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 ¿ secret number*

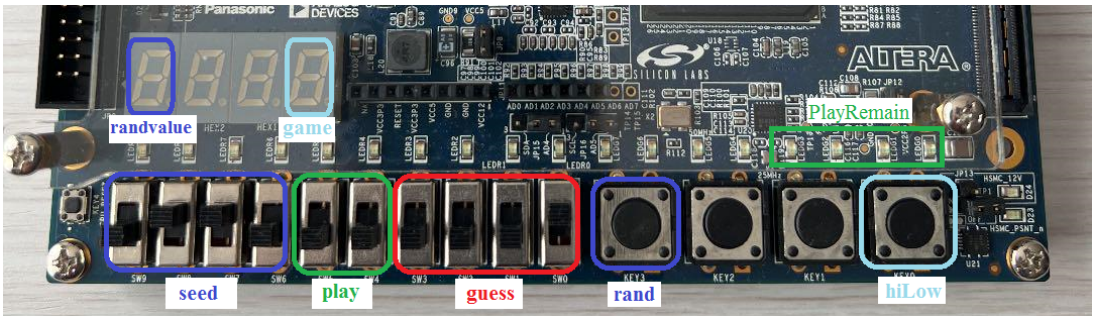


Figure 4.1: The input and output you should use to realize the High/Low Guessing game.

- 'I' when *guess* = *secret number*
- 'L' when *guess* ¡*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, `y0` or `y1` to the N-bit output, `F`, depending on the value of a 1-bit select signal, `s`. When `s` = 0, `F = y0` and when `s` = 1, `F = y1`. 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 `y1` or `y0` 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 instantation of a 2:1 mux in Figure 4.2.

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

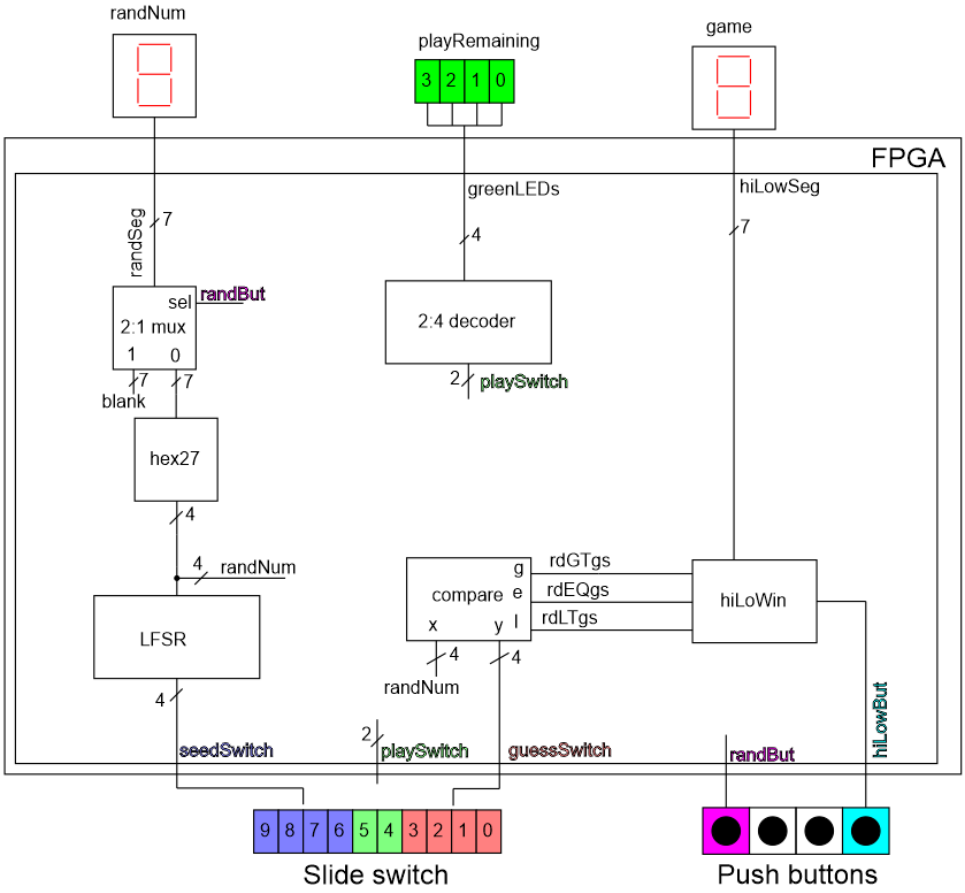


Figure 4.2: System architecture for the guessing game.

```
// Module instantation 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 instantation, 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 occuring 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.

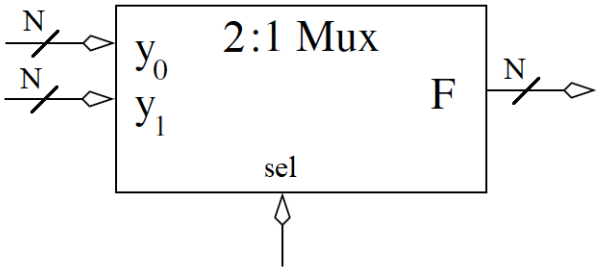


Figure 4.3: The schematic representation of a 2:1 mux.

Compilation Report - hiLow

Port Connectivity Checks: "genericMux2x1:muxMsbHex"

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

Figure 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.

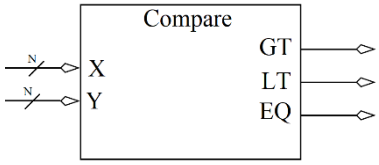


Figure 4.5: A schematic representation of a N-bit comparator.

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.

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

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

Listing 4.2: 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 compataror in the hiLow digital circuit
genericComparator #(4) randVsGuess(randNum, guessSwitch, \
                    randGTguess, randEQguess, randLTguess);
```

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

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.

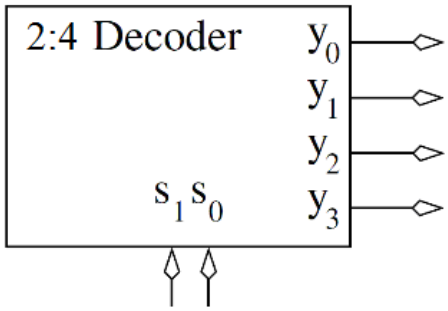


Figure 4.6: A 2:4 decoder.

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.

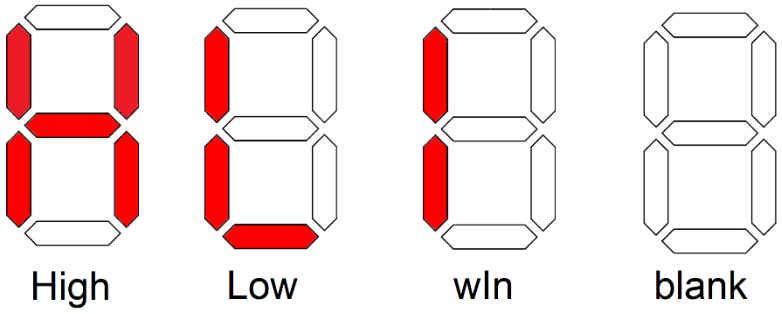


Figure 4.7: The illuminated patterns to inform the guesser about the magnitude of their guess.

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'bxxxxxxx;
                default: hotColdSeg = 7'bxxxxxxx;
        endcase
```

You should put this Verilog code in the hiLow module as one of the many concurrent statement. This means that you should not have a separate Verilog file for the hiLowWin. Remember that the output type from an always/case statement must have the "reg" qualifier, not "wire".

4.9 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 $I_2 \ldots I_0$ being shifted one bit to the left on their way to the outputs $O_3 \ldots O_1$. The output O_0 is formed by computing $I_2 \hat{} I_0$ where "^" is the xor operation.

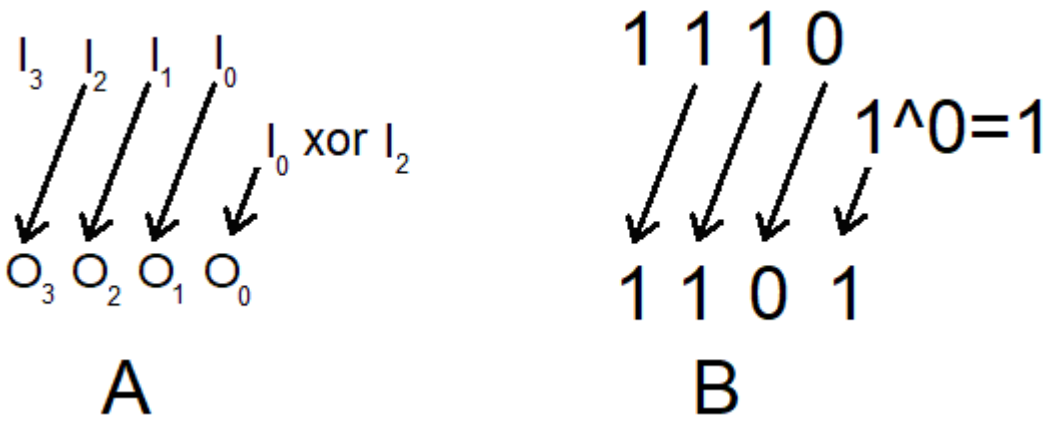


Figure 4.8: A) A schematic illustration of a 4-bit LFSR operation. B) The input 4b'1110 produced the output 4b'1101.

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^{th} 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 repsoduced in Figure 4.9.

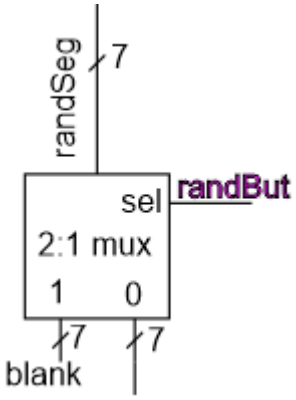


Figure 4.9: A small piece of hardware from Figure 4.2.

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

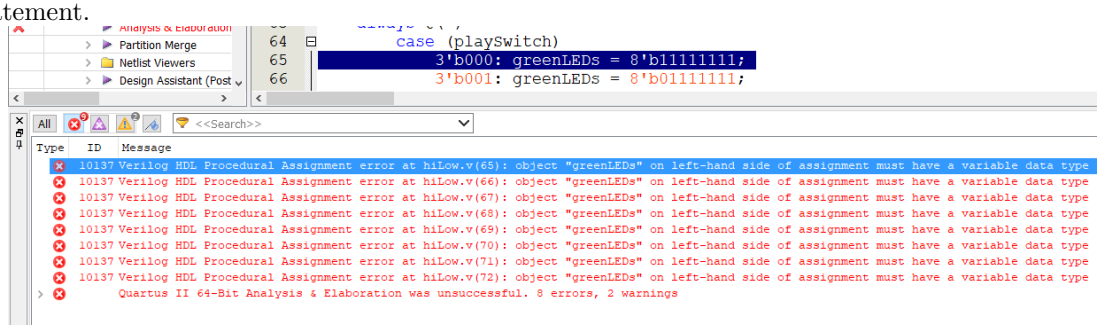
- Completed pin assignment table for all the signals in Table 4.3.
- Demonstrate your completed circuit to a lab team member.

4.14 Debugging Tips

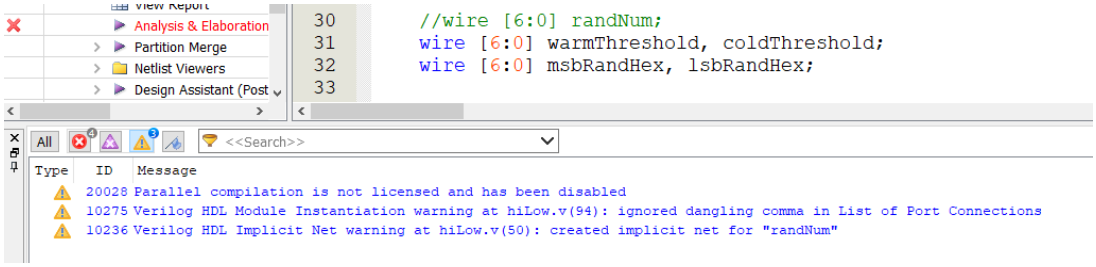
This laboratory typically generates a variety of new errors that you have not seen before. This section on useful debugging techniques will help you more effectively interpret the compilers output to locate errors in your code. The following is an example story of someone debugging their code...

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.

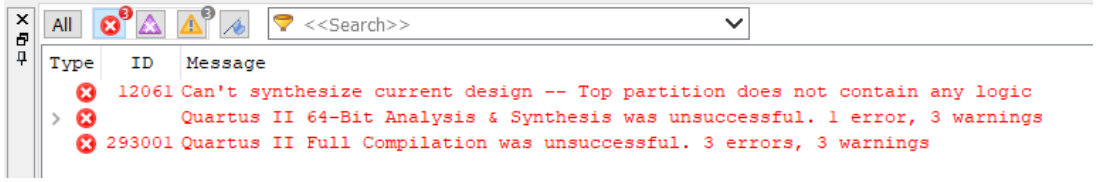
Th error below is a result of defining the output `wire {[}7:0{]} greenLEDs`. the output should have `reg{[}7:0{]} greenLEDs;` because greenLEDs is the output of an always/case statement.



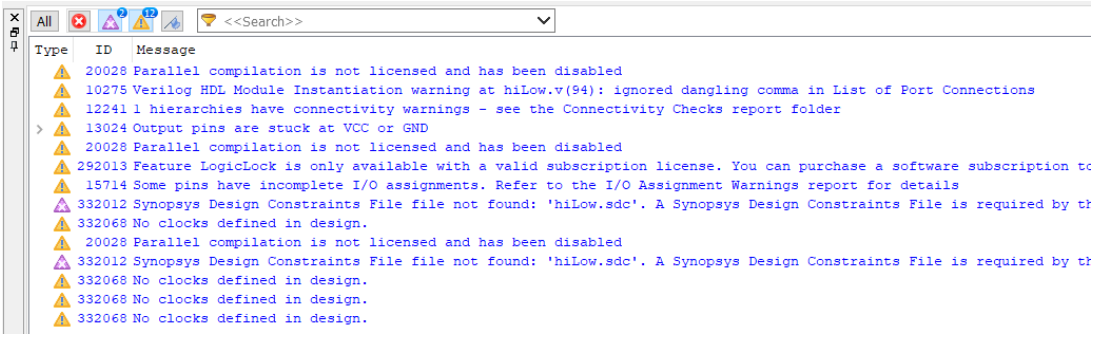
The following error is the result of forgetting to include the declaration of randNum. The top warning always appears and the second is a result of an unused output on the adder (more about this in the next lab).



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



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”.



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