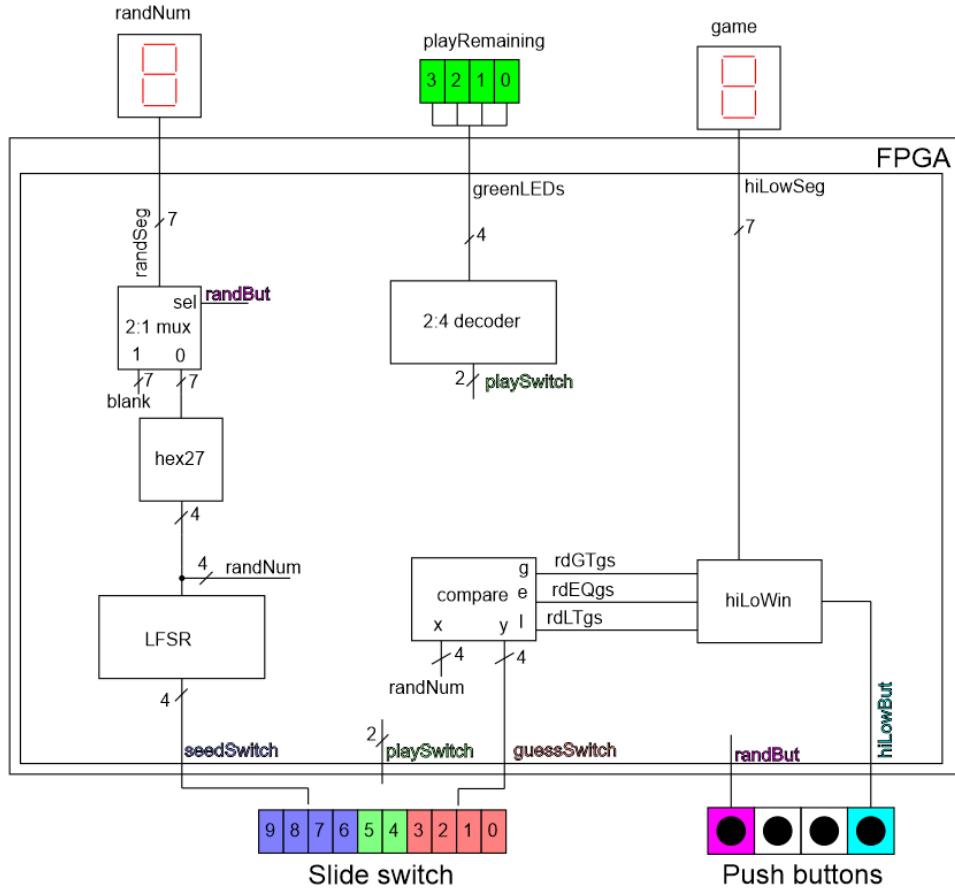


Figure 4.2: System architecture for the guessing game.

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

The signal width, `N`, shown in Figure 4.3 is a placeholder for an integer value that describes the width of the `y1`, `y0`, `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.

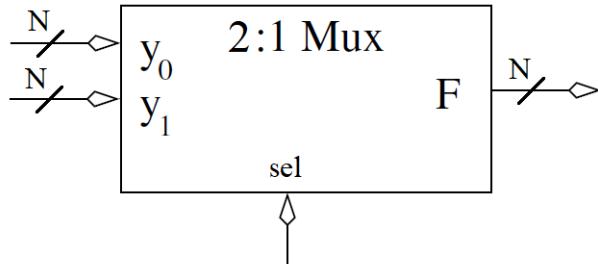


Figure 4.3: The schematic representation of a 2:1 mux.

Port Connectivity Checks: "genericMux2x1:muxMsbHex"			
	Port	Type	Severity
1	y1	Input	Info
2	f	Output	Warning
3	y1	Input	Warning
4	y0	Input	Warning

Details

1. Port y1 is an input and is connected to VCC. Severity: Info.

2. Port f is an output and is wider than the port expression (7 bits) it drives; bit(s) "f[7..?]" have no fanouts. Severity: Warning.

3. Port y1 is an input and is smaller than the input port (8 bits) it drives. Extra input bit(s) "y1[7..?]" will be connected to GND. Severity: Warning.

4. Port y0 is an input and is smaller than the input port (8 bits) it drives. Extra input bit(s) "y0[7..?]" will be connected to GND. Severity: Warning.

Figure 4.4: Forgetting the generic specifier in a 2:1 Mux will generate this report.

## 4.5 Module: Compare

A N-bit comparator is a basic building block in many digital systems. The N-bit comparator shown in Figure 4.5 checks the relative magnitude of the two N-bit inputs  $X$  and  $Y$  and sets one of the three outputs equal to 1, one's-hot output, depending on their relation to each other.

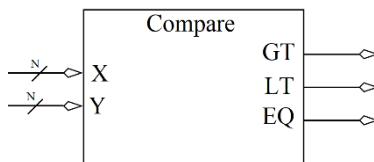


Figure 4.5: A schematic representation of a N-bit comparator.

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

- $\text{GT} = 1$  when  $X > Y$  else  $\text{GT} = 0$
- $\text{EQ} = 1$  when  $X == Y$  else  $\text{EQ} = 0$
- $\text{LT} = 1$  when  $X < Y$  else  $\text{LT} = 0$

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 4.2: Top, the module definition for the comparator. Bottom, module instantiation of a comparator in Figure 4.5. 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 comparator 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.

## 4.6 Module: hexToSevenSeg

You should use the hexToSevenSeg module you developed in a previous lab. Note, the name of this module was shortened in Figure 4.2 to hex27 in order to save space and make the schematic more readable.

## 4.7 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.

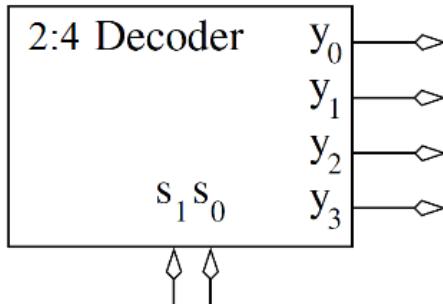


Figure 4.6: A 2:4 decoder.

The first few rows for the truth table for the 2:4 decoder are shown in Table 4.1.

Table 4.1: Partial truth table for the 2:4 decoder.

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

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

You should implement the 2:4 decoder in the hiLow module using an always/case statement similar to the one used to implement your hexToSevenSeg. You should put this Verilog code

in the hiLow module as a (large) concurrent statement. **This means that you should not have a separate Verilog file for the 2:4 decoder.** Remember that the output type from an always/case statement must have the “reg” qualifier, not “wire”.

#### 4.8 Module: hiLowWin

The hiLowWin functionality converts the output from the comparator into the illuminated patterns shown in Figure 4.7 when the **hiLow** button is pressed. The “I” from “wIn” is needed because you cannot make a “W” on a 7-segment display.

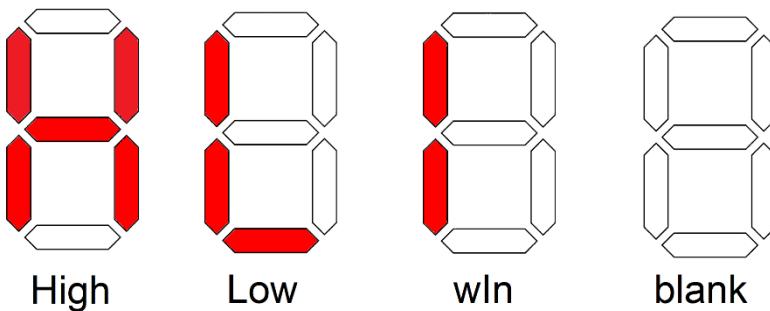


Figure 4.7: The illuminated patterns to inform the guesser about the magnitude of their guess.

You should implement hiLowWin inside the hiLow module using an always/case statement similar to the one used to implement your hexToSevenSeg. You will need to create a vector out of the 4-separate inputs using the parenthesis operator as shown in Listing 4.3. Note that the code shown Listing 4.3 is incomplete.

Listing 4.3: 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”.

#### 4.9 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 4.8.

Figure 4.8 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.

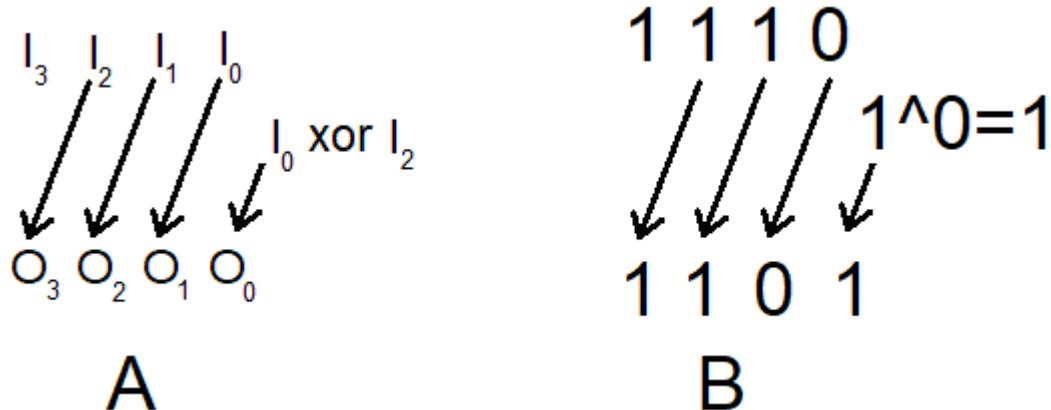


Figure 4.8: A) A schematic illustration of a 4-bit LFSR operation. B) The input 4b'1110 produced the output 4b'1101.

Let's use Table 4.2 to understand what happens if the input in Figure 4.8 was 7'b1110, which, when interpreted as a decimal number, is 14. The upper 3-bits of output are formed by shifting the input left by one bit. The least significant bit of the output is formed by computing  $1 ^ 0$  which equals 1. The resulting output is 4'b1101, when interpreted as a decimal number, equals 13. Fill in the next blank row of Table 2 using decimal 13 as an input. Repeat for the last row of the table.

Table 4.2: The first iteration of the LFSR shown in Figure 4.8 when started at decimal 14.

$O_3$	$O_2$	$O_1$	$O_0$	decimal
1	1	1	0	14
1	1	0	1	13

If you continued the output from the shift operation performed in Table 4.2 you would eventually find a decimal number that repeats because there are only 16 different combinations of 4-bits. Call this repeat number the nexus. The length of the sequence of numbers a nexus back to itself is the length of the sequence. The length of the sequence generated by the operation in Figure 4.8 is 15. This means that if Table 2 had 15 rows and you filled them all in, you would get 14 on the 14<sup>th</sup> 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 4.2 waiting #20 between inputs. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Switch radix

to unsigned decimal for input and output (right click on signal name in wave pane and select radix -> unsigned).

#### 4.10 Module: hiLow

The hiLow module stitches together all the modules and contains all the signals shown in Figure 4.2. 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 4.2. To provide guidance on this process let's focus on the 2:1 mux from Figure 4.9. This 2:1 mux is reproduced in Figure 4.9.

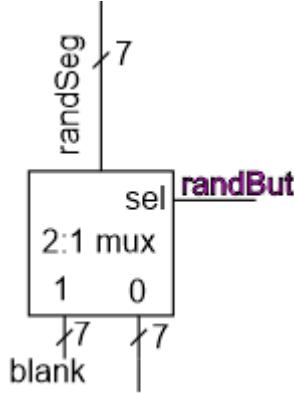


Figure 4.9: A small piece of hardware from Figure 4.2.

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 4.9, the output of the 2:1 mux is already named **randSeg** and the select line is named **randBut**. The data input **y1** will have a constant value **7b'1111111**, needed to produce a blank 7-segment display. The input **y0** is the output from a hexToSevenSeg module, 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\textquotesingle b1111111, RandHex, randBut, randSeg)
```

Once you get the hang of it, you are just translating the system architecture of Figure 4.2 into words.

For the hiLow module, 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).

## 4.11 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         |

## 4.12 Pin-Assignment and Synthesis

Use the image of the development board in Figure 4.1 and the information in the board User Guide to determine the FPGA pins associated with the input and output devices used by the hiLow module.

Table 4.3: 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        |             |
| 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 4.14 for some useful debugging tips.

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

### 4.13 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 4.2.
- [Link](#) Complete the testbench for the lfsr module. Create timing diagram that asserts the four inputs listed in Table 4.2 waiting #20 between inputs. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Switch radix to unsigned decimal for input and output (right click on signal name in wave pane and select radix -> unsigned).

#### 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 4.3.
- Demonstrate your completed circuit to a lab team member.

## 4.14 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 top pane displays a portion of the Verilog code:

```

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

```

The bottom pane shows the error log:

```

10137 Verilog HDL Procedural Assignment error at hilow.v(65): object "greenLEDs" on left-hand side of assignment must have a variable data type
10137 Verilog HDL Procedural Assignment error at hilow.v(66): object "greenLEDs" on left-hand side of assignment must have a variable data type
10137 Verilog HDL Procedural Assignment error at hilow.v(67): object "greenLEDs" on left-hand side of assignment must have a variable data type
10137 Verilog HDL Procedural Assignment error at hilow.v(68): object "greenLEDs" on left-hand side of assignment must have a variable data type
10137 Verilog HDL Procedural Assignment error at hilow.v(69): object "greenLEDs" on left-hand side of assignment must have a variable data type
10137 Verilog HDL Procedural Assignment error at hilow.v(70): object "greenLEDs" on left-hand side of assignment must have a variable data type
10137 Verilog HDL Procedural Assignment error at hilow.v(71): object "greenLEDs" on left-hand side of assignment must have a variable data type
10137 Verilog HDL Procedural Assignment error at hilow.v(72): object "greenLEDs" on left-hand side of assignment must have a variable data type
Quartus II 64-Bit Analysis & Elaboration was unsuccessful. 8 errors, 2 warnings

```

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 top pane displays a portion of the Verilog code:

```

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

```

The bottom pane shows the warning log:

```

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
10236 Verilog HDL Implicit Net warning at hilow.v(50): created implicit net for "randNum"

```

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 top pane displays a portion of the Verilog code:

```


```

The bottom pane shows the error log:

```

12061 Can't synthesize current design -- Top partition does not contain any logic
Quartus II 64-Bit Analysis & Synthesis was unsuccessful. 1 error, 3 warnings
293001 Quartus II Full Compilation was unsuccessful. 3 errors, 3 warnings

```

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.

---

## Laboratory 5

# High Low Guessing Game With Hints

---

### 5.1 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

### 5.2 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 5.1 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*

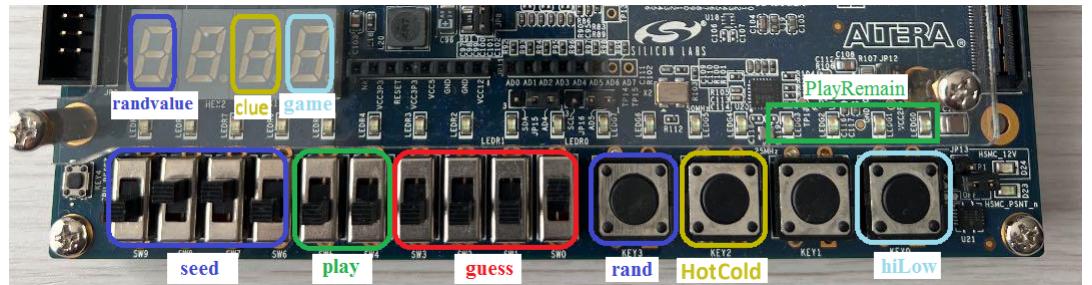


Figure 5.1: The input and output you should use to realize your digital system.

and coldThreshold as shown in Figure 5.2. This figure can be interpreted as follows:

- If difference < warmThreshold the guess is Hot
- If (difference  $\geq$  warmThreshold) and (difference < coldThreshold) the guess is Warm
- If difference  $\geq$  coldThreshold the guess is Cold

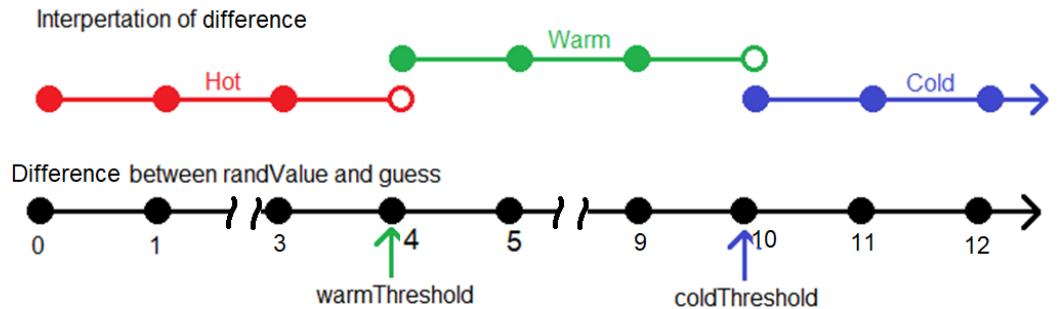


Figure 5.2: The interpretation of the quality of a guess in terms of thresholds.

Explore the relationship between guess, secret number and the quality of the guess by completing Table 5.1. Assume a 4-bit word size for guess and the secret number and use warmThreshold = 4 and ColdThreshold=10.

Table 5.1: Determine the quality of a guess at the secret number.  
Your answer may be a number, pair of numbers, a range or a pair of ranges

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

The 7-segment display called clue will communicate the quality of the user's guess to the

user. It will do this by displaying ‘C’ if the guess is Cold, ‘A’ if the guess is wArm, ‘H’ if the guess is Hot.

### 5.3 System Architecture

You will use the system architecture shown in Figure 5.3 to design your circuit.

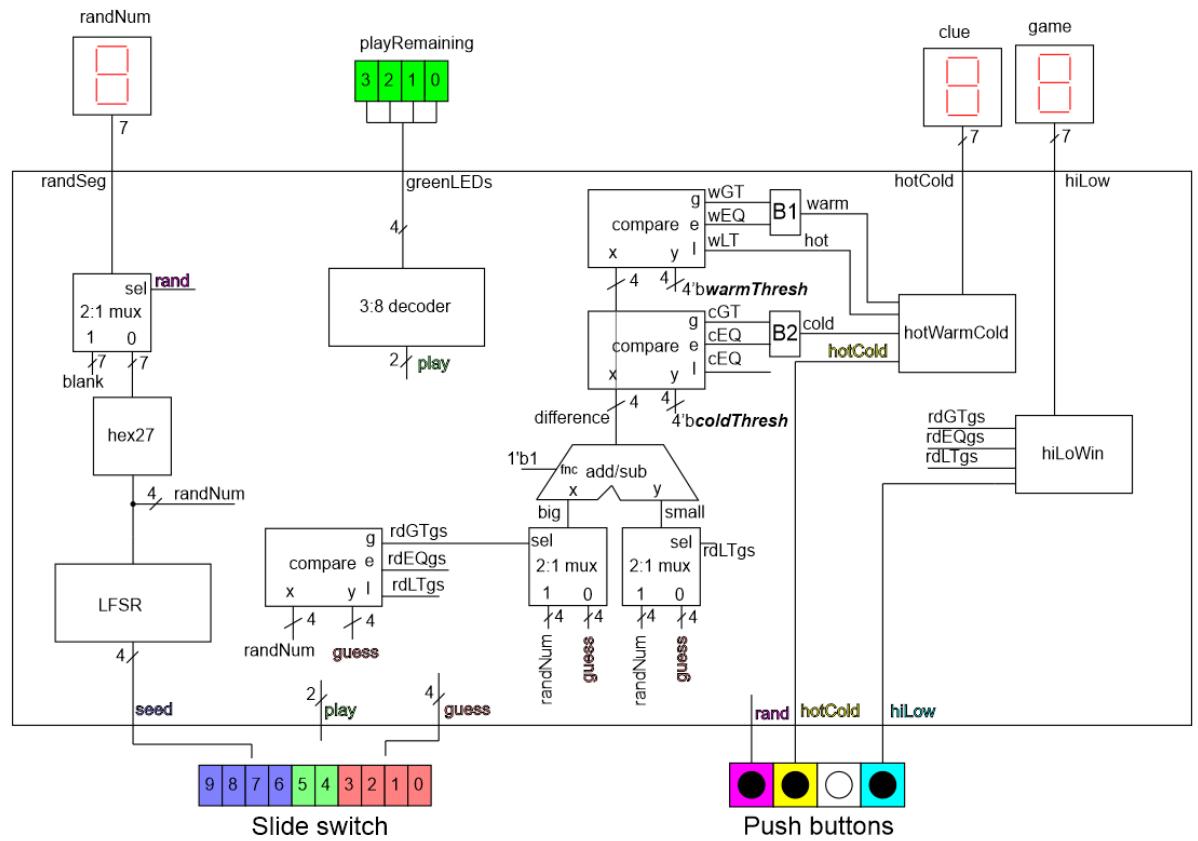


Figure 5.3: System architecture for the guessing game with hint.

You should use last week’s lab as a starting point. The new circuitry is shown in the middle of Figure 5.3. 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 *x,y* inputs are 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 5.2.

## 5.4 Module: 2:1 Mux

This module was discussed in Lab 4.

## 5.5 Module: Compare

This module was discussed in Lab 4.

## 5.6 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 5.3. Since the inputs to the adder subtractor in the system architecture will not generate overflow, the overflow outputs from this adder subtractor are not needed. When you do not need an output from a module, you can leave its parameter slot unfilled. This explains the pair of empty fields at the end of the module instantiation shown in Listing 5.1.

Listing 5.1: Top, module definition for an adder subtractor. Bottom, module instantiation of the adder subtractor in Figure 5.3.

```
// 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.

## 5.7 Logic: hotWarmCold

The goal of this section is to determine the logic inside the hotWarmCold logic block in Figure 5.3. 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 5.2.

Listing 5.2: 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 5.3.

To understand the logic inside the B1 and B2 logic blocks let's work through a set of examples in Table 5.2. In these examples you will let:

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

as a first step, let's determine the outputs of the warm comparator. The x input to the warm comparator 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:

- $w_{GT} = 1$
  - $w_{EQ} = 0$
  - $w_{LT} = 0$

Using similar reasoning we find 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 5.2: Complete the following table, let  $warmThresh = 4$  and  $coldThresh = 10$ . Leave comparator outputs which re 0 blank.

Now, you need to use the values in Table 5.2 to determine the logic for each of the three outputs from the “discrete logic” block shown in Figure 5.3. To do this 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:guessWithHintSys}
```

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.

Once you have the logic or the *hot/warm* and *cold* signals you can start building the *hotWrmold* 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 5.4 when the *hotCold* button is pressed. The 7-segment display should be blank when the *hotCold* button is unpressed.

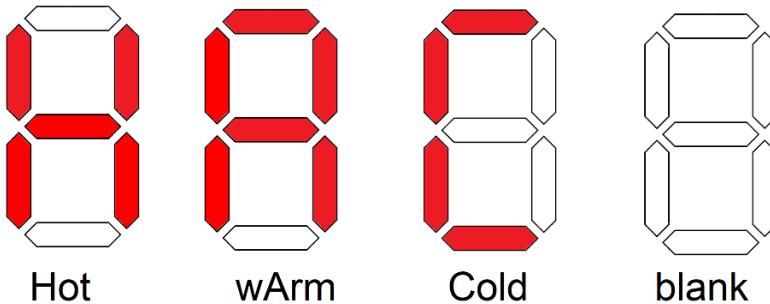


Figure 5.4: The illuminated patterns to inform the guesser about the magnitude of their guess.

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 5.3.

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

```
/*
*****
// Logic to display the quality of the guess
//      Hot = 'H' = <show binary code>
//      wArm = 'A' = <show binary code>
//      Cold = 'C' = <show binary code>
//
```

```
//          hex[0]
//          |
//          hex[5] |   _____|   hex[1]
//          |           |       |
//          |   _____|   hex[6]
//          hex[4] |   |       hex[2]
//          |   |       |
//          |   _____|
//          hex[3]
//*****
```

## 5.8 Module: hiLow

This is the top-level module in Figure 5.3, 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 5.4. If you got the previous lab working correctly, then you should use the bottom declaration in Listing 5.4.

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

```
// 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.

## 5.9 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 5.3 contains the values that you will use to test your circuit. Before using the testbench, you need to understand what your circuit should output. The signal names in the top row of Table 5.3 are borrowed from the system architecture in Figure 5.3. Fill in the missing binary and decimal values for the cells in the guess, big, small and difference columns. In the Comment column, put the quality of the guess as either “Hot”, “Warm” or “Cold”.

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

| Test | seed    | randNum | guess     | big | small | Difference | Comment |
|------|---------|---------|-----------|-----|-------|------------|---------|
| 1    | 4'b1010 | 4'b0100 | 4'b1111   |     |       |            |         |
| 2    |         |         | =14       |     |       |            |         |
| 3    |         |         | 4'b1101 = |     |       |            |         |
| 4    |         |         | 4'b1000 = |     |       |            |         |
| 5    |         |         | 4'b0111 = |     |       |            |         |
| 6    | 4'b1111 | 4'b1110 | 4'b0011   |     |       |            |         |
| 7    |         |         | =4        |     |       |            |         |
| 8    |         |         | =5        |     |       |            |         |
| 9    |         |         | 4'b1010 = |     |       |            |         |
| 10   |         |         | 4'b1011 = |     |       |            |         |
| 11   |         |         | 4'b1110 = |     |       |            |         |

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 5.5.

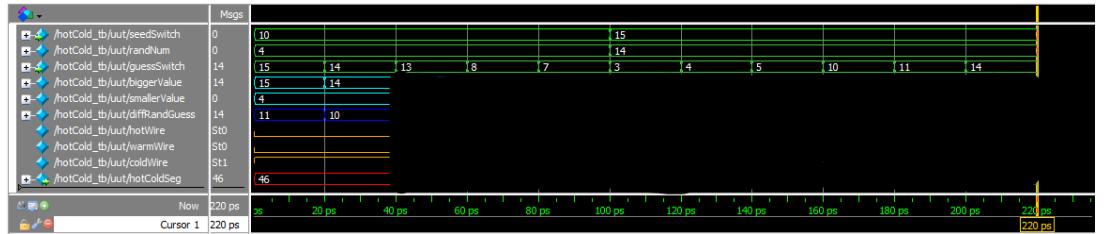


Figure 5.5: A partially obscured timing diagram generated by the testbench.

## 5.10 Pin-Assignment and Synthesis

Use the image of the C5G Development Board in Figure 5.1 and the information in the User Guide to determine the FPGA pins associated with the input and output devices used by the hiLow module.

Table 5.4: Pin-assignment for the High Low Guessing Game with Hints.

| Segment | randSeg | hotColdSeg | hiLowSeg |
|---------|---------|------------|----------|
| seg[6]  | AC22    |            |          |
| seg[5]  |         |            |          |
| 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 5.12 for some useful debugging tips.

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

## 5.11 Turn in

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

### System Architecture

- Complete Table 5.1.

### Discrete Logic block

- Complete Table 5.2
- [Link](#) Logic for hot, warm, and cold signals

### Module: hiLow

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

### Pin-Assignment and Synthesis

- Complete pin-assignment table or ll the signals in Table 5.4.
- Demonstrate your completed circuit to a lab team member.

## 5.12 Debugging Tips

Even when your program executes successfully, you may get the warnings shown in Figure 5.6. These are mainly the result of the unused overflow outputs from the adder subtractor. You can filter out all the compile messages by clicking on the yellow triangle (with the blue three in this case) on the top line of the console window. Note, if there are several related warnings, they will have one top-level warning with all the instances accessible by clicking the expander arrow (it looks like “>”) to the left of the warning triangle.

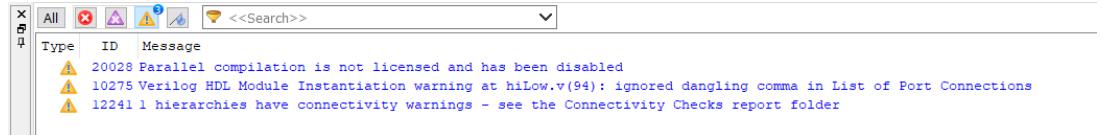


Figure 5.6: The messages console filtered by warnings.

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 5.7, 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.

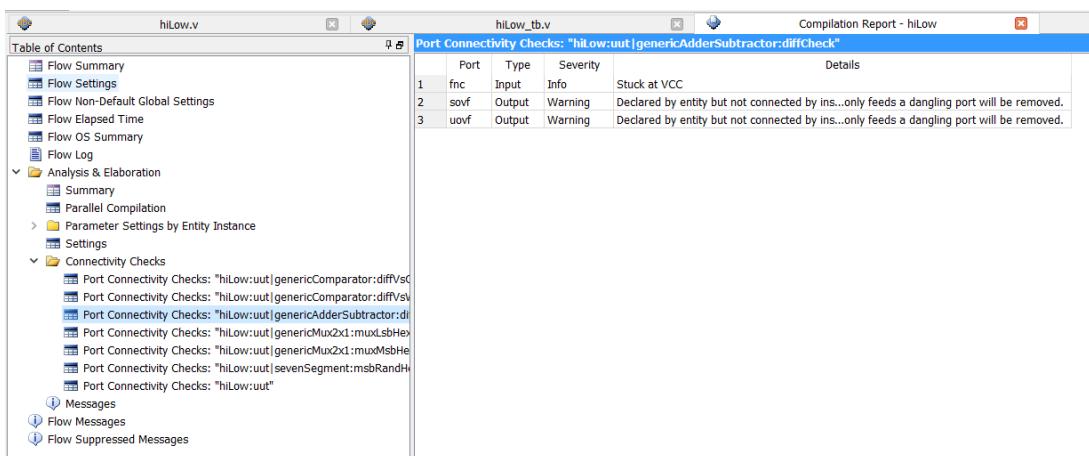


Figure 5.7: Connectivity Checks report for a working hiLow circuit.



---

## Laboratory 6

# Calculator With Friendly Output

---

### 6.1 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

### 6.2 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 6.1 and display the results as decimal, base-10, values, not as hexadecimal values.

On the surface, this should require nothing more than connecting some slide switches to the x and y inputs of an adder/subtractor which sends its output to a 7-segment display. And for the most part this is correct. However, instead of displaying the input and output of the adder as hexadecimal values, you will display them as 2-digit decimal values.

The user input and output are shown in Figure 6.1. The user enters a pair of 4-bit operands using the left-most slide switches, **xSlide** and **ySlide**. The value entered for **xSlide** is displayed on the two (red) **xDisplay** 7-segment displays. The value entered for **ySlide** is displayed on the two (green) **yDisplay** 7-segment displays. The leftmost the **addSub** buttons specify the operation performed on **xSlide** and **ySlide**. The result is **xSlide + ySlide** or **xSlide - ySlide**.

The **interp** button determines how the values are displayed on the 7-segment display. When unpressed, the 7-segment displays show the decimal value, when pressed, the 7-segment displays show 2's complement. This will be explained in the next section. As we have only 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.

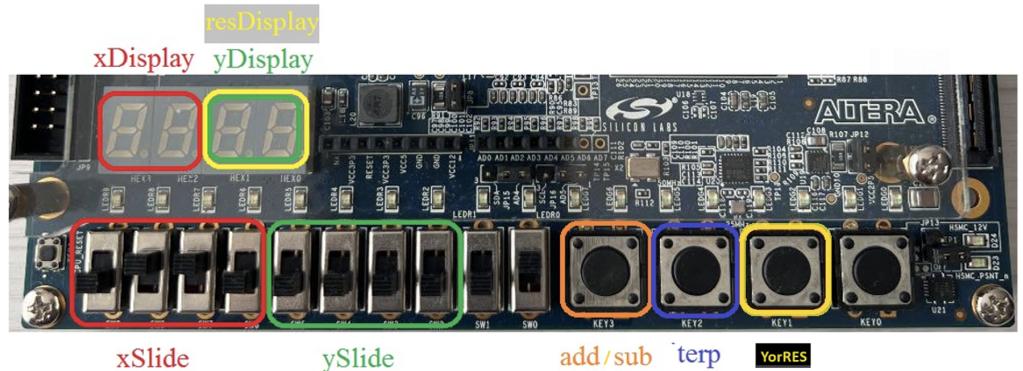


Figure 6.1: The input and output of the calculator digital circuit.

### 6.3 System Architecture

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

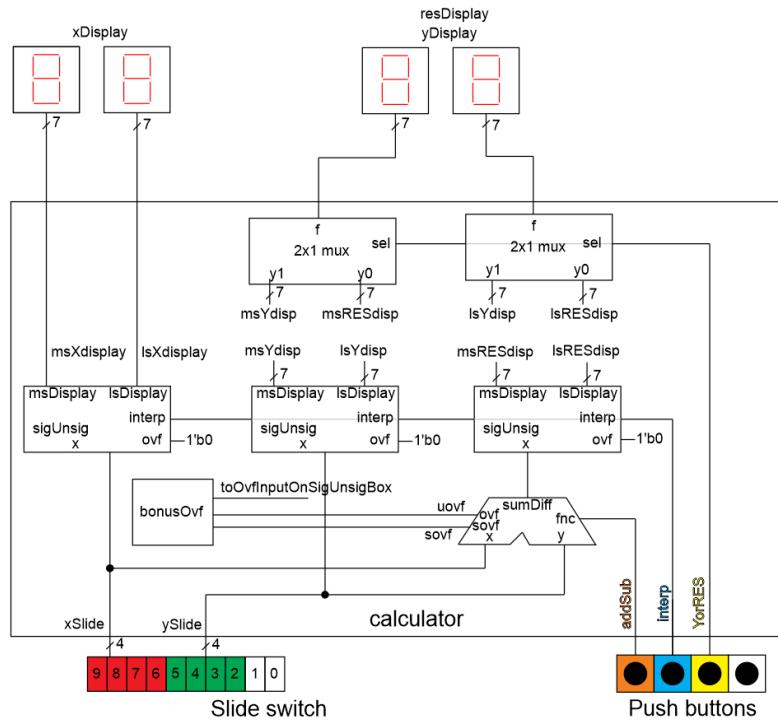


Figure 6.2: The system architecture of the calculator.

## 6.4 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 6.2. 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 6.1.

Listing 6.1: Module declaration for the `sigUnsig` module.

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

The 4-bit input `x` is interpreted as either signed (2's complement value) when `interp = 1` or unsigned (regular binary number) when `interp = 0`. The `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 6.1 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 6.1: 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 6.1. 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 6.2. 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 6.2: Logic that determines the output of the 4:1 muxes in Figure 6.4.

```

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 6.3, 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".

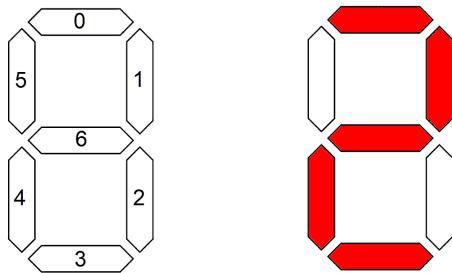
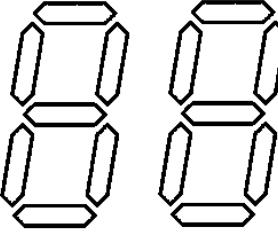
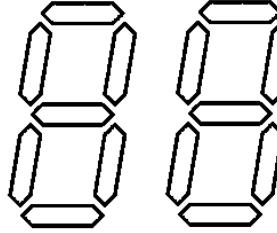
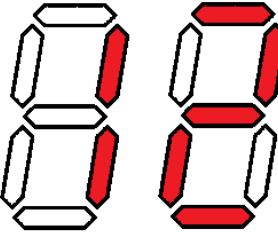
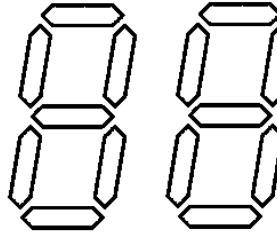
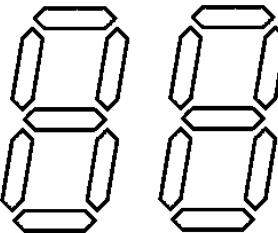
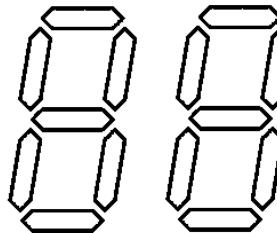


Figure 6.3: The logical arrangements of the segments in a 7-segment display.

Test your understanding of the **signUnsign** output by completing Table 6.2. 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 6.2: 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<br>interp = 1<br>ovf = 0 |    | 4'b0111<br>interp = 0<br>ovf = 0 |    |
|                                  | msDisplay = 7'b<br>lsDisplay = 7'b                                                  |                                  | msDisplay = 7'b<br>lsDisplay = 7'b                                                    |
| 4'b1100<br>interp = 0<br>ovf = 0 |   | 4'b1000<br>interp = 1<br>ovf = 0 |   |
|                                  | msDisplay = 7'b1111001<br>= 7'h79<br>lsDisplay = 7'b0100100 =<br>7'h24              |                                  | msDisplay = 7'b<br>lsDisplay = 7'b                                                    |
| 4'b1100<br>interp = 1<br>ovf = 0 |  | 4'b1010<br>interp = 1<br>ovf = 1 |  |
|                                  | msDisplay = 7'b<br>lsDisplay = 7'b                                                  |                                  | msDisplay = 7'b<br>lsDisplay = 7'b                                                    |

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

Complete Figure 6.4 by adding the following:

- Connect the inputs of the 4:1 mux using the logic described in Listing 6.2.

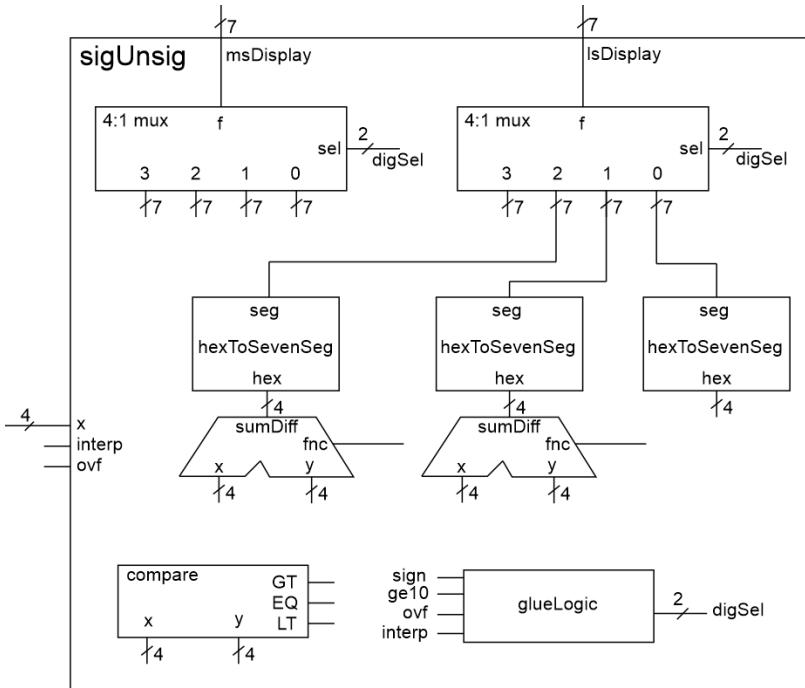


Figure 6.4: The internal architecture of the signUnsig module.

- Connect the inputs of the and adder subtractors using the logic described in Listing 6.2.
- 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 6.4.

#### `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 6.2. 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 6.3 for the `glueLogic` box in Figure 6.4.

Table 6.3: 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 6.2. However, an always case statement requires each of the 16 difference cases to be explicitly

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

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

```
always @(*)
casez ({ovf, interp, xGE10, x[3]}) 
    4'b01?0: digSel = 2'b00;
    default: digSel = 2'b11;
endcase
```

### **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 6.1.
- 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

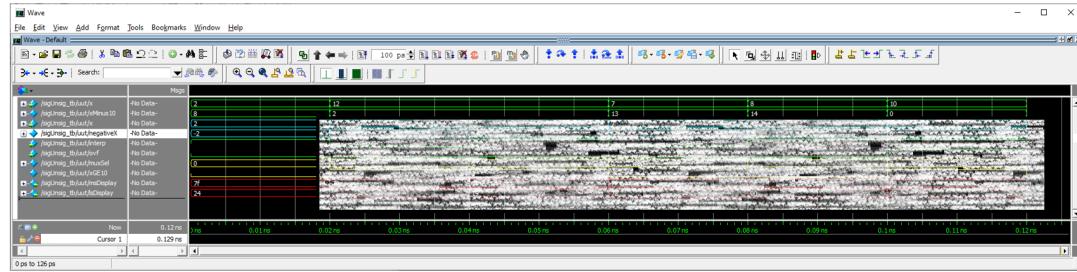
## 6.5 Testbench

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.



## 6.6 Pin-Assignment and Synthesis

Use the image of the FPGA Development Board in Figure 6.1 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 6.4: 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.

## 6.7 Turn in

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

### signUnsig Module

- Complete Table 6.1.
- Complete Table 6.2.
- Complete the code in Listing 6.2.
- Complete Figure 6.4, including:
  - Constant values on inputs of 4:1 mux
  - Constant value on the input of the right-most hexToSevenSeg
  - Value on the input of the adder subtractors
  - Values on the input of the comparator
- Complete Table 6.2.
- Complete Table 6.3.
- **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 6.5.

### Pin-Assignment and Synthesis

- Complete the pin assignment in table 6.4.

## 6.8 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.