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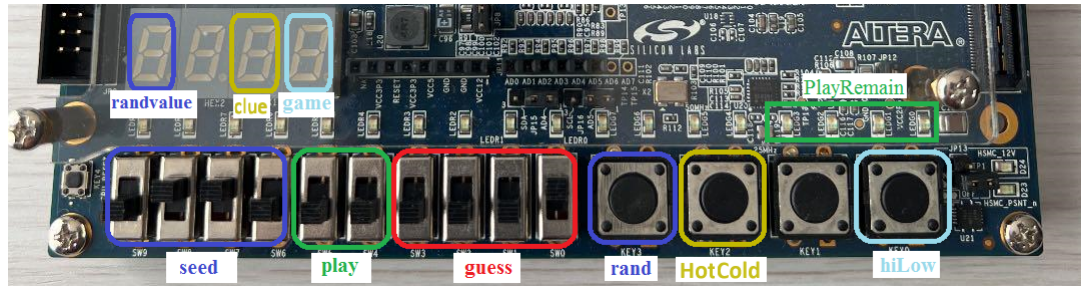
Figure 5.1: The input and output you should use to realize your digital system.

cret number is computed. This difference, called *difference* is compared against warmThreshold and coldThreshold as shown in Figure 5.2. This figure an be interpreted as follows:

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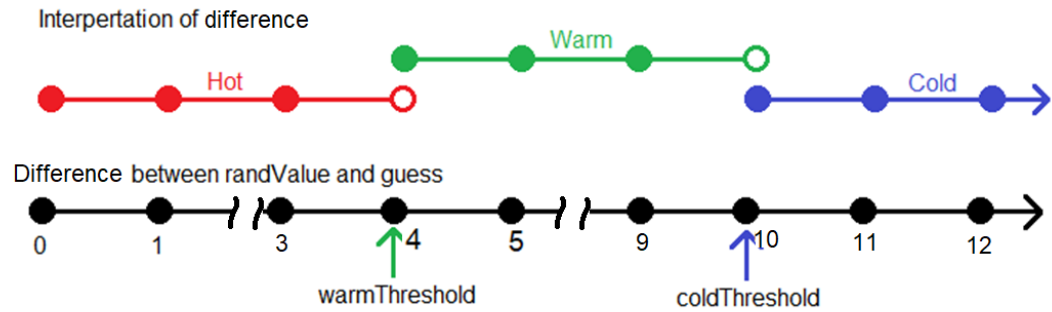- If difference >= coldThreshold the guess is Cold

Figure 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 **C**old, 'A' if the guess is w**A**rm, 'H' if the guess is **H**ot.

5.3 System Architecture

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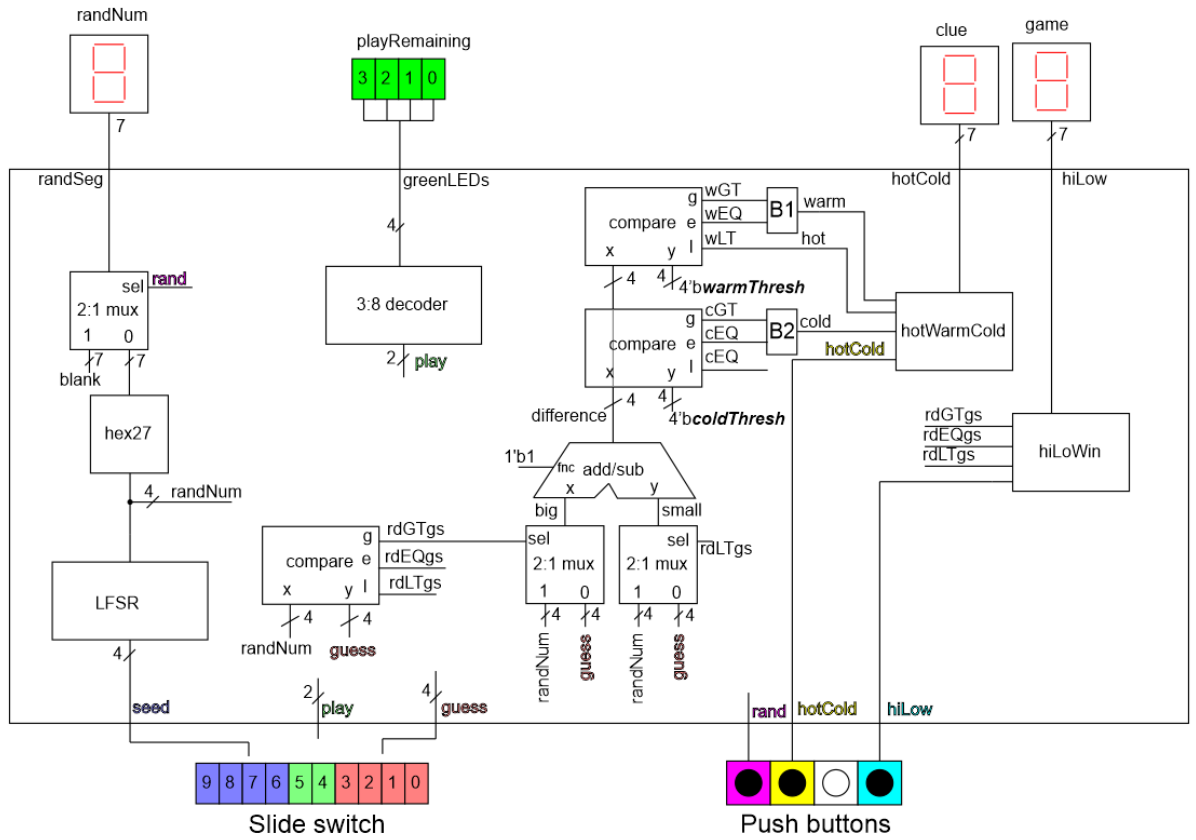

Figure 5.3: System architecture for the guessing game with hint.

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 it's fun input is hardwired to 1'b1, hence it is always computing the diferene 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 instantation 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 comprarator. The x input o the waarm omprator is equal to 9 (the value of *difference*) and the y input equals 4 (the value of *warmThresh*). Since 9 is greater than 4, then:
- wGT = 1
- wEQ =0
- wLT =0

Using similr reasoning we ind that:
- cGT = 0
- cEQ =0
- cLT =1

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

Table 5.2: Complete the following table, let warmThresh = 4 and coldThresh = 10. Leave comprator outputs which re 0 blank.

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

Now, you need to use the values in Table 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.

One you hve the logi or the hot/wrm nd cold signals you an strt building the hotWrmold logi 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.
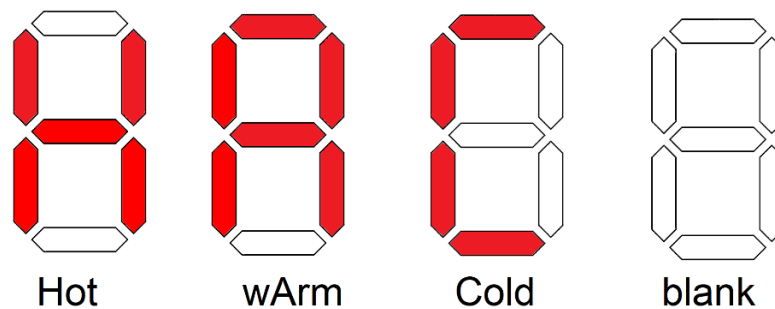
Figure 5.4: The illuminated patterns to inform the guesser about the magnitude of their guess.

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'b1111				
2			=14				
3	4'b1010	4'b0100	4'b1101 =				
4			4'b1000 =				
5			4'b0111 =				
6			4'b0011				
7			=4				
8			=5				
9	4'b1111	4'b1110	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.
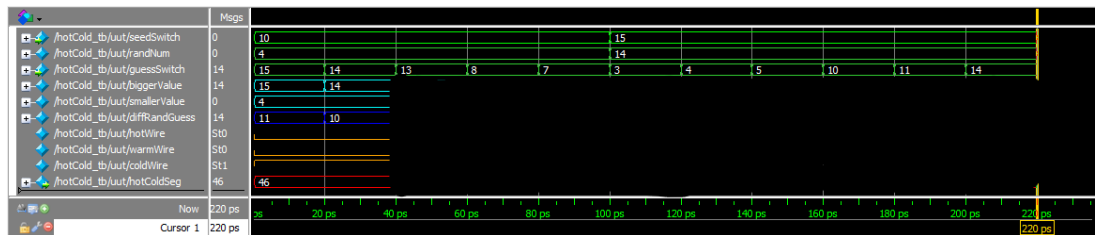

Figure 5.5: A partially obscured timing diagram generated by the testbench.

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.
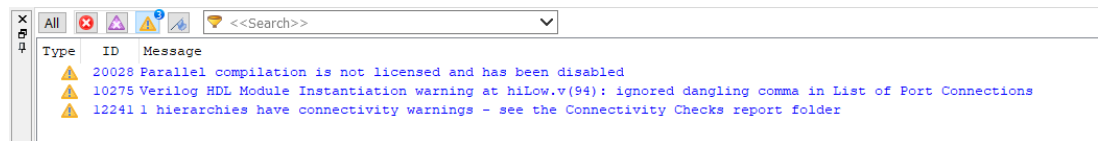

Figure 5.6: The messages console filtered by warnings.

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.