Contents

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

Laboratory 1

High Low Guessing Game With Hints

1.1 Objective

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

The Guessing Game with Hints

This week's assignment asks you to add some enhanced functionality to the guessing game. Since we are adding functionality to the guessing game, it's worth reviewing the guessing game because we will use some of the terms in the description of our enhanced functionality. The game starts with the secret keeper generating a *secret number* between [0 and 15], inclusive. Once the *secret number* is decided, the guesser makes a *guess*, a number in the interval [0 to 15] inclusive, and tells this to the secret keeper. The secret keep then replies to the guesser if *guess* is less than, equal to, or greater than the *secret number*. The game continues with repeated guesser/secret keeper exchange until the guesser correctly identifies the *secret number*.

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

The user input and output, shown in Figure 1.1 are the same as last week's assignment with the exception of the **hotCold** button and **clue** 7-segment display.

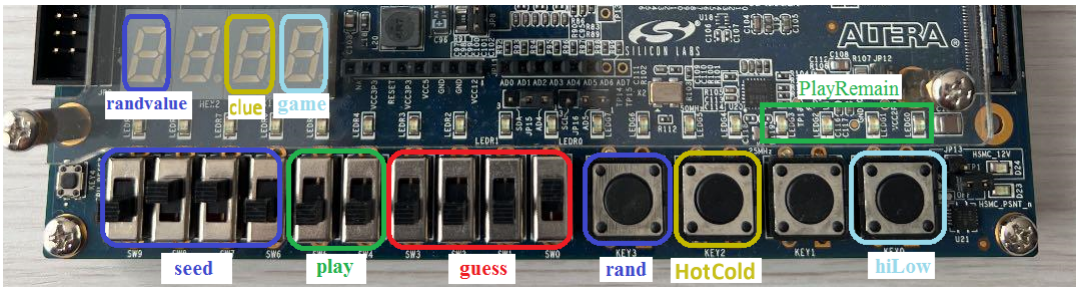

Figure 1.1: The input and output you should use to realize your digital system.

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

The player can request a more refined evaluation of their guess by pressing the **hotCold** button. To make this evaluation, the absolute value of the difference between the *guess* and *secret number* is computed and then comapred against warmThreshold and coldThreshold as shown in Figure 1.2.

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

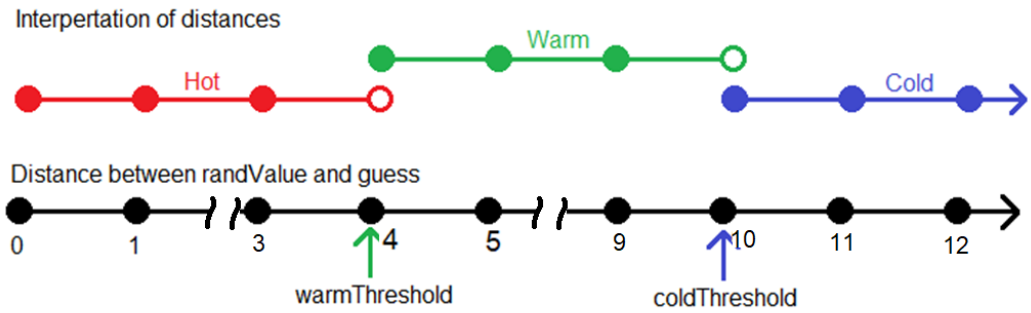

Figure 1.2: The interpretation of the quality of a guess in terms of thresholds.

The difference is always a positive number. For example, if the user's *guess* was 43 and the secret number was 45, then the difference would be 2 making this guess Hot (according to Figure 1.2). If *guess* was 45 and the *secret number* was 43, the difference would be 2 and this guess would also be Hot. Explore the relationship between guess, secret number and the Quality of the guess by completing Table 1.1.

Table 1.1: Determine the quality of a guess at the secret number. Your answer may be a number, pair of numbers, a range or a pair of ranges. Assume a 4-bit word size for guess and the secret number and warmThreshold = 4 and ColdThreshold=10.

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

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

System design

There are an almost unlimited number of ways that you could implement this digital system. For this lab, I want you to use the system architecture shown in Figure 1.3. A few comments about the visual notation used in this schematic are in order.

1) Lines with the same name are connected together.

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

3) The input signals are color-coded to correspond to the colors used in Figure 1.1.

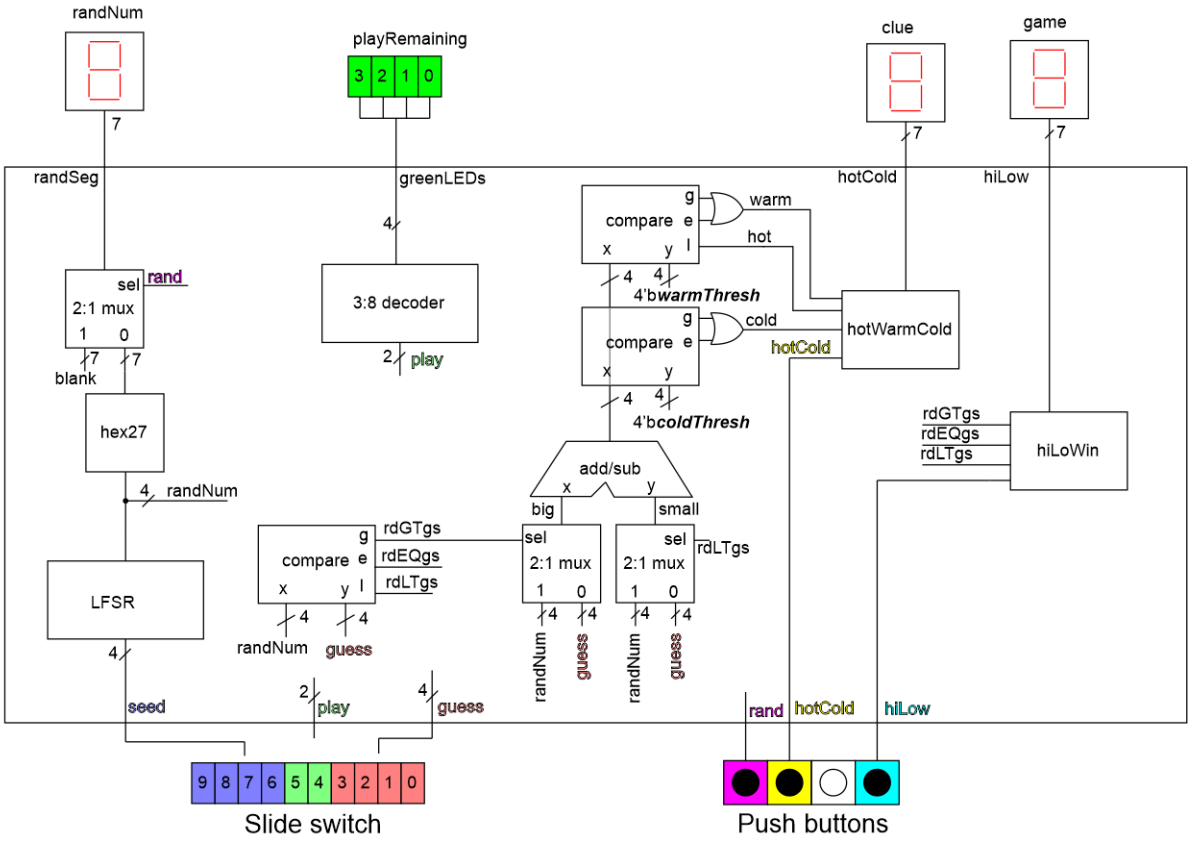

Figure 1.3: System architecture for the guessing game with hint. The select inputs on the big and small 2x1 muxes is intentionally not shown – you will need to figure this out.

The new circuitry is shown in the middle of Figure 1.3 starting with the comparator whose inputs are *randNum* and *guess*. Actually, this comparator should already be in your circuit because it provides its outputs as the input to the hiLoWin module used in the previous lab. This comparator is needed to determine the difference between *randNum* and *guess* as follows.

The difference between the *guess* and *secret number* (called *randNum* in Figure 1.3)that you computed in Table 1.1 is the absolute value of the difference between *guess* and the *secret*

number. We will realize this functionality by placing the larger of *guess* or *randNum* on the x input of the adder subtractor shown in Figure 1.3. The smaller of *guess* or *randNum* is placed on the y input of the adder subtractor. This routing of x and y to the adder subtractor is performed by a pair of 4-bit 2x1 muxes whose select inputs are controlled by one of the comparator's three outputs. The adder subtractor in Figure 1.3 is configured to subtract by hardwiring its fnc input to 1'b1. You will need to determine which single output from the comparator feeds the select input for this pair of muxes.

Let's call the output of the adder subtractor *difference*. This *difference* is compared to the warmThreshold and coldThreshold using a pair of comparators. The values of warmThresh and coldThresh are set in code using the signal declaration and signal assignment statements shown in Listing 1.1.

Listing 1.1: The signal declaration and assignment for guess thresholds.

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

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

1.2 2:1 Mux Module:

This module was discussed in Lab 4.

1.3 Compare Module:

This module was discussed in Lab 4.

1.4 Add/Sub Module:

A N-bit adder subtractor is a basic building block in many digital systems. The N-bit adder subtractor shown in Figure 1.4 adds its N-bit input x and N-bit input y when fnc = 0 and places the result on the N-bit output subDiff. When fnc = 1 the sumDiff output equals x-y. When the inputs and output are interpreted as a 2's complement values, the sovf output equals 1 when the computation results in an overflow. When the inputs and output are interpreted as binary numbers, the uovf output equals 1 when the computation results in an overflow.

I have provided you the Verilog code for the N-bit adder subtractor on Canvas. This module instantiates N full-adders. Thus, you will need to include the full adder module contained in the file fullAdder.v in your project to instantiate a genericAdderSubtractor instance. Listing 2 shows the module declaration for the genericAdderSubtractor. Note that the output from the module follows the fnc input. The module instantiation shown in Listing 2 corresponds to the system architecture shown in Figure 1.3. Since the inputs to the adder subtractor in the system architecture will not generate overflow, the overflow outputs from this adder subtractor are not needed. When you do not need an output from a module, you can leave its parameter slot unfilled. This explains the pair of empty fields at the end of the module instantiation shown in Listing 2.

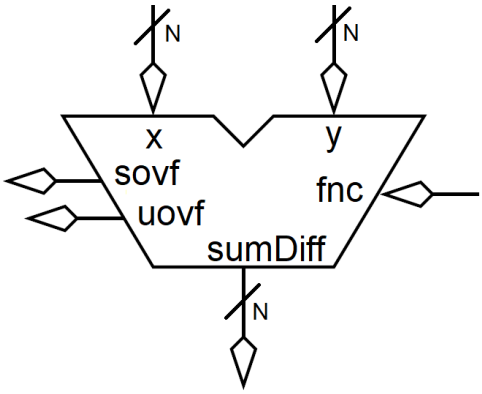

Figure 1.4: A schematic representation of a N-bit adder subtractor.

Listing 1.2: Top, module definition for an adder subtractor. Bottom, module instantation of the adder subtractor in Figure 1.3.

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

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

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

Discrete Logic block:

The input for this logic will come from the output of the two comparators that compare difference and the warm or cold threshold, see Figure 1.3. It would be helpful to take a moment to copy down the logic between the adder subtractor and the hotWarmCold module output in your notes. We will call the comparator that compares *difference* (the output from the adder subtractor) and *warmThresh*, the warm comparator. The other comparator will be called the cold comparator for obvious reasons.

The warm and cold comparators generate a total of 6 signals sent to the "discrete logic" box in Figure 1.3, but the logic in "discrete logic" does not need all of them. To get a better handle on this, let's look at some examples to help uncover the relationship between the magnitude of the difference between the *guess* and *randNum* and the output of the warm and cold comparators. Note, that in the system architecture shown in Figure 1.3, *difference* is applied to the x input of the comparators and the *warmThresh* and *coldThresh* are applied to the y inputs.

Let's assume *coldThresh* = 10 and *warmThresh* = 4. Complete Table 1.2 by comparing the value in the column labeled "*difference*" to *warmThresh* and *coldThresh* and asserting the appropriate comparators outputs. Note, the comparator outputs are prefixed by "w" for warm comparator output and "c" for cold comparator output.

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

Table 1.2: Complete the following table to determine which comparator outputs are needed to determine the quality of a guess. Let warmThresh = 4 and coldThresh = 10.

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

Now, you need to use the values in Table 1.2 to determine the logic for each of the three outputs from the "discrete logic" block shown in Figure 1.3. To do this, look at the conditions that cause each of the outputs and write an expression using AND and OR to describe when that output equals 1.

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

For this block of code:

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

hotWarmCold block:

You will implement the hotWarmCold logic using an always/case statement that takes in as input the 3 outputs from the "discrete Logic" block, *hot*, *warm* and *cold*. These three outputs form a 1's hot code because a guess can only be one of these three conditions at a time. When the **hotCold** button is pressed, the output of the hotWarmCold block forms an illuminate 7-segment representation of the quality of the guess shown in Figure 1.5. The 7-segment output from the hotWarmCold block should be blank when the **hotCold** button is unpressed.

For this block of code:

- Use an always/case statement.

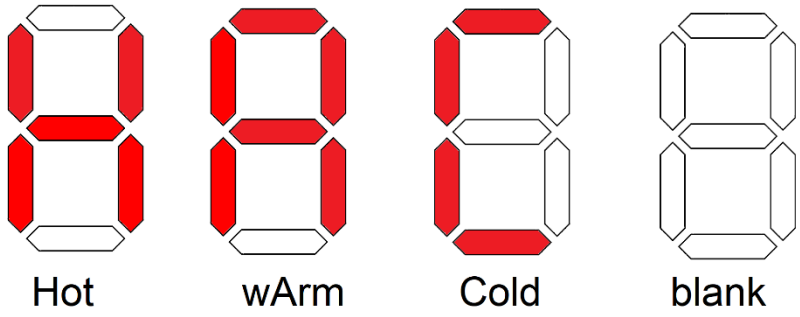

Figure 1.5: The illuminated patterns to inform the guesser about the magnitude of their guess.

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

Listing 1.3: 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]
//*****************************************
```

1.5 hiLow module:

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

you should use the top module declaration in Listing 1.4. If you got the previous lab working correctly, then you should use the bottom declaration in Listing 1.4.

Listing 1.4: The module declaration for the enhanced hiLow module if you did or did not get the previous lab working.

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

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

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

For this block of code:

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

1.6 hiLow_tb module:

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

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

Table 1.3 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 1.3 are borrowed from the system architecture in Figure 1.3. 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 1.3: Table : The values used in the hiLow testbench.

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

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

1.7 Pin Assignment:

Use the image of the Development Board in Figure 1.1 and the information in the User Guide to determine the FPGA pins associated with the input and output devices used by the hiLow module.

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

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

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

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

1.8 Turn in:

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

System Architecture

- Complete Table 1.1.

Discrete Logic block:

- Complete Table 1.2
- Link Logic for hot, warm, and cold signals

hiLow Module:

- Link Verilog code for the body of the hiLow module (courier 8-point font single spaced), leave out header comments.
- Complete Table 1.3.
- Link Run the testbench for the hiLow module provided on Canvas. Produce a timing diagram with the following characteristics. Zoom to fill the available horizontal space with the waveform. Color inputs green and outputs red. Order the traces from top to bottom as

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

  I do not want the signals from the testbench, but rather the signals from inside the hiLow module. You can do this in ModelSim, by expanding the hiLow_tb instance in the left ModelSim pane shown in Figure 1.6 and selecting "uut". Since uut is an instance of the hiLow module, all the signals accessible in the hiLow module are shown in the center Object pane in Figure 1.6. You can add any of these signals by clicking on them and dragging them into the rightmost Wave pane shown in Figure 1.6.

  When compete, your testbench should look like the timing diagram in Figure 1.7.

Demo:

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

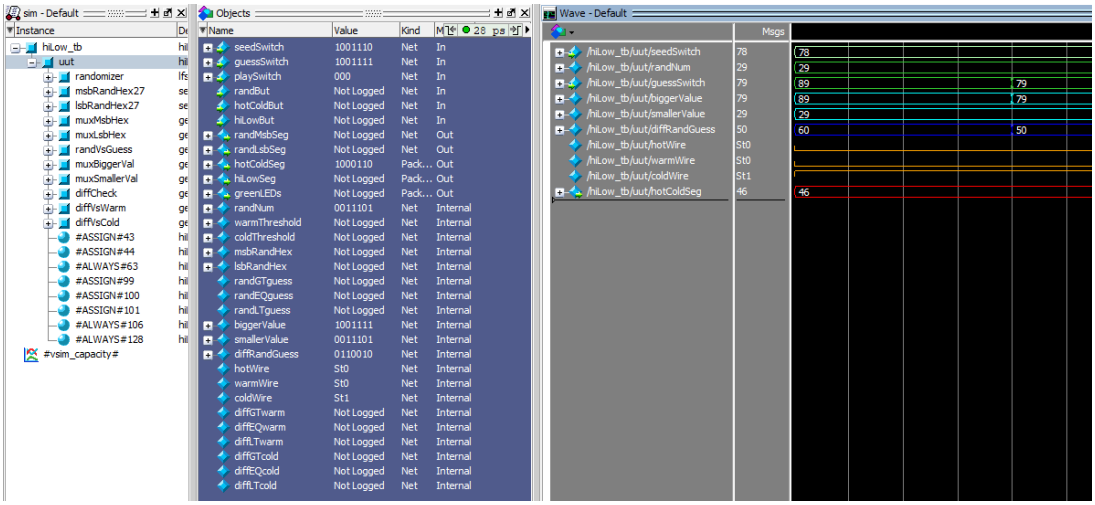

Figure 1.6: Use the design hierarchy to add signals to the testbench.

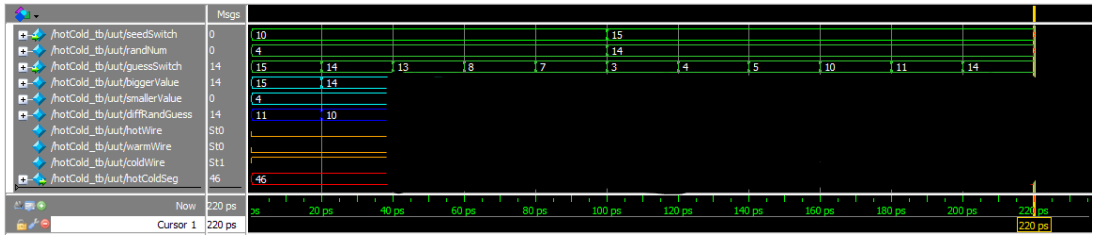

Figure 1.7: A partially obscured timing diagram generated by the testbench.

Clearing errors - Tips from the professor

When my program executed successfully, I got the warnings shown in Figure 1.8. These are mainly the result of the unused overflow outputs from the adder subtractor. You can filter out all the compile messages by clicking on the yellow triangle (with the blue three in this case) on the top line of the console window. Note, if there are several related warnings, they will have one top-level warning with all the instances accessible by clicking the expander arrow (it looks like “>”) to the left of the warning triangle.

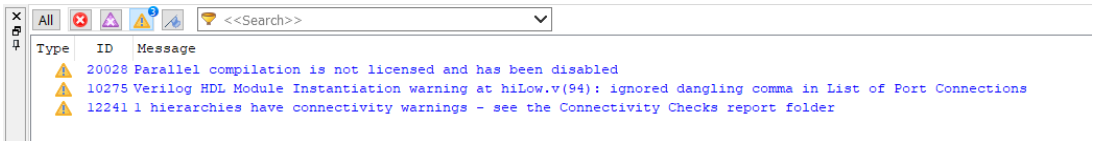

Figure 1.8: The messages console filtered by warnings.

I have found the Connectivity Checks folder in the Compilation Report to help me quickly track down errors. To use it, open the Connectivity Checks folder, click on a Port Connectivity Checks item and read the report in the right pane. In the report shown in Figure 1.9, I selected

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

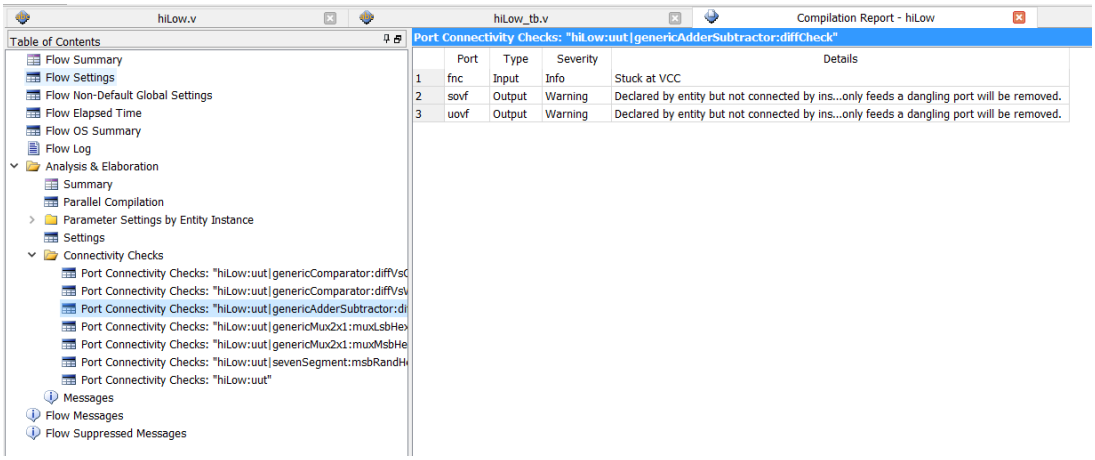

Figure 1.9: Connectivity Checks report for a working hiLow circuit.