EENG 284 – Digital Design

Lab 4- The Guessing game

# Objective

The objective of this lab is to use a combination of basic building block and custom combinational logic blocks to realize a complex digital circuit.

**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** 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 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 Cyclone V GX board using the inputs and outputs shown in Figure 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.

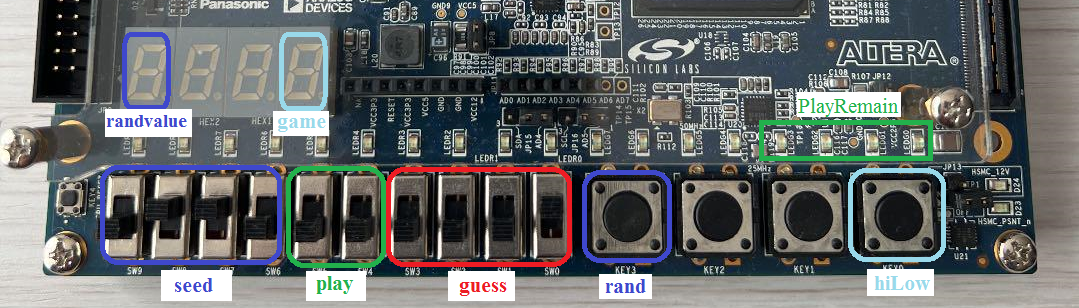


Figure 1: The input and output you should use to realize your digital system.

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*
* ‘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.

**System design**

There are an 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 2. A few comments about the visual notation used in this schematic are in order.

1. Lines with the same name are connected together. For example, the rand button input is connected to the 2:1 mux in the upper left corner of the FPGA.
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. For example, the rand signal is sent to the select input of both 2:1 muxes in the upper left corner of the schematic. These unconnected connections are called “air-wires.”
3. The input signals are color-coded to correspond to the colors used in Figure 1.

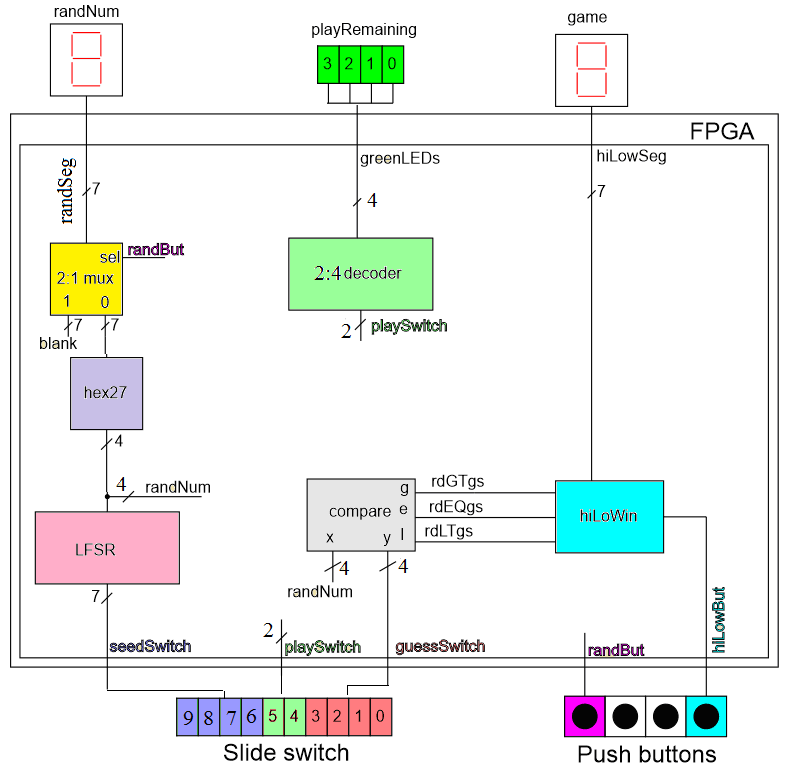


Figure 2: System architecture for the guessing game.

# 2:1 Mux Module:

A 2:1 multiplexer, a mux for short, is a basic building block in many digital systems. The 2:1 mux shown in Figure 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**.

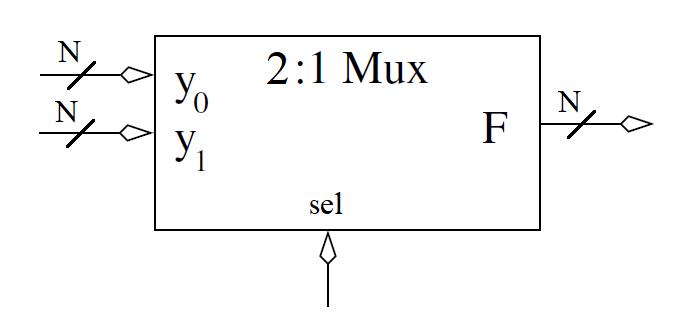


Figure 3: The schematic representation of a 2:1 mux.

You may notice that the data inputs of the 2:1 muxes in Figure 2 have their **y0** and **y1** data inputs denoted as ‘0’ and ‘1’ respectively. This is done to save space and increase clarity in the schematic.

I have provided you the Verilog code for a 2:1 mux 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

// Module definition for the 2:1 mux

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

// Module instantation for a 2:1 mux in the hiLow digital circuit

genericMux2x1 #(7) muxHex(7'b1111111, RandHex, randBut, randSeg);

Listing 1: Top, module declaration for a 2:1 mux. Bottom, module instantation of a 2:1 mux in Figure 2.

order shown in the module declaration shown in the top two lines in Listing 1.

The signal width, **N**, shown in Figure 3 is a placeholder for some integer value in an instantiation. I designed the 2:1 mux Verilog module so that you could specify the value of **N** using the #() specifier shown in the bottom line of code in Listing 1. A component that you can instantiate with different signal widths is called “generic”. This modifier is frequently included in the module’s name. The default value for the signal width is 8-bit. Be careful, Verilog will allow you to instantiate a genericMux2x1 module without the #() specifier, defaulting to 8-bit signals for **F**, **y0** , and **y0** and allowing you to provide signals with different vector sizes. It will do this with a warning that you can find in the Compilation Report tab -> Analysis & Synthesis folder -> Connectivity Checks folder. Click on the offending module and you will see the following error report. Not much, but you have been warned.

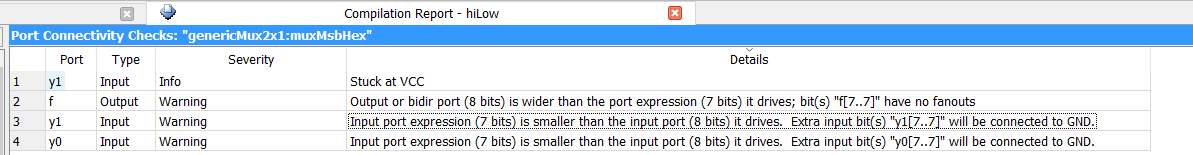


Figure 4: Warning that you messed up your vector sizes in a module instantiation.

# Compare Module:

A N-bit comparator is a basic building block in many digital systems. The N-bit comparator shown in Figure 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.

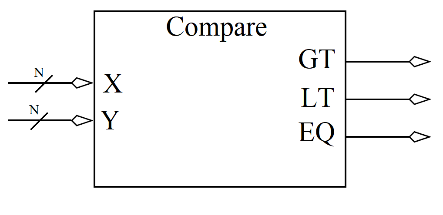


Figure 5: A schematic representation of a N-bit comparator.

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

I have provided you the Verilog code for the N-bit comparator 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 declaration shown in the top two lines in Listing 1.

Listing 2: Top, the module declaration for the comparator. Bottom, module instantiation of a comparator in Figure 2.

// 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 vector width of the **X** and **Y** inputs using the #() specifier. As an example, I’ve provided an instantiation from my hiLow module in the lower two lines in Listing 2. Like the mux, the default vector width for the inputs is 8-bits. Same warning about vector size mismatch applies to comparators.

# hexToSevenSeg Module:

# You should use the hexToSevenSeg module you developed in a previous lab. Note, the name of this module was shorted in Figure 2 to hex27 in order to save space and make the schematic more readable.

# 2:4 Decoder Module:

The module labeled 2:4 decoder interprets the 2-bit input s1, s0 as a 2-bit binary number that we will call **s**. All the y outputs whose subscript is less than or equal to **4-s** will have an output of 1. All the y outputs whose subscript is greater than **3-s** will have an output of 0.

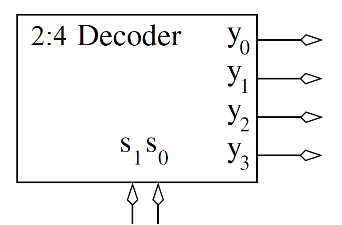


Figure 6: A 2:4 decoder.

The first few rows for the truth table for the 2:4 decoder are shown in Table 1.

Table 1: Partial truth table for the 2:4 decoder.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| s1 | s0 | y3 | y2 | y1 | y0 |
| 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”.

# hiLowWin:

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

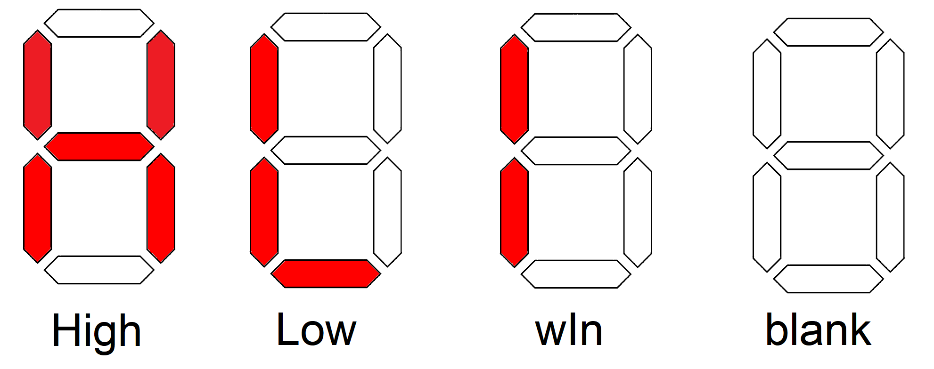


Figure 7: The illuminated patterns to inform the guesser about the magnitude of their guess.

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 3. Note that the code shown Listing 3 is incomplete.

Listing 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”.

# LFSR Module:

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 8.

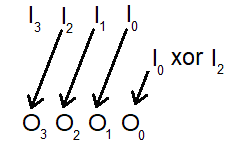


Figure 8: A schematic illustration of a 4-bit LFSR operation.

Figure 8 shows 3 of the 4 input bits I2 … I0 being shifted one bit to the left on their way to the outputs O3 … O1. The output O0 is formed by computing I2 ^ I0 where “^” is the xor operation.

Let’s use Table 2 to understand what happens if the input in Figure 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 7’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 2: The first iteration of the LFSR shown in Figure 8 when started at decimal 14.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| O3 | O2 | O1 | O0 | decimal |
| 1 | 1 | 1 | 0 | 14 |
| 1 | 1 | 0 | 1 | 13 |
|  |  |  |  |  |
|  |  |  |  |  |

If you continued the output from the shift operation performed in Table 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 8 is 15. This means that if Table 2 had 15 rows and you filled them all in, you would get 14 on the 14th row. Can you figure out what number is excluded from the sequence?

For the lfsr module, I want you 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 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).

# hiLow Module:

The hiLow module stiches together all the modules created and provided to you. The hiLow module declaration contains all the signals shown in Figure 2. I’ve provided my module declaration here so that you can more easily relate these to the pin assignments in the pin assignment section.

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

To complete this module, you will need to instantiate the modules in Figure 2. To see how to do this, I’ve grabbed the 2:1 mux out of the system architecture and reproduced it in Figure 9. Let’s write the Verilog code to instantiate this module in the hiLow module.

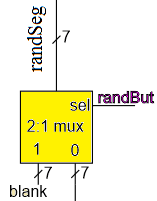


Figure 9: A small piece of hardware from Figure 2.

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 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, I named this signal **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 unique name. I called my instance **muxHex**,
* Put the system architecture signals in their corresponding locations in the module

genericMux2x1 #(7) muxHex(7'b1111111, RandHex, randBut, randSeg);

Once you get the hang of it, you are just translating the system architecture of Figure 2 into words.

For the hiLow module, I want you to:

* Use the module declaration:

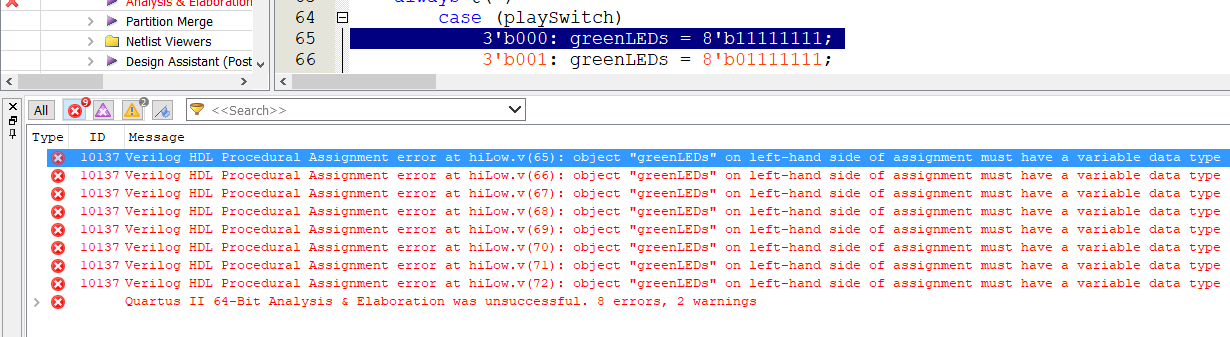
module hiLow(seedSwitch, playSwitch, guessSwitch, randBut, hiLowBut, randSeg, greenLEDs, hiLowSeg);

* Make the seedSwitch, playSwitch and 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 and hiLowBut inputs are not vectors.
* Use a vector for the randSeg output with wire type.
* Use a vector for the greenLEDs and hiLowSeg output with reg type.
* My module had 3 internal vectors (wire type) and 3 internal one bit signals (shown in the system architecture).
* When you instantiate a module
* 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
  + t\_seedSwitch unsigned green trace
  + t\_guessSwitch unsigned green trace
  + t\_playSwitch unsigned green trace
  + t\_randBut default green trace
  + t\_hiLowBut default green trace
  + LFSR output unsigned yellow
  + t\_randSeg hex red trace
  + t\_hiLowSeg hex red trace
  + t\_greenLEDs default red trace

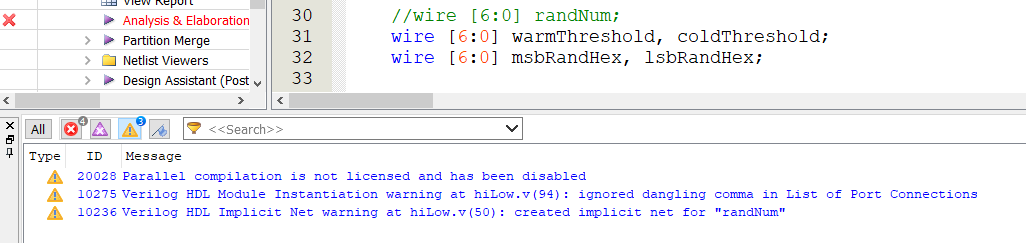
# Errors:

After I put together all the components, I ran Start Analysis & Elaboration. It took me a while to find and fix all my errors. I found that by clicking on the Error icon (red x) or Warning icon (yellow triangle) in the console area, I could eliminate a lot of the clutter and focus on the reporting.

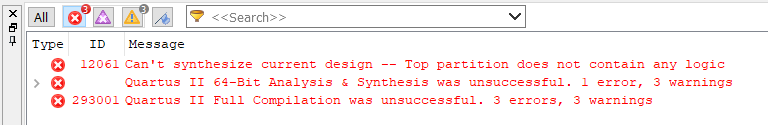
I defined output wire [7:0] greenLEDs; I should have used output reg [7:0] greenLEDs; because greenLEDs is the output of an always/case statement.



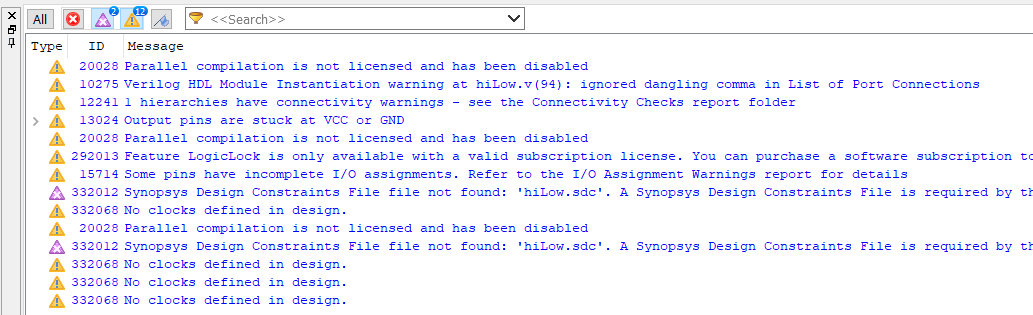
I forgot to include the declaration of randNum – actually I just commented it out to get this error. The top warning always appears and the second is a result of an unused output on my adder (more about this in the next lab).



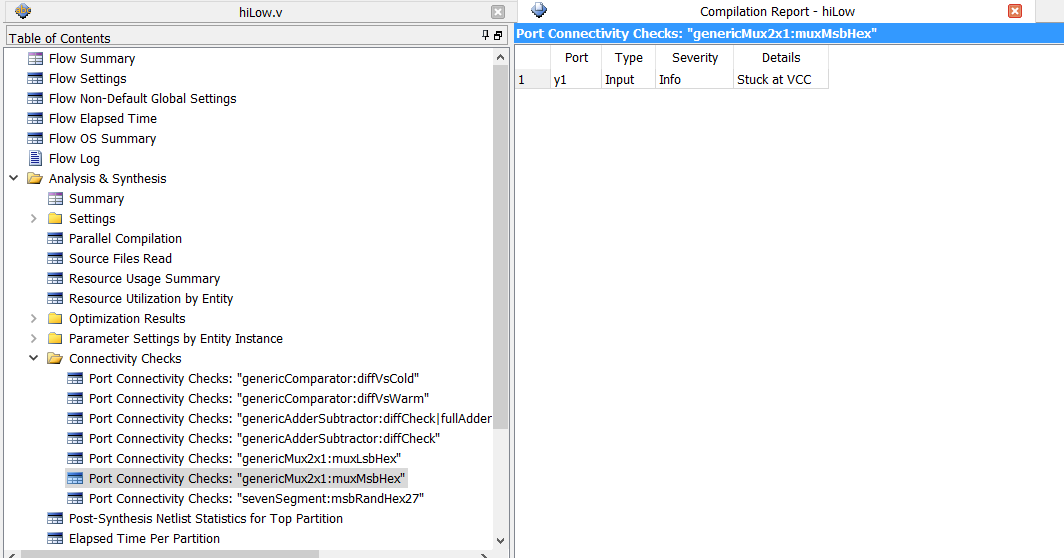
I accidently left a testbench as the top-level entity and tried to synthesize.



These are all the Critical Warnings and Warnings that I got on my 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. This report means that I hardwired one of the inputs to the 2:1 mux to all 1’s in order to blank out the 7-segment display until the button was pressed.



# Pin Assignment:

Use the image of the Cyclone V GX board in Figure 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. Note, this is where I had most of my errors.

|  |  |  |
| --- | --- | --- |
| 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] |
|  |  |  |  |

# 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.

# LFSR Module:

* Verilog code for the body of the module (courier 8-point font single spaced), leave out header comments.
* A completed Table 2.
* Complete the testbench for the lfsr module. Create timing diagram that asserts the four inputs listed in Table 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).

# hiLow Module:

* 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 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
  + t\_seedSwitch unsigned green trace
  + t\_guessSwitch unsigned green trace
  + t\_playSwitch unsigned green trace
  + t\_randBut default green trace
  + t\_hiLowBut default green trace
  + LFSR output unsigned yellow
  + t\_randSeg hex red trace
  + t\_hiLowSeg hex red trace
  + t\_greenLEDs default red trace

# Pin Assignment:

* Completed pin assignment table for all the signals in the hiLow module.