Lab EE 271: Circuit Theory

Winter 2018

***Lab 7***

*Shawnna Cabanday*

*Professor Rania Hussein*

*Lab Section:* Tuesday from 1:15 - 3:15

*Date submitted: March 7, 2018*

**Abstract**

The purpose of this lab was to fully design and implement a 271 final capstone project from a list of provided lab designs. I chose the mastermind game that simulates the multi-bit encryption locks for electronic locks.

**Introduction**

The purpose of this lab is to fully demonstrate how Systematic Verilog programming can be used to produce complex binary logic circuits virtually, which can then be practiced physically on the FPGA. A systematic verilog program will be completely designed based on simplified expressions and truth tables, which will be verified using generated waveforms by ModelSim and then downloaded to a Field Programmable Gate Array (FPGA) to demonstrate the physical program. The ultimate task for this lab was to design and build a mastermind game that simulated the multi-bit encryption locks used in many electronic locks. The system generates a random passcode consisting of a 4-digit number in base-4. For the purpose of my lab, I interpreted this as each digit only being able to be the numbers 0, 1, 2, or 3. The user will have to input a passcode in binary using switches in an attempt to match the randomly generated passcode. The user is given infinitely many tries, but with each guess the FPGA will display how many digits were correct and how many digits were misplaced. Additionally, instead of the FPGA displaying how many attempts were made, the code inputted into the machine will be displayed on the hexes.

# **Materials**

* Altera Quartus II Lite Program

## (5CSEMA5F31) DE1-SoC FPGA Development Board

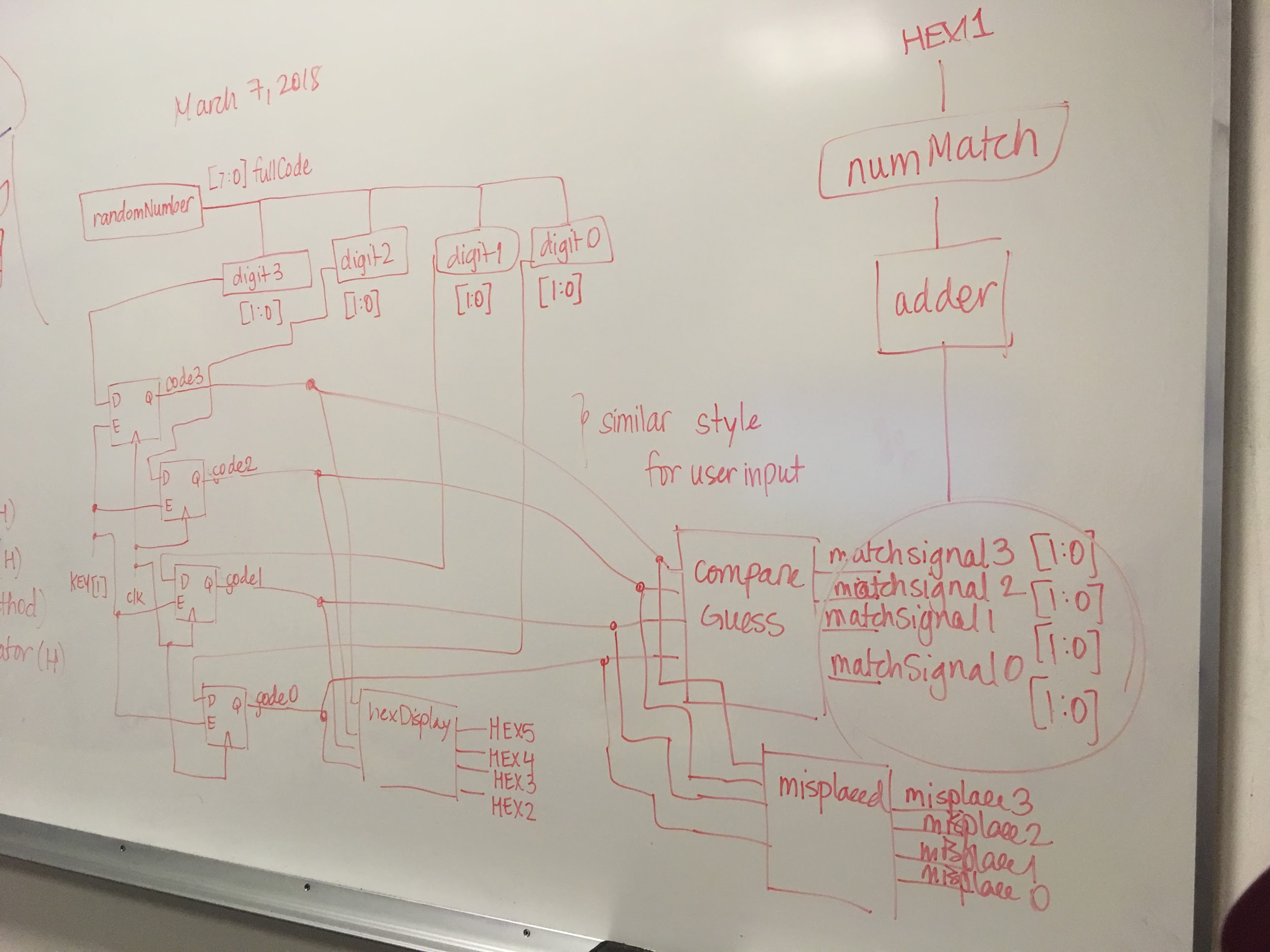
* Ethernet to USB Cable
* Power cord for FPGA

# 

# **Procedures**

## **Lab Objective: Design Problem -- Tug of War**

*1) Design (Written in previous lab report)*

Before creating the code, a block diagram of the connected FSMs was designed. Figure 1.1 illustrates the sketch of the block diagram for the main method. 

**Figure 1.1: Block Diagram Depicting Part of Final Mastermind Main Method**

Each decimal digit of the 4-digit number was broken down into separate two-bit binary numbers for the purpose of comparing each guessed digit with each randomized digit in the full 4-digit numbers.

As depicted in the diagram, there were several helper methods that were required. The following section will detail the purposes of each helper method used in the main **mastermind** method.

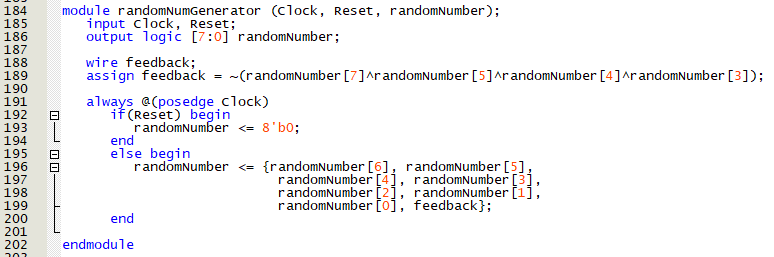
*2) Systematic Verilog Implementation*

7 main modules were created for this design problem. They include:

* **randomNumberGenerator:** Generates a random 8-bit binary passcode (4-digit decimal passcode)
* **Twobitreg:** loads and saves an inputted value for comparison purposes
* **hexDisplayDecoder:** loads inputted bit signals to output to a 7-segment HEX display
* **determineMisplace:** determines the number of misplaced binary digits when comparing the guess and the computer generated randomNumber
* **compareGuess:** determines the number of correctly placed binary digits when comparing the guess and the computer generated randomNumber
* **Mastermind:** main method that connects all helper methods together
* **checkWinner:** identifies whether or not the all numbers are matched and sends a signal to LED to turn it on

**“randomNumberGenerator” module**

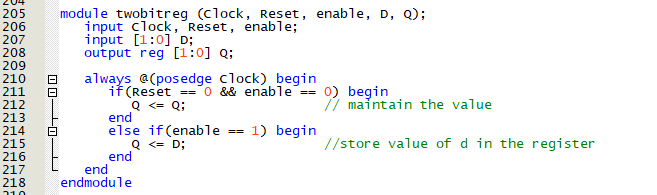
The randomNumberGenerator module was a helper method for the mastermind module to generate a random 8-bit random number. This 8-bit number was generated using an linear feedback shift register and was stored using the twobitreg module and then compared to another inputted binary number through the switches. The LFSR module doesn’t need any clock or reset inputs, although I did include them for the purpose of testbenches. The randomNumberGenerator outputs a 8-bit number that is split into 4 two-bit binary numbers, each called digit3, digit2, digit1, and digit0, respecitively. The stored and saved value takes the digit [3:0] values and then renames them to code3, code2, code1, and code0, for each decimal number respectively. I XOR’d randomNumber[7], randomNumber[5], randomNumber[4], randomNumber[3]. An output is given every clock cycle, so it is continuously running throughout. The clock is mapped to the clock from the main module, and “randomNumber” is mapped to the signal fullCode in the main mastermind module. Figure 1.2 depicts a sample of the randomNumberGenerator module.



**Figure 1.2: randomNumberGenerator Module Sample Code**

**“twobitreg” module**

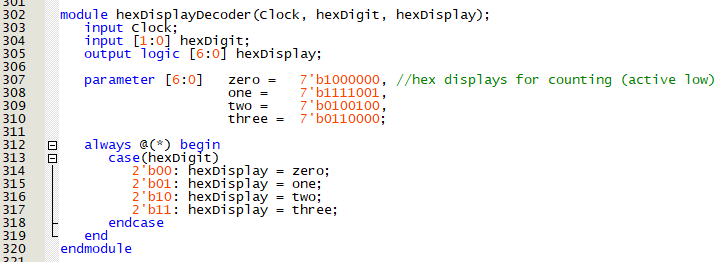
The twobitreg module was a helper method that was used to store two-bit binary values for comparison purposes. The module has inputs from the clock, reset, and had an enabler that controls when the twobitreg would store the value and continue to store it. This module was especially helpful for capturing and then storing the randomly generated number from the randomNumberGenerator module in order to use it for comparing with a user input. The twobitreg was also used to save user inputs entered into the machine. To store the user input, the enable button was KEY[0]. To store the randomly generated value, the enable button was KEY[1]. Figure 1.3 depicts a sample of the twobitreg module.



**Figure 1.3: twobitreg Module Sample Code**

**“hexDisplay” Module**

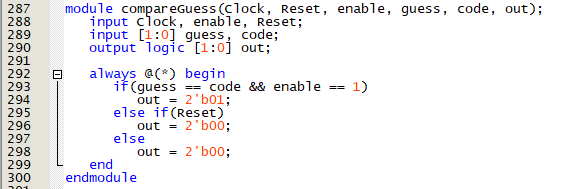
The hexDisplay module is the primary block for creating the output on the hex displays to illustrate the code inputted, the number of correctly matched digits, and the number of misplaced digits. The module takes in input signals from the twobitreg module to determine what state should be outputted. Those signals are connected with wires or regs to ensure that the correct state is transported to the hexDisplay module to determine what number should be displayed on the keyboard. The four hex displays on the farthest left are set to be controlled by the input switches and the remaining two hex displays to the right of the four hex displays are used to display the number of correctly matched and the number of misplaced, respectively. The hexDisplay module was instantiated for each decimal digit in 4-decimal guess and passcode. Figure 1.4 depicts a sample of the hexDisplay module.



**Figure 1.4: hexDisplay Module Sample Code**

**“compareGuess” module**

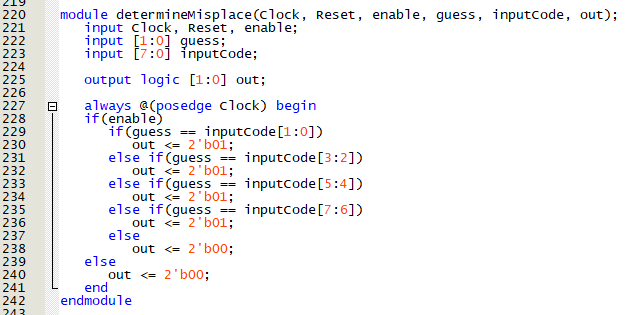
The compareGuess module takes in a two-bit binary input from a digit of the fullCode and another two-bit binary number from a digit in the same index from the guess passcode inputted from the switches. The two digits are then compared to one another based on an if statement, and if the result is true, a two-bit binary number named matchSignal is outputted with a decimal value of one (2’b01). The compareGuess module was instantiated for each decimal digit in the 8-bit binary guess code (user input) and randomly generated passcode. Each digit was named match\_signal3, match\_signal2, match\_signal1, and match\_signal0, respectively. The resulting two-bit matchSignals were summed together to determine the total number of matched numbers between the guess and the passcode. Figure 1.5 depicts a sample of the compareGuess module.



**Figure 1.5: compareGuess Module Sample Code**

**“determineMisplace” module**

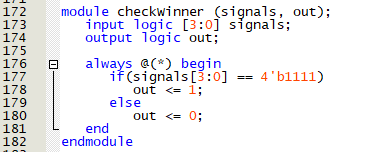
The determineMisplace module takes in signals from the output of the compareGuess module, a decimal number from the guess code, and a decimal number from the randomly generated number to determine the number of misplaced digits. Determining whether or not the number was misplaced required the matchSignal to be false on input. If matchSignal was true, then the digit did not need to be determined if it was misplaced because it was already indicated that it was in the correct spot. Figure 1.6 depicts a sample of the determineMisplace module.



**Figure 1.6: determineMisplace Module Sample Code**

**“checkWinner” module**

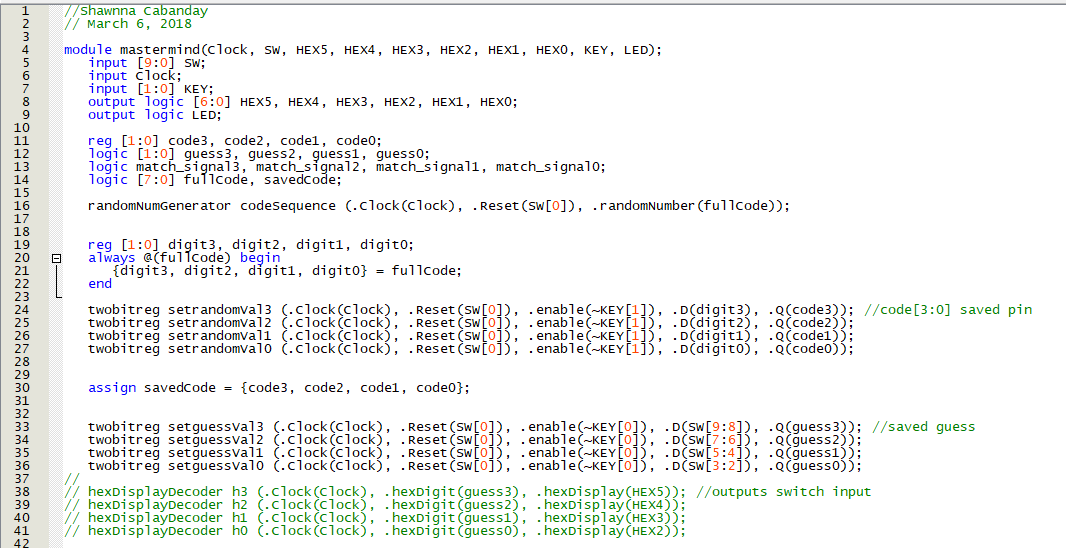
The checkWinner module receives input signals from the match\_signal wires which are connected to the output of compare module. The logic receives the input signals and identifies whether or not all of them are true for all four matching digits. If the result is true, the module outputs a signal that connects to a LED to turn it on an indicate that the code guess matches the generated passcode. Figure 1.7 depicts a sample of the mastermind module.



**Figure 1.7: checkWinner Module Sample Code**

**“mastermind” Module**

The main mastermind Module instantiates each of the described modules above to implement the mastermind simulator on the FPGA. Figure 1.8 depicts a sample of the mastermind module.

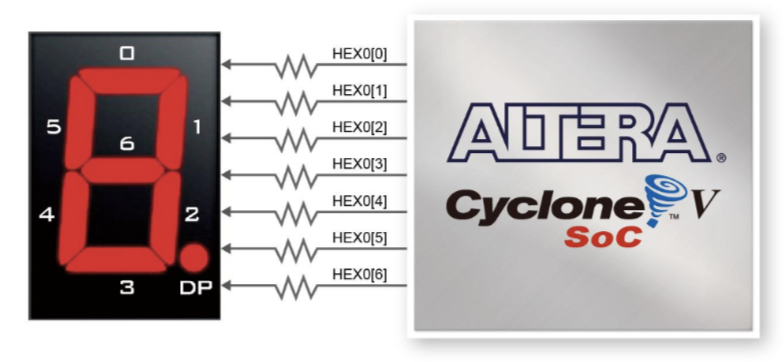


**Figure 1.8: mastermind Module Sample Code**

*3) ModelSim Generated Waveforms and Putting onto FPGA*

Most of the modules were simulated using test benches in ModelSim. Validity and accuracy of results were determined before any implementation on the FPGA. The design was then downloaded onto the FPGA using the module mastermind.

Input and output pins were mapped according to the DE1-SoC manual. Finally, the “.sof” file created from the Verilog script and pin assignment was downloaded to the FPGA by powering the board, connecting it to computer with a USB to Ethernet cable, and adding the file to the FPGA data platform. Table 1.1 provides part of the output pin assignments used for the hex displays. All hex displays were used. Finally, the size of the FSM was computed based on “LC Combinationals” and “LC Registers”.



**Figure 1.1: HEX Display Pin Assignment**

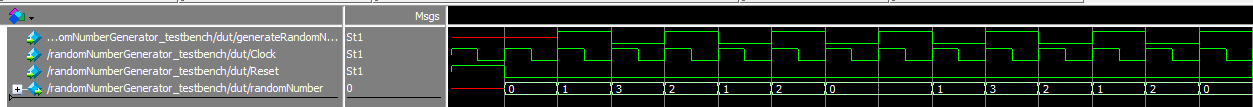
|  |  |  |
| --- | --- | --- |
| **Table 1.1: Output Pin Assignment** | | |
| Output | | Pin Placement |
| HEX0 Display | HEX0[0] | PIN\_AE26 |
| HEX0[1] | PIN\_AE27 |
| HEX0[2] | PIN\_AE28 |
| HEX0[3] | PIN\_AG27 |
| HEX0[4] | PIN\_AF28 |
| HEX0[5] | PIN\_AG28 |
| HEX0[6] | PIN\_AH28 |

# **Results and Analysis**

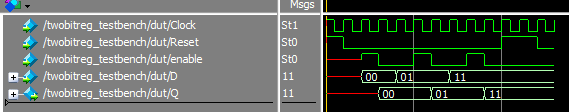
There were several logic errors that I encountered while conducting this lab. For the majority of the lab, syntax errors in systematic code required intense troubleshooting, but there was a significant amount of logic errors compared to previous labs. After troubleshooting, I found that a lot of the error was because I was using logic versus wire versus reg for a majority of variable statements. Also, for awhile I could not get my LFSR to randomly generate code and be able to save part of that code with my twobit register. It took me one sleepless night to realize that I needed an enabler on the register that would hold whenever the enabling signal was false and would store a new value when the enabling signal was true. Eventually, I was able to attain results that properly demonstrated the behavior of the correct logic for all modules. Figure 2.1 illustrates the generated waveforms for the randomNumberGenerator module. I was also able to successfully implement the logic for the twobitreg module (Figure 2.2). Figure 2.4, 2.5, 2.6, and 2.7 illustrate the generated waveforms for the hexDisplayDecoder, compareGuess, mastermind, and checkWinner module, respectively.

There are still existing issues that I did not feel the need to address, but could potentially fix in the future, when given extra time. In the second video of the demo, it appears that when I input the code by pressing and releasing the button, the two hexdisplays to the farthest right do not both reset to 0. This could be addressed by adding additional if-else statements in the hexdisplay module that will reset if the enable input button was 0. Additionally, the reset button does not work for the machine. To fix this, it requires that I add if else statements within an always block for modules that need to go to a reset state. Furthermore, I did not utilize a counter or limit the number of input tries for the user. To address this, I would need to add a separate functional state machine (FSM) module that counts the number of tries inputted and then in an if-else statement, if a certain number of tries was conducted, such as 7 seven tries, the entire machine should have reset. You would have different signals and wires instantiating from the counterModule and sends signals to the hexdisplays to reset them to 0, if need be.

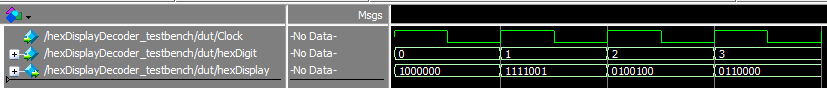
The number of the components was determined from the final report. Figure 2.6 illustrates the total number of resources utilized.



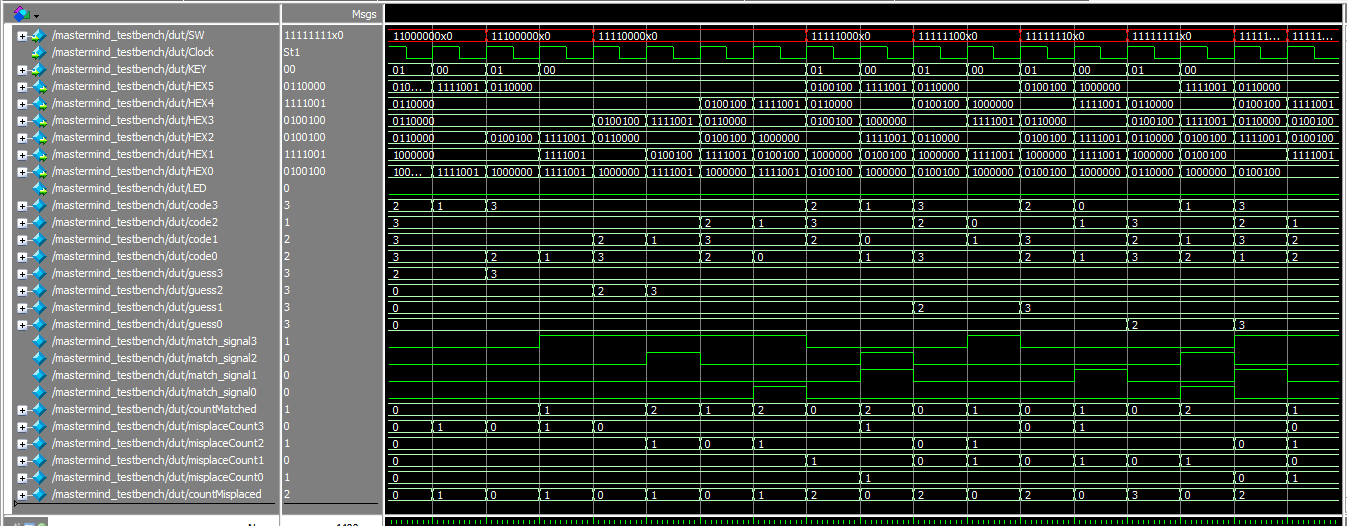
**Figure 2.1: ModelSim Waveforms for randomNumberGenerator Module**

****

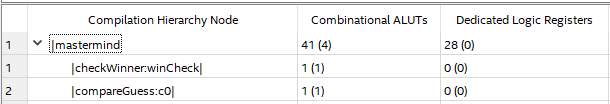
**Figure 2.2: ModelSim Waveforms for twobitreg Module**

****

**Figure 2.3: ModelSim Waveforms for hexDisplayDecoder Module**

****

**Figure 2.5: ModelSim Waveforms for mastermind Module**

****

**Figure 2.6: Resource Utilization**

# **Conclusion**

The use of computer aided simulations in Quartus and Verilog programming allows for increased efficiency in the steps and processes needed to develop and verify digital logic circuits on the FPGA. Debugging and determining the major sources for issues is a long process for all simulations and Verilog programming. Designs should always be taken in steps and with large projects such as these, they should be attempted early in the execution process. If I was given more time to perfect this project, additional components added to the code would be included.